Computers & Security 157 (2025) 104548

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

HGAN4VD: Leveraging Heterogeneous Graph Attention Networks for enhanced Vulnerability Detection

Yucheng Zhang [a], Xiaolin Ju [a,*], Xiang Chen [a,b,*], Misbahul Amin [a], Zilong Ren [a]

[a] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China
[b] State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Vulnerability detection
Code features
Graph simplification
Graph Attention Network

ABSTRACT

Detecting vulnerabilities is crucial for mitigating inherent risks in software systems. In recent years, there has been a significant increase in developing effective vulnerability detection approaches, many of which leverage deep learning technologies. These methods provide notable advantages, including automated feature extraction and the ability to train models autonomously, thereby improving the efficiency and accuracy of the detection process. However, existing methods encounter two significant limitations. Firstly, code analysis lacks granularity and does not fully leverage semantic and syntactic information within code structures, resulting in suboptimal performance. Secondly, approaches based on Graph Neural Networks (GNNs) inherently struggle to capture long-distance relationships between nodes in code structures. In this paper, we propose HGAN4VD, a novel vulnerability detection method that utilizes heterogeneous intermediate source code representations to address these limitations. HGAN4VD comprises two components: a heterogeneous code representation graph, which is constructed by creating diverse code representations and simplifying the graph to reduce node distances, and a Heterogeneous Graph Attention Network, which incorporates two attention layers to calculate node-level and semantic-level attention. Experiments on three widely used datasets demonstrate that HGAN4VD outperforms state-of-the-art methods by 1.5% to 7.7% in accuracy and 3.8% to 12.2% in F1 score metrics, affirming its effectiveness in learning global information for code graphs used in vulnerability detection. Furthermore, we demonstrate the generalization capability of our method on Java and Python datasets, suggesting its potential for broader applicability.

1. Introduction

Software vulnerabilities, known as bugs or weaknesses, present substantial risks to confidentiality, integrity, and data availability. We calculate the number of vulnerabilities over the past decade using data from the National Vulnerability Database (NVD) of the United States. As shown in Fig. 1, 37197 new Common Vulnerabilities and Exposures (CVEs) were recorded in 2024. This represents a 28% improvement compared to the 29066 vulnerabilities reported in 2023. For example, Apple disclosed a critical zero-day vulnerability (CVE-2023-41064) in ImageIO,[1] affecting millions of applications. Exploited through a zero-click method, this vulnerability enables attackers to install Pegasus spyware without requiring any user interaction, thus posing a significant threat. With the rapid evolution of information technology, vulnerability detection has become critical for safeguarding the security of software systems.

Traditional vulnerability detection methods mainly rely on static analysis (Yamaguchi et al., 2013; Du et al., 2019; Xu et al., 2017) and dynamic analysis (Li et al., 2017) techniques to identify potential vulnerabilities in software systems. The static analysis method relies on predetermined rules to detect vulnerabilities, effectively identifying vulnerabilities that conform to these established criteria. However, they often struggle with accuracy when dealing with highly complex code, particularly in cases of obfuscation or polymorphism, and may generate false positives or negatives due to their reliance on predefined patterns (Landman et al., 2017; Yamaguchi, 2017).

For languages such as C/C++, where the syntax and structure of code are well-defined, static analysis is particularly effective in identifying vulnerabilities. Nevertheless, its applicability to other programming languages, such as Python or Java, may be restricted by disparities in language features (e.g., dynamic typing, garbage collection).

* Corresponding authors at: School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China.
E-mail addresses: yc.zhang@stmail.ntu.edu.cn (Y. Zhang), ju.xl@ntu.edu.cn (X. Ju), xchencs@ntu.edu.cn (X. Chen), misbahul.amin.ai@gmail.com (M. Amin), Zilongren23@gmail.com (Z. Ren).

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=CVE-2023-41064

https://doi.org/10.1016/j.cose.2025.104548
Received 7 December 2024; Received in revised form 9 April 2025; Accepted 25 May 2025
Available online 14 June 2025

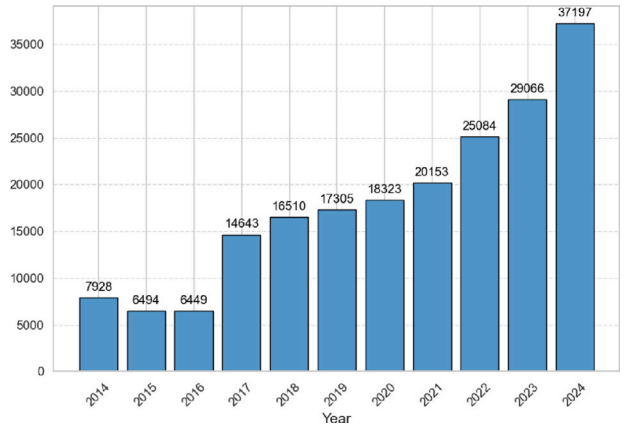

Fig. 1. Annual Distribution of CVEs.

Our method will be expanded to accommodate future multi-language projects, focusing on overcoming language-specific obstacles in feature extraction and graph construction.

Dynamic analysis method allows for real-time detection of vulnerabilities. However, they often have strict real-time performance requirements and may miss vulnerabilities that do not manifest during execution or depend on specific execution paths (Aggarwal and Jalote, 2006). A potent method for minimizing false positives and negatives has emerged: hybrid analysis, which integrates static and dynamic techniques. Hybrid analysis can offer a more comprehensive and precise vulnerability detection by capitalizing on the advantages of both methodologies. Although our current methodology emphasizes static analysis, future research will explore integrating hybrid techniques to enhance detection accuracy.

Previous studies have attempted to overcome these limitations through using machine learning and deep learning based methods (Backes et al., 2009; Shankar et al., 2001; Shar et al., 2014; Li et al., 2018; Russell et al., 2018; Li et al., 2021c; Dam et al., 2017; Wu et al., 2022, 2017; Guo et al., 2020a; Zhao et al., 2021; Li et al., 2021b; Tang et al., 2023b,a; Wang et al., 2023b; Tang et al., 2024). Machine learning-based methods (Backes et al., 2009; Shankar et al., 2001; Shar et al., 2014) learn the features of vulnerabilities using custom metrics, often performing poorly on complex datasets. Deep learning-based methods (Li et al., 2021a, 2018; Lin et al., 2017; Russell et al., 2018; Li et al., 2021c; Dam et al., 2017; Wu et al., 2022, 2017; Guo et al., 2020a; Zhao et al., 2021; Li et al., 2021b; Tang et al., 2023b,a; Wang et al., 2023b; Tang et al., 2024) train models by automatically evaluating metrics. Research has shown that deep learning-based methods, such as Bi-directional Long Short-Term Memory (BiLSTM) (Li et al., 2018), Convolutional Neural Network (CNN) (Guo et al., 2020a), Bi-directional Gated Recurrent Unit (BiGRU) (Li et al., 2021b), Bidirectional Recurrent Neural Networks (BiRNN) (Li et al., 2021b), can achieve better results in function-level vulnerability detection (Steenhoek et al., 2023; Cao et al., 2022). These algorithms rely on sequence mining of code features, demonstrating the feasibility of deep learning for code sequence learning. However, they ignore the syntax and semantic information in the code (Feng et al., 2020).

Recent research has primarily concentrated on the automatic learning of code representation, especially Graph deep learning, which can perfectly match graph learning by representing source code as intermediate graph structures. By parsing source code into Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG), complex hierarchical information in the source code can be captured more effectively (Wang et al., 2020; Wen et al., 2023; Wang et al., 2023a). Subsequently, widely adopted deep learning models are employed to detect vulnerabilities based on the node tokens derived from these graphs. Despite the promising results of these graph-based deep-learning approaches, several limitations remain.

(1) Limited representation of the code. Traditional methods typically parse code into graph structures and then utilize word2vec (Mikolov et al., 2013) to initialize node embeddings. However, these methods primarily focus on the shallow semantics of nodes, failing to distinguish between their semantic and syntactic information.

(2) Learning features unrelated to vulnerabilities. Function-level vulnerability detection encompasses the entire function, offering the advantage of capturing comprehensive vulnerability information. However, it also introduces numerous extraneous statements unrelated to vulnerabilities, which can lead to reduced detection accuracy.

(3) Inadequate models. Simply merging information from different edges results in a homograph in which all edges and nodes can only be treated as the same type, which causes the semantic and syntactic structures of various graphs to become entangled, making it difficult for models to distinguish between them.

To address the deficiencies above of the existing work, we propose HGAN4VD, a new source-oriented vulnerability detection framework based on heterogeneous graphs. The primary innovation of our method lies in the integration of heterogeneous information and meta-paths to construct a novel data structure, the Semantic Heterogeneous Information Network (SHIN). This structure employs graph attention networks to capture semantic relationships between graph nodes and their neighbors. Precisely, given a code snippet, we first extract multiple code structure graphs that contain features from different forms of code, including Abstract Syntax Trees (AST), Data Flow Graphs (DFG), Control Flow Graphs (CFG) and Program Dependency Graphs (PDG) (Ferrante et al., 1987). These graphs are integrated into heterogeneous graphs with multiple node types and edge types and then pruned according to certain rules. The purpose of this approach is twofold: to eliminate the influence of irrelevant factors on vulnerability detection efficiency and to reduce the distance between non-adjacent nodes. This addresses the long-distance dependency problem inherent in the GNN model (Zhou et al., 2020). Then, a semantic heterogeneous information network is constructed based on meta-paths and input into a Graph Attention Network to predict vulnerabilities in code fragments. Thus, HGAN4VD not only integrates rich code syntax and semantic information into a graph representation but also establishes a comprehensive semantic heterogeneous information network, allowing the model to understand deeper semantic information.

We evaluated the effectiveness of HGAN4VD on three datasets, the Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2022) and Schmidt et al. (2007), We compared HGAN4VD with five existing deep learning-based methods. The accuracy of HGAN4VD improved by 1.5% to 7.7%, and F1 score increased by 3.8% to 12.2%.

In summary, the main contributions of our work can be summarized as follows:

- We propose a novel vulnerability detection framework HGAN4VD, which employs an innovative method to obtain the fine-grained semantic information of code. This approach constructs a semantic heterogeneous information network, a multi-graph generated based on meta-paths, by aggregating the code information of multiple structure graphs.
- We design an encoder based on a two-layer heterogeneous graph attention network. This method encodes heterogeneous graphs using multi-faceted semantic contexts, enabling the generated code representation to capture more structural, syntactic, and semantic features of the code.
- We also implemented prototypes of HGAN4VD and evaluated its effectiveness using three widely recognized standard datasets. The experimental results demonstrate the efficacy of HGAN4VD in vulnerability detection. The code is available in the published repository.[2]

[2] https://github.com/VDHGANcode/VDHGAN

The rest of this paper is organized as follows. Section 2 provides background on code representation in vulnerability detection and deep learning models for vulnerability detection. Section 3 details the framework and specifics of our proposed method. Section 4 shows our empirical settings. Section 5 analyzes the findings related to the research questions. Section 6 discusses key aspects of our model beyond the main experiments, including hyper-parameter sensitivity, overfitting risks, cross-language generalization, method limitations, and its scalability and applicability. Section 7 summarizes related studies to our work and emphasizes the novelty of our research. Finally, Section 8 summarizes our work and shows potential future directions.

2. Research background

This section mainly introduces the background of advanced technologies exploited by HGAN4VD.

2.1. Graphical representation of source code

Graphical representation of code refers to transforming source code into visual representations that facilitate human or machine analysis, understanding, and manipulation. In our research, we extracted the Code Property Graph (CPG) (Yamaguchi et al., 2014) of a function, a multiple graph containing attributes such as AST, CFG, and PDG. We extract information on the following graph structures from CPG:

Abstract Syntax Tree (AST) (Cai et al., 2019) is an ordered tree representation of the abstract syntax of code. In the AST, each node represents the smallest lexical unit, and each edge signifies the parent–child relationship between nodes.

Control Flow Graph (CFG) is a graphical representation of code that illustrates all possible paths during execution. Nodes in CFG represent basic blocks, which can be statements or conditions. Edges in CFG represent control flow transitions through directed connections.

Program Dependence Graph (PDG) (Ferrante et al., 1987) is a program representation that generates data dependency relationships and explicitly controls dependencies. It includes two types of relationships: Data Dependency (DD) and Control Dependency (CD). Data dependency edges represent the def-use relationships, each labeled with a variable that is defined in the source node and used in the target node. Control dependency edges represent essential control flow relationships between predicates and statements.

Natural Code Sequence (NCS) (Wang et al., 2020) connects all leaf nodes of AST in natural order according to the source code's natural sequence. It reflects the programming logic of functions based on the order in which code appears in the function code. This solves the problem that information between leaf nodes in the AST cannot flow in the graph. However, this approach may also link semantically unrelated nodes, making the representation less ideal.

While our current implementation focuses on C/C++ code, the graphical representations used in our method (AST, CFG, PDG, etc.) are language-agnostic. They can be adapted to programming languages like Java and Python. However, language-specific features (e.g., dynamic typing in Python or garbage collection in Java) may require graph-construction adjustment. Future work will explore extending our method to multi-language projects, addressing these language-specific challenges.

Following previous studies (Wu et al., 2022; Zhou et al., 2019), we utilize the Joern tool (Yamaguchi et al., 2014) to extract various types of graphs and construct a heterogeneous graph with nodes representing statements in the code. In addition, we considered the type information of each node after parsing to enhance the graph's node representation. A visual example of a heterogeneous graph is shown in Fig. 2.

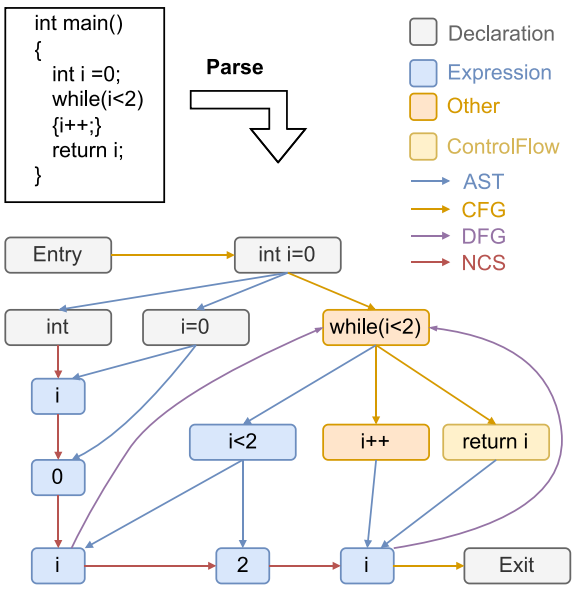

Fig. 2. Heterogeneous Graph representation of an Example Function.

2.2. Vulnerability detection based on graph

Early Machine Learning-based vulnerability detection methods used code features as input for vulnerability detection, such as sequence codes of different lengths (Shin and Williams, 2013; Wen et al., 2014). Deep Learning-based methods have been proven to generate features automatically (Guo et al., 2020b; Yu et al., 2020), so more and more Deep Learning-based methods (Russell et al., 2018; Li et al., 2018; Lin et al., 2019) are being applied in vulnerability detection. Due to the ability of graph structures to effectively represent the semantic information of code, several studies (Zhou et al., 2019; Cao et al., 2021; Wu et al., 2021) have begun to utilizing GNN to detect vulnerabilities. Graph Neural Networks (GNNs) (Zhou et al., 2020) are a class of neural networks aimed at solving graph-related tasks end-to-end. Unlike traditional neural networks that process data in a tabular or sequential format, GNNs can effectively capture and exploit the relational information encoded in graph structures.

Graph Convolutional Networks (GCNs) (Kipf and Welling, 2016) are among the earliest and most well-known types of GNNs. They extend the concept of convolutional neural networks to graphs, enabling the aggregation of information from neighboring nodes. The core concept involves aggregating a node's features with its neighbors by learning a functional map to update the node's representation. However, GCN primarily focuses on learning local features of nodes through neighborhood aggregation. Therefore, when the graph structure is too large, GCNs often struggle to capture long-range dependencies between nodes (Fu and Tantithamthavorn, 2022).

Graph attention network (GAT) (Veličković et al., 2017) combines the self-attention mechanism with a graph convolutional neural network for the first time, by calculating the attention coefficient between nodes as the weight of neighbor information aggregation, the importance of neighboring nodes can be distinguished. In addition, multiple independent attention mechanisms (i.e. multi-head attention mechanisms) are applied to calculate implicit states, and output representations are obtained through concatenation or averaging to stabilize the learning process.

Gated graph neural network(GGNN) (Li et al., 2015) has added a gated recurrent unit (GRU), which takes the information of neighboring nodes as input and the state of the nodes themselves as hidden states. This allows the model to selectively remember the hidden information of nodes and their neighbors, improving the long-term propagation ability of graph structure information.

2.3. Attention mechanism

Attention mechanism (Vaswani et al., 2017) is a powerful tool in machine learning, enabling models to focus on different parts of the input data selectively. There are two main types of attention mechanisms: self-attention and multi-head attention.

Self-attention, also known as intra-attention or internal attention, allows a model to attend to different positions of the input sequence to compute a representation of each position. It calculates the importance of each element in the sequence concerning the other elements, enabling the model to capture long-range dependencies and relationships within the input data.

Multi-head attention extends the idea of self-attention by performing multiple attention operations in parallel. In this mechanism, the input is transformed into multiple representations by applying different linear projections, and self-attention is applied independently to each of these representations. The results are then concatenated and linearly transformed to produce the final output. Multi-head attention allows the model to attend to different parts of the input differently, facilitating more prosperous and diverse representations.

Attention mechanism has been widely used in various machine learning tasks, including natural language processing (Bahdanau et al., 2014), computer vision (Yang, 2020), and graph-based learning (Wen et al., 2023; Wang et al., 2023a). They have demonstrated effectiveness in capturing complex patterns and dependencies in the data, leading to state-of-the-art performance in many applications.

2.4. Static vs. Dynamic analysis in vulnerability detection

In the context of software vulnerability detection, static and dynamic analyses are two widely used paradigms, each with distinct advantages and limitations.

Static analysis techniques analyze the source code without executing the program. These approaches are efficient and scalable for large codebases and can detect potential vulnerabilities before software deployment. However, static methods often suffer from false positives due to their limited ability to reason about runtime behavior, such as execution paths dependent on input values or runtime environments.

Dynamic analysis, in contrast, inspects the behavior of software during execution. It is more effective at detecting vulnerabilities manifest only at runtime, such as memory corruption, race conditions, or logic flaws dependent on specific inputs. However, dynamic methods face challenges such as path explosion, high performance overhead, and incomplete coverage due to limited test cases or execution traces.

Hybrid analysis combines static and dynamic techniques to leverage their complementary strengths. While hybrid approaches can significantly reduce false positives and negatives, they usually have higher implementation complexity and resource demands.

In this work, HGAN4VD primarily focuses on static analysis to ensure scalability and efficiency in processing large-scale open-source C/C++ codebases. By leveraging static representations like AST, CFG, DFG, and PDG, our model captures syntactic and semantic patterns in source code for vulnerability prediction. However, we acknowledge that static analysis alone may miss vulnerabilities dependent on dynamic execution contexts. Addressing this gap is part of our planned future research, which will explore integrating runtime and hybrid analysis techniques to improve detection accuracy further.

3. Our approach

In this section, we introduce the detailed architecture of HGAN4VD. As shown in Fig. 3, HGAN4VD consists of three parts: Generate Graphical representation of code, Heterogeneous graph attention network module, and Vulnerability Detection module. Next, we will provide detailed explanations for each part.

3.1. Constructing heterogeneous graph

Data Preprocessing. Specifically, for a source code dataset $D = \{f_1, f_2, \ldots, f_n\}$, we perform vulnerability detection on each function-level code snippet f_i.

Graph Generation. In our study, we processed each code snippet f_i in the following steps: (1) Extraction of the code structure graph, which obtains the structural representation of the function through a code parser. (2) The initialization encoding of nodes, which obtains the features of each statement through a pre-trained Word2Vec model. Currently, functional vulnerability detection methods based on deep learning mostly use Abstract Syntax Trees(AST) as code representation. AST is a tree representation used to describe the syntax structure of program code. However, AST lacks program control information and data dependency information, which can be filled by the Data Flow Graphs(DFG) and the Control Flow Graphs(CFG). Following previous works (Wu et al., 2022; Siow et al., 2022), all types of graphs were extracted using the Joern tool (Yamaguchi et al., 2014).

Specifically, utilizing the Joern parsing function allows us to extract a set of nodes and edges $G = \{V, E\}$, where each node $v = \{id, statement, ntype\} \in V$ represents a statement and records its ID, statement, and type of that node, each edge $e = \{src, dst, etype\} \in E$ records the starting node, destination node, and edge type of that edge. Referring to the Devign's method (Zhou et al., 2019), we stored code statements corresponding to node attributes through dictionary data types and recorded node types corresponding to node information through node mapping tables, constructing a comprehensive heterogeneous graph that includes different node types and edge types. Following the steps above, the graph is simplified to eliminate redundant information. Subsequently, the graph is transformed into an adjacency matrix based on the starting nodes of the edges within the graph.

The node set $V = \{v_1, v_2, \ldots, v_n\}$ contains a set of type nodes. Each node $v \in V$ has the attribute $ntype \in \{Sizeof, Identifier, ReturnStatement \cdots\}$ indicating its type. According to statistics, 69 types of $ntype$ are in the parsed node set V. We categorize these node types into four parent types based on their syntax. For the type of each node, we consider parent types $S_v \in \{Identifier, Expression, Declaration, Other\}$. For each directed edge $e = (src, dst, etype) \in E$, it and $Te \subset \{AST, CFG, DFG, PDG\}$ represent the origin of the connection from node v_{src} to node v_{dst}. Specifically, we replace the $ntype$ of each node v with its parent type and retain its original $ntype$ with a string $S_v$for subsequent node vectorization.

Initialize node representation. In addition to the structural information of functions, the semantic information of code statements is also important. Since the attributes of nodes are textual representations of code, and text belongs to the character type, it cannot be directly used as node attributes for model computation. Therefore, it is necessary to prioritize converting textual information into node feature vectors. Encoding statements in a function using pre-trained models is a standard method in the programming language domain (Zhou et al., 2019; Wen et al., 2023). We convert the statement of nodes into quantifiable vectors and utilize them as the initial features for nodes. Initially, HGAN4VD employs a lexical analyzer to acquire basic tokens from the code of nodes. After obtaining the vulnerability heterogeneity graph, considering that the cause of the vulnerability is incorrect code logic or incomplete consideration of the impact of data before and after, it is not significantly related to the naming of variables and functions (Li et al., 2019). The function and variable names in the token are mapped to the form of symbol names $FUNx$ and $VARx$, where x represents the order in which the functions and variables appear. The purpose of doing this is to prevent them from interfering with the initial functionality of nodes, as different programmers defining function and variable names can bring some text noise, and symbolic processing can improve the ability to obtain common features of vulnerabilities. Subsequently, HGAN4VD uses a pre-trained Word2Vec model (Mikolov

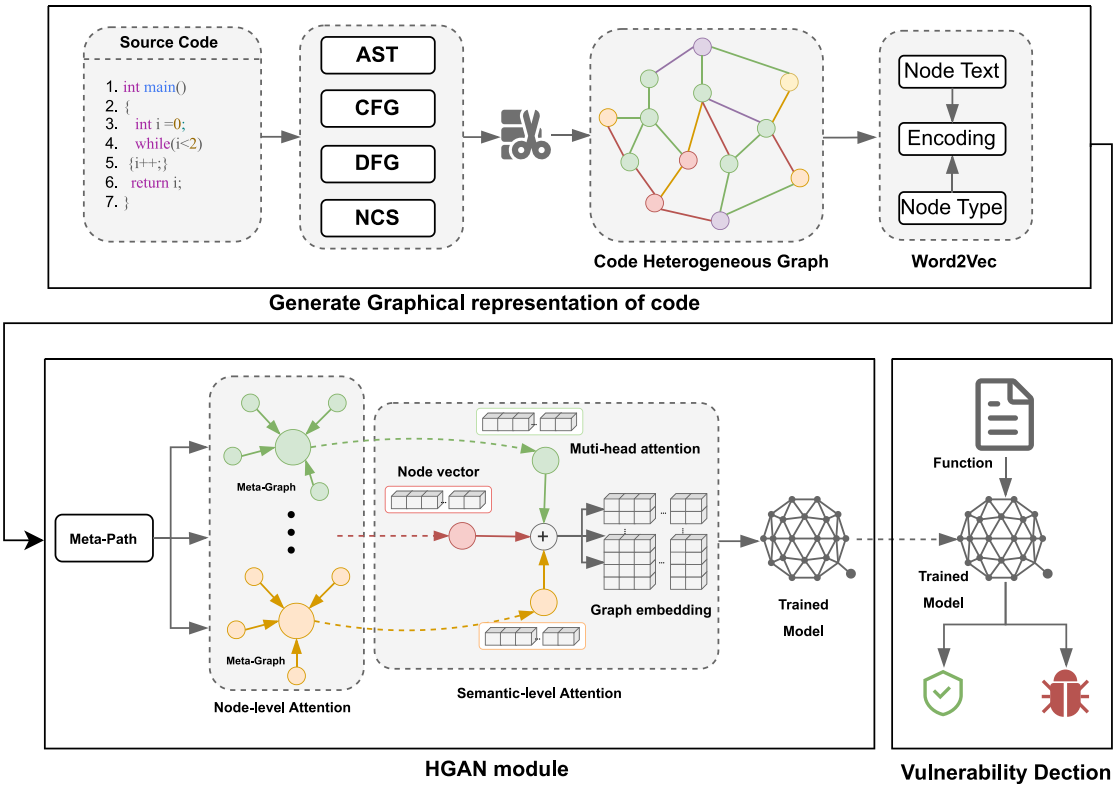

Fig. 3. Overall Framework of Proposed Method HGAN4VD.

et al., 2013) to obtain the primary embedding for each node. The corpus for the pre-trained word embedding model is constructed using mapped tokens from all training samples, with the token dimensions constrained to fewer than 100 to reduce computational costs. Finally, to capture hidden information about the feature types of nodes, we encode each type as an integer and concatenate the encoding of the node type with the obtained node embedding as the feature representation for the node. We represent the statement of each node as x_i, and the representation of that node $h_i = model(x_i)$, where $model$ represents the mathematical expression of the pre-trained model. The final node embedding of the graph can be represented as $H_{node} = \{h_1, h_2, \ldots, h_N\}$.

3.2. Simplification of heterogeneous graphs

The initial code structure graph extracted via the Joern (Yamaguchi et al., 2014) contains a significant amount of redundant information, increasing the number of nodes and edges. In heterogeneous graph neural networks (HGNNs), redundant or noisy nodes and edges can negatively impact model efficiency and accuracy. Prior studies (Liang et al., 2022; Liu et al., 2022) have shown that excessive structural complexity in graphs can lead to several issues: (1) increasing the distance between semantically related nodes, making it harder for message-passing mechanisms to capture meaningful relationships; (2) introducing noise in message propagation, thereby diluting important vulnerability-related patterns; and (3) unnecessarily increasing computational complexity, which hinders scalability in large codebases. To address these challenges, we designed two graph simplification strategies, HGS1 and HGS2, to systematically reduce redundancy in the fused heterogeneous graph while preserving critical structural and semantic information.

Firstly, we consider merging nodes with duplicate information. When constructing the code structure graph, Joern (Yamaguchi et al., 2014) categorizes nodes into various types. For example, Expression-type nodes are classified into subtypes like $AndExpression$, $SizeofExpression$, $AssignmentExpression$, and $others$. Although this approach enhances semantic information, it also increases the size of the graph. As shown in Fig. 2, the variable "i" appears redundantly in the child nodes of both "int $i = 0$" and "$i = 0$", with "int $i = 0$" serving as the parent node for "$i = 0$". Consequently, in such cases, we retain the node "i" closer to the parent node and eliminate the node "i" farther away.

Secondly, if node i is the only parent node of node j and their values are the same, this will result in a longer distance between node i and the child nodes of node j. Since the values of nodes v and j are the same, they can be merged to shorten the distance between nodes.

The simplification process is outlined in Algorithm 1 and Algorithm 2, where the simplification processing in two strategies, respectively. For HGS1, the algorithm performs breadth-first traversal on the input graph. Whenever two nodes v and j have the same value and share a common parent node, the edges between v and j and node j are deleted, and the type of v is changed to the parent node type(lines 10–12). Similarly, for HGS2, when nodes v and j share the same value and v is the sole parent node of j, the two nodes are merged (lines 9–10). HGS1 removes isolated nodes and prunes weakly connected structures that contribute minimally to vulnerability detection, ensuring that retained subgraphs focus on relevant code semantics. HGS2 further refines the graph by reducing edge redundancy and optimizing node connectivity, enhancing the effectiveness of subsequent graph-based learning. Similar graph simplification techniques have been explored in recent studies (Xu et al., 2020; Wen et al., 2023; Ba et al., 2025), demonstrating their potential to improve model robustness and computational efficiency in deep learning-based vulnerability detection tasks.

3.3. Meta-graph generation

In vulnerability detection, generating code structure graphs is essential for capturing key structures and dependencies in source code to identify potential security vulnerabilities. To enhance this process, we propose meta-graph generation based on meta-paths (Wang et al., 2019) to better represent the complexity of code structures.

Algorithm 1 Heterogeneous Graph Simplification Strategy 1 (HGS1)

Input: Original Code Structure Graphs: $G = \{g_1, g_2, \cdots, g_n\}$
Output: Simplified Code Structure Graphs: G_1
Function: $HGS1$

$G_1 \leftarrow \varnothing$
for each g_i in G **do**
  //do breadth-first traversal
  $Stack \leftarrow \varnothing$
  Push $g_i.root$ into $Stack$;
  while $Stack \neq \varnothing$ **do**
    $v \leftarrow$ Pop Stack
    for $j \in v.children$ **do**
      if $v.value = j.value$ and $v.type, j.type$ have the same parent $NodeType$ **then**
        Remove edge $< v, j >$ from g_i;
        Remove node j from g_i;
        $v.type \leftarrow v.type.ParentNodeType$
        Push $v.children$ into $Stack$;
      else
        Push j into $Stack$;
      end if
    end for
  end while
  $G_1 \leftarrow g_i$;
end for
return G_1

Algorithm 2 Heterogeneous Graph Simplification Strategy 2 (HGS2)

Input: Original Code Structure Graphs: $G = \{g_1, g_2, \cdots, g_n\}$
Output: Simplified Code Structure Graphs: G_2
Function: $HGS2$

$G_2 \leftarrow \varnothing$
for each g_i in G **do**
  $Stack \leftarrow \varnothing$
  Push $g_i.root$ into $Stack$;
  while $Stack \neq \varnothing$ **do**
    $v \leftarrow$ Pop Stack
    for $j \in v.children$ **do**
      if $v.value = j.value$ and $j.num\_children = 1$ **then**
        Remove edge $< v, j >$ from g_i;
        Remove node j from g_i;
        Push $v.children$ into $Stack$;
      else
        Push j into $Stack$;
      end if
    end for
  end while
  $G_2 \leftarrow g_i$;
end for
return G_2

In our approach, a meta-path is defined as an ordered sequence of different types of nodes, illustrating their semantic relationships. A meta-path θ is a sequence in the form of $\theta = N_1 \xrightarrow{R_1} N_2 \xrightarrow{R_2} \cdots \xrightarrow{R_n} N_{n+1}$, where N_i represents a state and R_i embodies a composite relationship between N_i and N_{i+1} (Liu et al., 2021). Therefore, the composite relationship between N_1 and N_{i+1} can be expressed as $R = R_1 \circ R_2 \circ \cdots \circ R_i$, where R represents the composite operator of different relationships. The meta-graph is a sub-graph pattern that describes the relationship between a target node and contextual nodes. We construct a meta-graph structure by defining a meta-path, such as $E \rightarrow I \rightarrow E$ representing a fixed semantic meta-path and $E \rightarrow D \rightarrow E$ representing another fixed semantic meta-path, where E, I, and D represent three different parent types of nodes (Expression, Identifier, Declaration). The auxiliary node I and D illustrate the sharing relationship between the target node E_1 and the context node E_2, where E_1 and E_2 represent two types of nodes with the Expression type. Each meta-path represents a distinct pattern of structure and dependency relationships, aiding in the more precise capture of semantic information within the code.

We define the set of all meta-graphs generated based on meta-paths as Semantic Heterogeneous Information Networks (SHINs). We formally define the SHIN as follows: A Semantic Heterogeneous information network SHIN $= \{G_1, G_2, \ldots, G_n\}$ is a set of meta-graphs generated based on predefined meta-paths, where n represents the number of meta-paths. Each meta-graph $G = (V, E, H)$ with node type mapping: $V : t \mapsto \mathcal{T}$ where $V = v_1, \ldots, v_N$ is the set of N nodes, E is the set of edges, $\mathcal{T}$ is the set of node types, and $H \in \mathbb{R}^{N \times D}$ is the feature matrix describing all nodes, where N represents the number of nodes, and D represents the node feature dimension initialized by the pre-trained model. A meta-graph is a sub-graph pattern describing the relationship between a pair of target and context nodes In contrast to existing meta-graph definitions (Wang et al., 2019), we explicitly specify four parent types of nodes to connect auxiliary nodes, and on this basis, up to 12 meta-paths (EDE, ECE, EOE, etc.) can be formed.

Unlike traditional GNN-based methods that primarily focus on local node information, meta-paths enable the capture of higher-order relationships between nodes. A meta-graph generated by the meta-path of length 2 can account for multiple node relationships to capture relationships between more distant nodes. This enables the deep learning model to capture more complex, higher-order relationships, thereby improving the performance of vulnerability detection tasks. Furthermore, using custom meta-paths provides scalability and adaptability, making them suitable for large graphs by focusing on specific, user-defined node relationships. By employing meta-path rules of length 2, we mitigate the vanishing gradient problem, improve the structural integrity of the graph, and reduce computational costs compared to methods that consider all possible paths within the graph.

3.4. Heterogeneous graph convolution module

For the heterogeneous graph representation SHIN of each code snippet, we introduce a graph neural network with two attention layers to learn the semantic and structural information of the code from SHIN. Firstly, there is a node-level attention layer where nodes in each meta-graph aggregate information from their neighboring nodes along the edges and utilize it to update the feature representation of nodes. HGAN4VD performs the node learning process based on meta-graphs extracted under different meta-paths. Subsequently, a vector attention-weighted aggregation is conducted on the vectors aggregated from each meta-path, followed by a downstream propagation.

We represent the initial representation of node V_i in meta-graph G_r as $h^0_{i,r}$. Therefore, the representation of node V_i in t state is $h^t_{i,r}$, while $h^{t+1}_{i,r}$ represents the process of clustering along the edges in the meta-graph G_r in $t+1$ state.

$$h^{t+1}_{i,r} = \text{aggregator}\left(h^t_{j,r}\right), \forall v_j \in N_{i,r} \tag{1}$$

where $N_{i,r}$ refers to the neighbors of node v_i in meta-graph G_r.

Node-level Attention layer. Given a meta-path, each node is associated with multiple neighbors determined by the meta-path. Distinguishing the differences among these neighbors and selecting the most informative ones is a challenge. Node-level attention measures the importance of these neighbors and allocates different levels of attention to them.

In updating the feature representation of nodes in each meta-graph, we employs attention mechanisms to distinguish the influence of neighboring nodes. Precisely, the correlation coefficient e is first calculated

between the nodes in graph G and their immediate neighbors. Subsequently, the attention coefficient a_i is computed for each neighbor relative to target node v_i.

$$e_{i,j}^{r} = m\left(\left[W h_{i,r}^{t} \middle\| W h_{j,r}^{t}\right]\right),\ j \in N_{i,r}\ , \tag{2}$$

$$a_{\mathrm{i,j}}^{r} = \mathrm{softmax_j}\left(\mathrm{e}_{\mathrm{i,j}}^{r}\right) = \frac{\exp\left(\sigma\left(\mathbf{a}_{r}^{\mathrm{T}} \cdot \left[\mathbf{h}_{\mathrm{i}}' \parallel \mathbf{h}_{\mathrm{j}}'\right]\right)\right)}{\sum_{\mathrm{k}\in\mathcal{N}_{\mathrm{i}}^{r}} \exp\left(\sigma\left(\mathbf{a}_{r}^{\mathrm{T}} \cdot [\mathbf{h}_{\mathrm{i}}' \parallel \mathbf{h}_{\mathrm{k}}']\right)\right)}\ , \tag{3}$$

where m represents the transformation matrix operation, which can map different types of node features to the same feature space, $\|$ represent cascading operation, $\sigma(\cdot)$ represents the activation function, $e_{i,j}^{r}$ represents the importance of node v_j to node v_i in the meta-graph r.

After obtaining the attention coefficients, a linear transformation is applied to the initial node representations, and the node representations are then updated by combining the attention coefficients. Given the scale-free properties of heterogeneous graphs, the graph data exhibits significant variability. To improve the stability of the training process, we extend node-level attention to multi-head attention.

$$h_{i,r}^{t+1} = \|_{k=1}^{K}\ \sigma\left(\sum_{j\in N_{i,r}} a_{ij}^{r^k} W^k h_{j,r}^{t}\right), \tag{4}$$

where $a_{i,j}^{k}$ denotes the kth head of $N_{i,r}$. W^k corresponds to the kth head of W, $\|_{k=1}^{K}$ represents the concatenation of results from K different attention heads.

Semantic-level Attention layer. In heterogeneous graphs, different meta-paths convey distinct semantic information. The primary challenge lies in selecting the most meaningful meta-paths and integrating these diverse semantic details. To address this, semantic-level attention is employed to evaluate the significance of various meta-paths and assign appropriate attention weights to each. The weight of each meta-path is computed by first applying a non-linear transformation to the node-level embeddings, followed by calculating the weight using the formula provided.

$$\mathrm{w_r} = \frac{1}{|\mathrm{V}|}\sum_{\mathrm{i}\in\mathrm{V}} \mathrm{q}^{\mathrm{T}} \cdot \tanh\left(\mathrm{W}\cdot \mathrm{h_{i,r}} + \mathrm{b}\right), \tag{5}$$

where V represents the set of all nodes, and q, W, and b are trainable model parameters. Softmax normalization is applied to the above results as semantic-level attention. After obtaining the weight coefficients, all node-level embeddings can be fused to obtain the final embedding h_i^r.

$$\beta_r = \frac{\exp\left(\mathrm{w}_r\right)}{\sum_{k=1}^{R}\exp\left(\mathrm{w}_k\right)}, \tag{6}$$

$$h_i' = \sum_{r=1}^{R} \beta_r \cdot h_{i,r}, \tag{7}$$

where β_r represents the weight of the generated meta-graph G_r under the meta-path r, R represents the number of meta-paths. A more significant value of β_r indicates a higher importance of the corresponding meta-path.

Classifier layer. In this stage, HGAN4VD extracts node features from the graph to obtain a feature representation of the function. Each node in the graph represents a basic block that encapsulates syntactic and semantic information. Therefore, HGAN4VD retrieves the function's feature by computing the average of all nodes' features in the graph.

$$H = \frac{1}{|V|}\sum_{i\in V} h_i', \tag{8}$$

where H represents the feature representation of the sample function.

HGAN4VD conducts graph-level classification to ascertain the vulnerability of a function. It accepts the function's feature representation as input and teaches a classifier to indicate whether the function is vulnerable. The classifier extracts additional abstract features at the function level by utilizing a linear transformation on the feature representation. To improve the extraction of function-level features, the proposed approach uses a multilayer perceptron (MLP) and applies the sigmoid function for classification.

$$\overline{y} = \mathrm{Sigmod(MLP}(H)), \tag{9}$$

where $\overline{y}$ is the final binary classification result of two dimensions. The first dimension represents the probability of non-vulnerability in the result, and the second represents the probability of vulnerability. Finally, the model takes the higher probability of both as the final output of vulnerability classification.

4. Experimental setup

This section provides a comprehensive examination of the evaluation process for HGAN4VD. The evaluation is structured to ensure a thorough and transparent analysis of the tool's performance and contributions to the field. Firstly, we outlined the research questions along with their underlying motivation. Next, we introduce the dataset used in the experiment and outline the data preprocessing steps. Subsequently, we describe the latest state-of-the-art baseline methods and compare their performance against our proposed model. Finally, we present a comprehensive overview of the experimental setup used for the evaluation. This includes a detailed description of the hardware and software configurations, parameter settings, evaluation metrics, and any other relevant experimental details.

4.1. Research questions

Our empirical study aims to answer the following research questions.

- **RQ1: How efficient is our proposed method, HGAN4VD, in detecting vulnerabilities?**
  This research question aims to assess the performance of HGAN4VD by conducting a comparative analysis against five baseline methods.
- **RQ2: How does each module contribute to the performance of HGAN4VD?**
  We proposed two graph simplification algorithms, HGS1 and HGS2, to eliminate redundant nodes from code structure graphs. In addition, we effectively capture semantic and syntactic information in heterogeneous graphs using a two-layer attention network (HGAN). Therefore, the research question was raised to investigate the impact of the graph simplification module and HGAN module on the performance of HGAN4VD.
- **RQ3: How does the selection of different meta-paths influence the performance of HGAN4VD?**
  The choice of meta-paths plays a crucial role in shaping the performance of HGAN4VD, as they define the semantic relationships captured within the heterogeneous graph representation of the code. Therefore, this research question set out to assess how different selections of meta-paths impact the effectiveness of HGAN4VD.
- **RQ4: How robust is HGAN4VD when trained on small or imbalanced datasets?**
  This research question investigates whether HGAN4VD is prone to overfitting when trained on limited or imbalanced data, by evaluating its performance under varying training set sizes and class distributions.

Table 1
Datasets statistics.

Dataset	Samples	#Vul	#Non-Vul	Vul Ratio
Devgin	22 361	10 067	12 294	45.02%
Reveal	18 169	1664	16 505	9.16%
Fan et al.	179 299	10 547	168 752	5.88%

4.2. Experimental subject

To investigate the effectiveness of HGAN4VD, we adopt three vulnerability datasets in our study, including Devign (Zhou et al., 2019), Reveal (Wei et al., 2020), and Schmidt et al. (2007). However, in the field of vulnerability detection, other publicly available datasets can also serve as alternative solutions, such as SARD (Li et al., 2021b), Juliet C/C++ (Black and Black, 2018), CodeXGLUE (Lu et al., 2021), etc. Given our focus on analyzing real-world project datasets, we opted not to use the three previously mentioned datasets containing artificially synthesized data. However, we acknowledge the existence of other datasets and encourage their exploration in future research endeavors.

The FFmpeg + QEMU dataset, provided by Devign, is manually labeled and derived from two open-source C projects. It comprises approximately 10,000 vulnerable entries and 12,000 non-vulnerable entries. On the other hand, the Reveal dataset is collected from two open-source projects: the Linux Debian Kernel and Chromium. This dataset includes around 2,000 vulnerable entries and 20,000 non-vulnerable entries. Additionally, Fan et al. curated a dataset from over 300 open-source C/C++ GitHub projects, encompassing 91 distinct vulnerability types recorded in the Common Vulnerabilities and Exposures (CVE) database from 2002 to 2019. The dataset consists of approximately 10k vulnerable entries and 177k non-vulnerable entries. We also performed three filtering steps on the above three datasets: (1) deleting functions with abnormal truncation; (2) deleting functions that failed to be parsed by the Joern tool (Yamaguchi et al., 2014); (3) deleting functions with more than 500 nodes after parsing to avoid noise. The statistical data of each experimental dataset after processing are shown in Table 1.

In our empirical study, we used a random sampling method to divide the corpus into three sets: a training set, a validation set, and a test set. The split ratio was 80% for training, 10% for validation, and 10% for testing. This ratio is consistent with the settings used in a previous study (Yu et al., 2022) to ensure a fair comparison.

4.3. Performance metrics

We employ four widely-used performance metrics to evaluate the performance of HGAN4VD and explain their relevance in vulnerability detection scenarios.

- **Precision**: $Precision = \frac{TP}{TP+FP}$. In vulnerability detection, Precision refers to the proportion of samples identified as vulnerabilities that are truly vulnerabilities. TP is the number of true positive samples, and FP is the number of false positive samples.
- **Recall**: $Recall = \frac{TP}{TP+FN}$. In the context of vulnerability detection, this metric represents the proportion of actual vulnerabilities correctly identified by the system. It measures the number of valid vulnerabilities detected relative to the total number of actual vulnerabilities. Here, FN refers to the number of false negative samples.
- **Accuracy**: $Accuracy = \frac{TP+TN}{TP+TN+FN+FP}$. In vulnerability detection, accuracy denotes the proportion of all samples that are correctly classified, regardless of whether they represent vulnerabilities or non-vulnerabilities. Here, FP refers to the number of false positive samples.
- **F1 score**: $F1score = 2 \times \frac{Precision \times Recall}{Precision+Recall}$. The F1 Score is the harmonic mean of Precision and Recall. It provides a balanced measure between Precision and Recall, especially crucial in vulnerability detection, that balances the aspects of detection accuracy and completeness.

4.4. Baselines

To evaluate the competitiveness of HGAN4VD, we conducted a comparative analysis against five state-of-the-art graph-based baseline methods. Specifically, we select SySeVR, Devign, Reveal, IVDetect, and AMPLE as baselines to ensure direct comparison with mainstream deep learning-based vulnerability detection methods. Among these, SySeVR represents an advanced token-based method, while the remaining four are advanced graph-based methods. Although these baselines primarily utilize deep learning techniques, they differ from our proposed heterogeneous graph neural network in structure and methodology. By benchmarking against these established methods, we aim to more accurately assess and highlight the advantages and effectiveness of our heterogeneous graph neural network in addressing vulnerability detection tasks.

- **SySeVR** (Li et al., 2021c) is a deep learning method that utilizes program slicing to generate four types of slices from suspicious points in the target program. It then learns vector features for each slice and performs classification and recognition based on these features.
- **Devign** (Zhou et al., 2019) builds a joint graph containing AST, CFG, DFG, and NCS, then uses GGNN for vulnerability detection.
- **Reveal** (Haiduc et al., 2010) divides code vulnerability detection into feature extraction and training phases. It leverages GGNN to extract features.
- **IVDetect** (Zhang et al., 2020) constructs PDG and utilizes GCN to learn the graph representation for vulnerability detection.
- **AMPLE** (Wen et al., 2023) uses the multi-head attention mechanism to calculate the weights of different edge types and then enhances the node representation by aggregating the attention scores of node edges.

For these baseline methods, we reproduced their models based on the descriptions provided in the original research and achieved performance results consistent with those reported. However, as the source code for the Devign model was not made available, we utilized the reproduced results reported by Wen et al. (2023).

4.5. Experimental settings

Our proposed method and baselines are implemented using the PyTorch framework. All experiments were implemented on a server composed of a multi-core CPU(Intel i7-13600K) and a GPU(NVIDIA GeForce RTX 4090). Following the work (Wen et al., 2023), we randomly divide the overall dataset into training, validation, and testing sets in a ratio of 8:1:1.

We utilized the open-source tool Joern (Yamaguchi et al., 2014) to build the original code structure diagram of the function, which is widely used in graph neural network-related works (Yamaguchi et al., 2014; Chakraborty et al., 2021; Wang et al., 2023a). The DGLv0.7.2 package is used to store and process graph-based data. Following the work (Wen et al., 2023), the dimension of the code text feature vector for each node in the vulnerability heterogeneity graph is 100. Although higher dimensions may lead to better results, we did not conduct experiments with higher dimensions due to hardware limitations. Our two-layer attention network consists of 8 attention heads and 12 meta-paths, with parameter complexity mainly determined by $O(K \times d^2)$ per meta-path for node-level attention and $O(R \times d)$ for semantic-level attention. To prevent overfitting, we apply parameter sharing across

Table 2
Hyper-parameters settings for experiments.

Category	Hyper-parameter	Value
Word2Vec	embedding_size	100
	epochs	50
HGAN4VD	num_metapath	12
	feature_size	100
	batch_size	64
	num_heads	8
	dropout	0.1
	epochs	100

meta-paths to reduce redundant weights, dropout (rate = 0.1) to prevent over-reliance on specific attention paths, and early stopping with a patience of 20 epochs. Additionally, L2 regularization is implicitly enforced through the RAdam (Liu et al., 2019) optimizer. The model is trained using the RAdam (Liu et al., 2019) optimizer for 100 epochs, with a learning rate of 1×10^{-3}. We did not use Adam (Kingma and Ba, 2014) because although Adam converges quickly, it is easy to converge to local solutions. At the same time, Adam can control the variance of the adaptive rate, resulting in better training performance (Liu et al., 2019). These measures ensure that model complexity grows linearly with the number of meta-paths rather than exponentially, maintaining efficiency even for smaller datasets.

Table 2 illustrates the specific hyperparameter setting of HGAN4VD.

5. Experimental results

5.1. RQ1: How efficient is our proposed method, HGAN4VD, in detecting vulnerabilities?

RQ1 aims to compare the HGAN4VD with five state-of-the-art baselines, which are illustrated in Section 4.4. Table 3 shows the overall results of the different methods concerning four evaluation measures as shown in Section 4.3, and we mark the best one of each metric in bold.

As presented in Table 3, our proposed method HGAN4VD outperforms all considered baselines. We observe that, across the three datasets, HGAN4VD exhibits superior accuracy and F1 scores compared to all baseline methods. Specifically, compared to AMPLE, HGAN4VD has improved accuracy by 7.7%, 1.5%, 2%, and F1 scores by 3.8%, 12.2%, 2.01% on the Devign, Reveal, and Fan et al. datasets, respectively. Moreover, the improvement in vulnerability detection performance of HGAN4VD compared to the Devign method is even more pronounced. Our method employs a heterogeneous graph to gather diverse semantic information and applies a two-layer attention mechanism to capture subtle code details within each semantic subgraph. In contrast, other GNN-based methods generate code representations directly from the original graph, lacking the capability to distinguish fine-grained heterogeneous features of the code effectively.

Our findings indicate that the vulnerability detection performance of the token-based method, SySeVR, is significantly inferior to that of the graph-based techniques. This performance gap can be attributed to SySeVR's loss of syntactic and semantic information during the source code transformation into token sequences. The experimental results further demonstrate that graph-based methods are highly effective in capturing both syntactic and semantic information from source code, leading to improved performance in vulnerability detection tasks. Therefore, the following observation can be made: HGAN4VD is more effective than state-of-the-art vulnerability detection methods.

> **Summary for RQ1:** HGAN4VD outperforms state-of-the-art vulnerability detection methods in terms of F1 score and accuracy, enabling more effective identification of vulnerabilities.

5.2. RQ2: How does each module contribute to the performance of HGAN4VD?

To answer this research question, we separately investigated the effects of graph simplification and heterogeneous graph attention networks on HGAN4VD.

(1) Graph simplification. Based on breadth-first traversal, we propose two heterogeneous graph simplification algorithms, HGS1 and HGS2, for dynamically removing redundant nodes from code structure graphs. Considering the impact of redundant control dependencies on vulnerability detection results, this section evaluates the performance of these two algorithms by comparing the code structure graphs processed by each algorithm with the original code structure graph without algorithmic processing and conducting ablation research on three datasets.

Table 4 presents the performance of the vulnerability detection models under different processing methods. From Table 4, it can be observed that the graph neural network trained on code structure graphs processed by Algorithm 1 and Algorithm 2 outperforms those trained on the initial code structure graphs without graph simplification algorithms. This indicates that both algorithms can remove redundant information from the code structure graphs without negatively impacting the model, as they ensure the removal of duplicate nodes without disrupting the original control dependencies. When both algorithms are used simultaneously, our approach achieves an average improvement of 7.2% in F1 score compared to not using any algorithms.

(2) HGAN Module. To evaluate the effectiveness of the two-layer attention network, we replaced it with several commonly used graph neural networks, including GCN, GGNN, and GAT, and conducted a comparative analysis with HGAN4VD. Additionally, we examined models utilizing only a single-layer attention network by removing the semantic-level attention layer and averaging the outputs obtained from the node-level attention layer.

As presented in Table 4, the two-layer attention network outperformed all other compared methods, demonstrating its superior effectiveness. This indicates that effectively utilizing the semantic and syntactic information in the heterogeneous graph can significantly improve the effectiveness of vulnerability detection. Furthermore, the two-layer attention network achieved better results than the single-layer attention network, underscoring the effectiveness of our subgraph partitioning approach. This method successfully distinguishes between different semantics by organizing them into separate meta-graphs and calculating their weights through the semantic-level attention layer.

> **Summary for RQ2:** The graph simplification algorithm and the two-layer attention network significantly contribute to the performance of HGAN4VD. Using the graph simplification algorithm resulted in an average improvement of 7.2% in F1 score across the three datasets, while employing the two-layer attention network led to an average increase of 9.39% in F1 score compared to the best baseline models across the three datasets.

5.3. RQ3: How does the selection of different meta-paths influence the performance of HGAN4VD?

We analyze the impact of meta-graphs derived from different meta-paths on the model's performance. In our approach, we classify 69 node types into four parent categories based on their functionality. With a meta-path length of 2, up to 12 unique meta-path combinations can be generated, resulting in 12 corresponding meta-graphs. While extending the meta-path length was considered, it was observed that increasing the length leads to a rapid and exponential growth in the number of meta-graphs, making it less practical. This will result in a small number of nodes in each meta-graph. Z. Wu et al. (2020) highlighted

Table 3
Comparison of HGAN4VD and baselines in terms of Accuracy, Precision, Recall, and F1 score Metrics.

Metrics \ Dataset	Devign				Reveal				Fan et al.			
Baseline	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
SySeVR	47.88	46.06	58.81	51.66	74.33	40.70	24.95	30.75	90.04	**30.91**	14.11	18.88
Devign	56.89	52.50	64.67	57.95	87.49	31.55	36.65	33.91	92.78	30.61	15.96	20.98
Reveal	61.08	55.60	69.70	62.20	81.77	31.55	61.21	41.24	87.10	17.22	33.04	22.87
IVDetect	57.26	52.33	57.30	54.84	–	–	–	-	–	–	–	–
AMPLE	62.16	55.64	**83.99**	66.94	92.71	51.06	46.15	48.48	93.14	29.92	34.58	32.11
HGAN4VD	**69.83**	**64.09**	78.79	**70.69**	**94.07**	**53.82**	**69.53**	**60.67**	**95.28**	22.88	**66.95**	**34.12**

Table 4
Ablation study results for various methods.

Metrics \ Dataset	Devign				Reveal				Fan et al.			
Setting	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
w/o HGS	64.53	59.68	71.41	65.02	92.52	45.10	63.23	52.65	87.39	16.76	59.91	26.20
w/o HGS1	67.94	62.26	77.60	69.09	92.58	45.55	65.09	53.59	87.61	18.30	66.89	28.74
w/o HGS2	65.39	59.73	76.81	67.20	92.67	46.06	66.95	54.57	94.34	18.96	64.22	29.27
w/o SAL	61.80	57.70	64.69	60.99	91.74	41.42	61.80	49.60	93.85	15.93	55.33	24.73
GCN	61.08	55.60	69.70	62.20	92.46	44.78	62.66	52.24	87.10	17.22	33.04	22.87
HGAN4VD	**69.83**	**64.09**	**78.79**	**70.69**	**94.07**	**53.82**	**69.53**	**60.67**	**95.28**	**22.88**	**66.95**	**34.12**

Table 5
Comparison of results across different meta-path selections.

Metrics \ Dataset	Devign				Reveal				Fan et al.			
nums	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score	Accuracy	Precision	Recall	F1 score
0	61.80	57.70	64.69	60.99	91.74	41.42	61.80	49.60	92.78	**30.61**	15.96	20.98
3	63.26	58.36	71.28	64.17	92.00	43.05	66.95	52.40	87.10	17.22	33.04	22.87
6	65.75	60.02	77.34	67.59	92.52	45.10	63.23	52.65	94.59	18.28	56.70	27.65
9	68.67	63.14	77.21	69.47	93.55	50.66	**71.39**	59.26	94.87	21.28	**67.15**	32.32
HGAN4VD	**69.83**	**64.09**	**78.79**	**70.69**	**94.07**	**53.82**	69.53	**60.67**	**95.28**	22.88	66.95	**34.12**

that insufficient nodes can result in limited information propagation and inadequate model training. Based on this observation, we opted not to consider more meta-paths. To further evaluate meta-graphs' impact, we trained the model using a two-layer attention network. We randomly removed three meta-graphs simultaneously, retraining the model to observe the resulting performance degradation. Given the randomness of meta-graph removal, we repeated the experiment 10 times and averaged the results to obtain the outcome.

Table 5 presents the relative decrease in four performance metrics corresponding to removing a given number of meta-graphs. The results demonstrate a positive correlation between the average attention weights assigned to the meta-graphs and the extent of performance degradation upon removal. This finding indicates that the model effectively assigns attention weights to meta-graphs, thereby enhancing the overall efficiency of vulnerability detection.

Summary for RQ3: Our model can assign higher attention weights to more important meta-graphs, thus better utilizing the complex information of the code structure graph.

5.4. RQ4: How robust is HGAN4VD when trained on small or imbalanced datasets?

To assess the model's robustness against overfitting, we evaluate HGAN4VD on different training sizes (100%, 50%, 25%, and 10%) while keeping the test set unchanged. As shown in Fig. 4, reducing the training size leads to a gradual decline in both F1-score and Recall, confirming the expected impact of reduced training data on model generalization.

(1)Ablation Study on Dataset Sizes. For the Devign dataset, F1-score and Recall decrease by 15.0% and 18.9%, respectively, when using only 10% of the training data. The Reveal dataset experiences a larger drop, with F1-score and Recall decreasing by 20.4% and 27.1%, respectively. The most significant decline is observed in the Fan et al. dataset, where F1-score and Recall drop by 30.6% and 37.4%, highlighting the challenges posed by data scarcity in highly imbalanced settings.

Despite these declines, the model does not exhibit extreme fluctuations or abrupt drops, suggesting that HGAN4VD is not overfitting to the full training set. The controlled degradation in performance validates the effectiveness of dropout regularization, parameter sharing, and early stopping, which prevent the model from over-relying on large training data and promote stable feature extraction across different dataset sizes.

(2)Class Imbalance Analysis. Given the inherent class imbalance in vulnerability detection tasks, we further analyzed the model's recall on the minority class (vulnerable samples). As training data decreases, Recall drops across all datasets, with Reveal and Fan et al. experiencing the most significant reductions due to their more skewed label distributions. In the Fan et al. dataset, Recall declines by 37.4%, emphasizing the difficulty in learning vulnerability patterns from limited positive samples. Devign, which has a more balanced sample distribution, shows a smaller Recall drop of 18.9%, indicating better resilience against class imbalance.

These results reaffirm that while HGAN4VD generalizes well across different dataset sizes, class imbalance remains a key challenge, particularly when training data is scarce. Future work could explore cost-sensitive learning, data augmentation, or adaptive loss functions tailored for highly imbalanced scenarios to mitigate overfitting risks and improve detection of rare vulnerabilities.

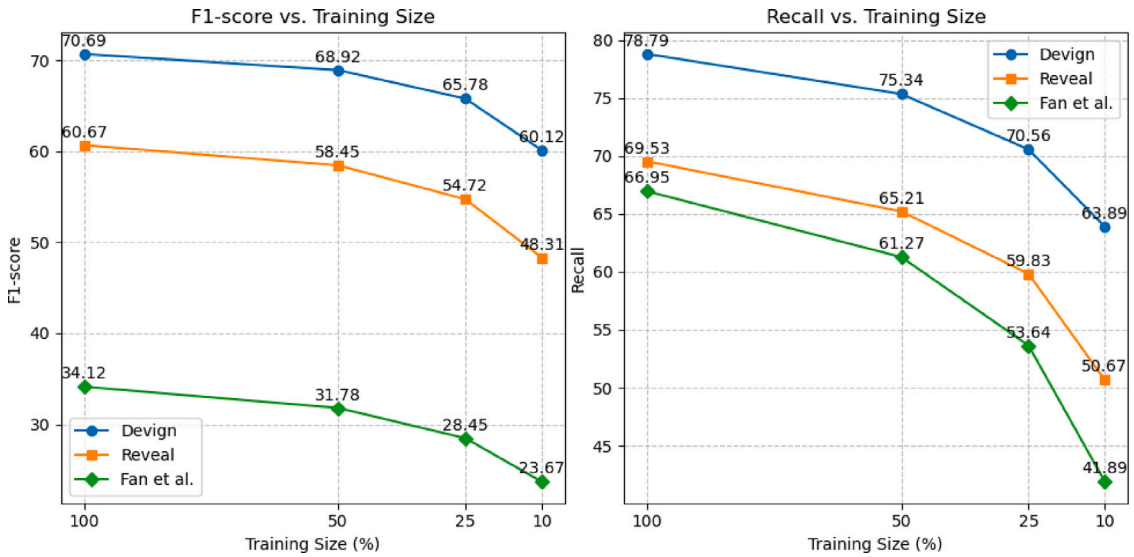

Fig. 4. F1 scores and Recall achieved under different Training Data Sizes.

Summary for RQ4: HGAN4VD maintains stable performance across varying training sizes and class distributions, showing resilience against overfitting and demonstrating good generalization even on limited or imbalanced datasets.

6. Discussion

6.1. Analysis of hyper-parameter settings

We perform a sensitivity analysis on the key hyperparameters of HGAN4VD, explicitly examining the learning rate and batch size. The F1 score was selected as the evaluation metric, as it offers a balanced measure of precision and recall, both critical for vulnerability detection tasks. The results, obtained across three datasets, are presented in Fig. 5, with all other hyperparameters fixed at their optimal values.

We can observe that for the Design and Reveal dataset, a learning rate of 1×10^{-4} and a batch size of 64 achieved the highest F1 scores. This suggests that these two datasets' code structure and semantic information have certain similarities, possibly due to their shared focus on widely used open-source C projects. These datasets may exhibit less code variation, making it easier for the model to converge to an optimal solution at a relatively higher learning rate and smaller batch size. In contrast, for the dataset of Fan et al. a learning rate of 1×10^{-5} and a batch size of 128 are the most effective. This dataset contains a more diverse range of code structures and vulnerabilities, which could lead to a more complex optimization process. Therefore, a lower learning rate and larger batch size are better suited to stabilizing the training process and avoiding potential overfitting. This highlights how the model's sensitivity to hyperparameters varies depending on the complexity and diversity of the data.

6.2. Cross-language generalization

To evaluate the generalization capability of HGAN4VD across programming languages, we conducted supplementary experiments on two widely used vulnerability detection datasets written in Java and Python. Specifically, we selected the Juliet Test Suite v1.3 (Java version) (National Institute of Standards and Technology, 2017) and the VUDENC dataset (Wartschinski et al., 2022). The Juliet Java dataset contains approximately 28,881 labeled code samples with diverse vulnerability types, synthesized under standardized CWE categories. The VUDENC dataset, on the other hand, consists of over 1,000 vulnerable functions labeled with different CWE categories.

Due to the lack of publicly reported baseline results from previous neural vulnerability detection methods on these datasets, we adopted the detection methods proposed in the original dataset papers as baselines. In Java, we refer to deep learning methods used in related research (Pang et al., 2015; Ma et al., 2017; Hovsepyan et al., 2012). In Python, we adopted the baseline results of the original VUDENC benchmark (Wartschinski et al., 2022), which reported performance for several static and neural vulnerability detectors. Our experimental setup, including model hyperparameters, training protocols, and evaluation metrics, remained consistent with the settings described in Section 4. This cross-language evaluation aims not to claim superiority over existing baselines but to demonstrate that our model architecture can be extended to other programming languages with reasonable performance, indicating its potential for broader applicability.

As presented in Table 6, HGAN4VD demonstrated competitive performance on both datasets, achieving better F1 scores than the respective baselines. These results provide preliminary evidence that the proposed heterogeneous graph representation and two-tier attention mechanism generalize well to other programming languages. However, we acknowledge the limited scale and coverage of these additional experiments. We plan to conduct more extensive cross-language and multi-language evaluations in future work.

Although HGAN4VD's metrics are slightly lower than the best results reported in each data set, they still achieve between 85% and 90% of the performance of the respective baselines on Java and Python. We further attribute the performance gap on the Python dataset to methodological differences. Specifically, the baseline method in VUDENC adopts a per-CWE training strategy, training a separate model for each vulnerability type. In contrast, HGAN4VD employs a unified model to detect all types of vulnerabilities, which, while more scalable, may perform poorly in highly imbalanced or specialized vulnerability settings. Similarly, differences in language semantics, such as dynamic typing and looser syntax in Python, may also pose challenges to generalization. However, these findings provide encouraging evidence that our graph-based representation and attention mechanism can be adapted to the Java and Python codebases, which shows promise for broader applicability across languages.

6.3. Threats and Limitations of HGAN4VD

Internal threats. The first internal threat to validity lies in the potential implementation errors of HGAN4VD. To address this, we

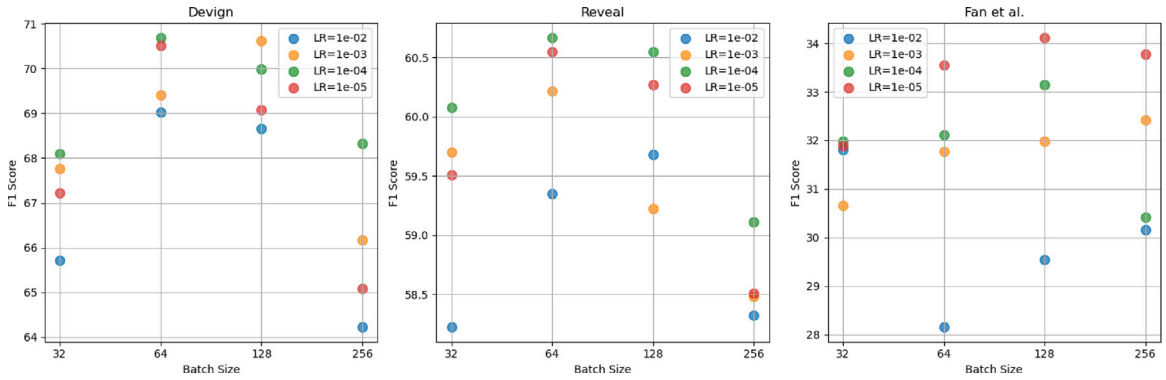

Fig. 5. F1 scores achieved under various Learning Rates and Batch Size configurations.

Table 6
Comparison of results across different programming languages.

Language	Baseline	Accuracy	Precision	Recall	F1 score
Java	Pang et al. (2015)	63	67	63	65
	VuRLE (Ma et al., 2017)	–	65	66	65
	Hovsepyan et al. (2012)	87	85	88	85
	HGAN4VD	74.5	73.1	73.8	76.9
Python	VUDENC (Wartschinski et al., 2022)	92.5	82.2	78.0	80.1
	HGAN4VD	68.9	67.5	68.2	70.8

utilized well-established libraries, such as PyTorch and DGL, to ensure robustness and minimize the likelihood of errors during implementation. The second internal threat arises from the use of Joern tool (Yamaguchi et al., 2014) to generate the Code Property Graph, which has known limitations. Specifically, Joern relies heavily on static analysis, which may fail to capture the code's dynamic behavior and complex logic. Additionally, due to the diversity of code structures, Joern might not cover all edge cases, potentially leading to incomplete or incorrect graph generation. Despite these limitations, Joern remains one of the most advanced tools for program dependency analysis and is widely used in the vulnerability detection community (Cao et al., 2022; Chakraborty et al., 2021; Zhou et al., 2019; Wen et al., 2023). Our evaluation shows that incorporating program dependencies significantly improves HGAN4VD's performance, making Joern's limitations not detrimental to the overall conclusions. The third internal threat is the baselines considered in RQ1. For these baselines, we replicated their baseline models based on the original research and achieved similar performance. However, since the Devign model did not provide source code, we relied on results reproduced by Wen et al. (2023), which might introduce slight variations.

External threats. The main external threat to this study is the corpus. We consider three datasets to evaluate HGAN4VD: FFMPeg+Qemu, Reveal, and Fan et al. These datasets, widely employed in prior vulnerability detection research (Wen et al., 2023; Zhou et al., 2019; Chakraborty et al., 2022), provide a comprehensive evaluation of our method. However, their focus on C-family languages may not fully reflect the diversity of code syntax and structure found in other programming paradigms. To mitigate this threat, we conducted preliminary cross-language experiments on the Juliet Test Suite v1.3 (Java) (National Institute of Standards and Technology, 2017) and VUDENC (Python) (Wartschinski et al., 2022), demonstrating the potential of our model to generalize to other languages. While these initial results are promising, we acknowledge their limited scale, and leave more extensive multi-language evaluation as future work.

Limitations. While the findings of this study highlight the effectiveness of HGAN4VD in vulnerability detection, several limitations remain. Firstly, the model generically treats vulnerabilities without accounting for the distinct characteristics of different vulnerability types. This generalized approach may result in suboptimal performance in cases where specific vulnerabilities exhibit unique patterns or behaviors not captured by the current feature representation. Future research will address this limitation by incorporating features specific to different types of vulnerabilities. Secondly, HGAN4VD relies heavily on static analysis and program dependency graphs. Although effective, this approach may not fully account for runtime vulnerabilities or dynamic execution patterns. To overcome this limitation, future work will explore integrating dynamic analysis techniques to complement the static analysis approach, enabling better detection of vulnerabilities that manifest only during program execution. Thirdly, the current evaluation has primarily focused on open-source C/C++ projects. To ensure the robustness and scalability of HGAN4VD across diverse environments, we plan to extend our evaluation to a broader range of software systems, including commercial projects developed in other programming languages. Moreover, HGAN4VD is designed to operate at the function level, and as such, it may not capture vulnerabilities that arise from interactions between multiple functions or modules. These include issues such as improper use of global variables, cross-function data flows, or module-level resource conflicts, which often require inter-procedural or inter-modular analysis to be detected. This limits the ability of HGAN4VD to handle system-wide or application-wide security assessments.

6.4. Complexity, Scalability and Applicability of HGAN4VD

HGAN4VD utilizes a heterogeneous graph attention network composed of two main modules: node-level attention and semantic-level attention layers. The overall time complexity of the node-level attention layer is $O(K|E|d)$, where K is the number of attention heads, $|E|$ is the number of edges in the meta-graph, and d is the feature dimension of each node. For semantic-level attention, the complexity is $O(R|V|d)$, where R is the number of meta-paths and $|V|$ is the number of nodes. Thanks to the graph simplification (HGS1, HGS2), the model reduces redundant nodes and edges, improving scalability on large datasets.

In practice, the preprocessing stage, which includes parsing code and constructing heterogeneous graphs, takes approximately 8 h and 50 min on the Devign dataset, 8 h on Reveal, and 82 h and 20 min on the Fan et al. dataset. The proposed framework benefits from stable convergence and performs well on large, imbalanced datasets thanks to regularization techniques such as dropout and early stopping.

HGAN4VD is suitable for large-scale C/C++ projects, including system software, embedded systems, industrial control, and security-critical open-source projects. Future work will extend the model to cross-module and multi-language scenarios.

7. Related work

Significant advancements have been made in integrating deep learning into various domains in recent years. This progress has led to the widespread adoption of deep learning techniques for feature extraction and the automated detection of code vulnerabilities and defects. Early research mainly involved advanced neural network architectures to process and understand code for defect prediction. Li et al. (2018) proposed VulDeePecker, a deep learning-based vulnerability detection framework for vulnerability detection. The framework processes C/C++ source code by slicing code segments involving function calls into "code gadgets". VulDeePecker employs a Recurrent Neural Network (RNN) for feature extraction and uses Bidirectional Long Short-Term Memory to address the issues of gradient vanishing and dependencies between preceding and subsequent directions. This approach reduces the false negative rate in vulnerability detection. However, the method treats code as natural language text, which results in the loss of rich semantic information, such as data flow and control flow in source code. Allamanis (Allamanis et al., 2017) pioneered graph neural networks to address the challenge of obtaining deep semantic features in code. Their method addresses two types of software problems: variable renaming and variable misuse. However, this approach does not attempt to solve the issue of vulnerability detection. Additionally, the model in the paper cannot handle inter-procedural code analysis. Given the complexity and abstraction of modern software, inter-procedural function calls are both common and critical, making this a notable limitation.

Recent studies (Steenhoek et al., 2023; Cao et al., 2022; Feng et al., 2020) focusing on code as text often treat source code as natural language, which results in the loss of unstructured semantic information inherent in code, such as control flow and data flow. Conversely, approaches employing graph neural networks (Wen et al., 2023; Wang et al., 2023a), extract unstructured features from code through compilation analysis, enabling the capture of potential semantic information. Compared to text-based methods, graph-based approaches are better equipped to represent the structural properties of code. However, existing graph-based methods predominantly utilize homogeneous graphs to represent code. These approaches fail to distinguish between edge types and do not support multiple edges. As a result, they treat data flow and control flow within code as the same type of edge, limiting the model's ability to deeply extract and represent the distinct features of these two types of semantic information. This limitation highlights the need for more sophisticated graph representations to capture the complex relationships within code better.

Recent research has investigated applying advanced machine learning techniques and attack graphs to mitigate the increasing complexity of cybersecurity threats. Liu et al. (2020) suggest a game theory-based method for defense decision-making in multistep attack scenarios. They employ game theory and attack graphs to model network vulnerabilities to optimize defense strategies. This method considers direct and indirect payoffs, including legal responsibility and counterattacks. This method is particularly effective in dynamic attack-defense environments, where attackers perpetually modify their strategies. To identify emerging threats, Nia et al. (2019) employ attribute-based attack graphs and self-avoiding random walks (SARW). By matching unknown network traffic to known threat patterns, their method obtains high sensitivity (up to 98%), rendering it suitable for real-time detection in intrusion detection systems (IDS). Liang et al. (2020) introduce FIT, a neural network-based tool for detecting vulnerabilities in firmware across various architectures in the context of IoT security. Ineffectiveness and efficiency, FIT surpasses state-of-the-art tools such as Gemini and Discover by employing a three-level ascribed control flow graph (3LACFG) and bipartite graph matching to compare binary functions. These studies highlight the effectiveness of integrating graph-based models with advanced learning methods to tackle system-level vulnerabilities and attack paths. While our work focuses on static vulnerability detection at the code level, future work could incorporate insights from attack graph modeling to capture cross-function or cross-module vulnerabilities better.

Our approach differs significantly from existing methods by addressing the limitations of text- and graph-based approaches. Unlike text-based methods (Steenhoek et al., 2023; Cao et al., 2022; Feng et al., 2020), which treat code as natural language and lose critical unstructured semantic information, our method leverages graph representations to preserve the structural and semantic relationships within code. Furthermore, unlike existing graph-based methods (Wen et al., 2023; Wang et al., 2023a), which predominantly use homogeneous graphs, our approach employs a heterogeneous graph representation to capture the complexity of code semantics better.

Specifically, our method distinguishes between edge types, such as data flow and control flow, and supports multiple edges between nodes. This allows the model to more accurately represent the relationships between code components, enabling deeper feature extraction and a more comprehensive understanding of the code's behavior. Additionally, our approach incorporates inter-procedural analysis, addressing the limitation of existing GNN-based methods that cannot handle function calls across procedures. By combining these innovations, our approach provides a more robust and effective framework for vulnerability detection, capable of capturing both the structural and semantic intricacies of modern software systems.

8. Conclusion

To address the issue of syntax and semantic information loss in existing deep learning-based software vulnerability detection, this paper proposes a new vulnerability detection method, HGAN4VD, based on heterogeneous intermediate representations of the source code. HGAN4VD utilizes Joern (Yamaguchi et al., 2014) to generate code structure graphs at the function level, constructs heterogeneous graphs for the code, proposes two simplification algorithms for heterogeneous graphs to remove redundant information, and then utilizes Word2Vec (Mikolov et al., 2013) to generate vectorized representations of heterogeneous graphs. Finally, a two-layer attention network is used to implement software vulnerability detection. Experiments on three benchmark datasets show HGAN4VD outperforms state-of-the-art baselines. These results highlight the effectiveness of HGAN4VD in leveraging global information from code graphs for vulnerability detection.

In the future, we aim to extend this model in several directions. First, we plan to extend our evaluation to more programming languages further and explore transfer learning techniques to enhance language adaptability. We also plan to extend the framework to support cross-function and cross-module analysis by incorporating inter-procedural dependency graphs or global resource graphs, which could further enhance the model's capability to detect system-level vulnerabilities. Furthermore, exploring more effective methods for classifying nodes in code graphs based on their functional roles will be another key direction of our research.

CRediT authorship contribution statement

Yucheng Zhang: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Data curation, Conceptualization. **Xiaolin Ju:** Writing – review & editing, Validation, Supervision, Methodology, Formal analysis, Data curation, Conceptualization. **Xiang Chen:** Writing – review & editing, Supervision, Methodology, Investigation. **Misbahul Amin:** Writing – review & editing, Visualization. **Zilong Ren:** Writing – review & editing, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

References

Aggarwal, A., Jalote, P., 2006. Integrating static and dynamic analysis for detecting vulnerabilities. COMPSAC'06, In: 30th Annual International Computer Software and Applications Conference, vol. 1, IEEE, pp. 343–350.

Allamanis, M., Brockschmidt, M., Khademi, M., 2017. Learning to represent programs with graphs. CoRR, abs/1711.00740 arXiv:1711.00740. URL http://arxiv.org/abs/1711.00740.

Ba, C.T., Interdonato, R., Ienco, D., Gaito, S., 2025. MARA: A deep learning based framework for multilayer graph simplification. Neurocomputing 612, 128712.

Backes, M., Köpf, B., Rybalchenko, A., 2009. Automatic discovery and quantification of information leaks. In: 2009 30th IEEE Symposium on Security and Privacy. IEEE, pp. 141–153.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. ArXiv preprint arXiv:1409.0473.

Black, P.E., Black, P.E., 2018. Juliet 1.3 Test Suite: Changes from 1.2. US Department of Commerce, National Institute of Standards and Technology ....

Cai, Z., Lu, L., Qiu, S., 2019. An abstract syntax tree encoding method for cross-project defect prediction. IEEE Access 7, 170844–170853.

Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136, 106576.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1456–1468.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng..

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2022. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9), 3280–3296. http://dx.doi.org/10.1109/TSE.2021.3087402.

Dam, H.K., Tran, T., Pham, T., Ng, S.W., Grundy, J., Ghose, A., 2017. Automatic feature learning for vulnerability prediction. ArXiv preprint arXiv:1708.02368.

Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y., 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 60–71.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. ArXiv preprint arXiv:2002.08155.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.

Guo, N., Li, X., Yin, H., Gao, Y., 2020a. Vulhunter: An automated vulnerability detection system based on deep learning and bytecode. In: Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21. Springer, pp. 199–218.

Guo, Z., Shen, Y., Bashir, A.K., Imran, M., Kumar, N., Zhang, D., Yu, K., 2020b. Robust spammer detection using collaborative neural network in internet-of-things applications. IEEE Internet Things J. 8 (12), 9549–9558.

Haiduc, S., Aponte, J., Moreno, L., Marcus, A., 2010. On the use of automated text summarization techniques for summarizing source code. In: 2010 17th Working Conference on Reverse Engineering. IEEE, pp. 35–44.

Hovsepyan, A., Scandariato, R., Joosen, W., Walden, J., 2012. Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th International Workshop on Security Measurements and Metrics. pp. 7–10.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. ArXiv preprint arXiv:1412.6980.

Kipf, T.N., Welling, M., 2016. Semi-supervised classification with graph convolutional networks. ArXiv preprint arXiv:1609.02907.

Landman, D., Serebrenik, A., Vinju, J.J., 2017. Challenges for static analysis of java reflection-literature review and empirical study. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 507–518.

Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 627–637.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. ArXiv preprint arXiv:1511.05493.

Li, Y., Wang, S., Nguyen, T.N., 2021a. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303.

Li, Z., Zou, D., Wang, Z., Jin, H., 2019. Survey on static software vulnerability detection for source code. Chin. J. Netw. Inf. Secur. 5 (1), 1–14.

Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2021b. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans. Dependable Secur. Comput. 19 (4), 2821–2837.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021c. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secur. Comput. 19 (4), 2244–2258.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. ArXiv preprint arXiv:1801.01681.

Liang, F., Qian, C., Yu, W., Griffith, D., Golmie, N., 2022. Survey of graph neural networks and applications. Wirel. Commun. Mob. Comput. 2022 (1), 9261537.

Liang, H., Xie, Z., Chen, Y., Ning, H., Wang, J., 2020. FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. Comput. Secur. 95, 101823. http://dx.doi.org/10.1016/j.cose.2020.101823.

Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans. Dependable Secur. Comput. 18 (5), 2469–2485.

Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2539–2541.

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., Han, J., 2019. On the variance of the adaptive learning rate and beyond. ArXiv preprint arXiv:1908.03265.

Liu, Z., Qian, P., Wang, X., Zhu, L., He, Q., Ji, S., 2021. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. ArXiv preprint arXiv:2106.09282.

Liu, X., Yan, M., Deng, L., Li, G., Ye, X., Fan, D., Pan, S., Xie, Y., 2022. Survey on graph neural network acceleration: An algorithmic perspective. ArXiv preprint arXiv:2202.04822.

Liu, J., Zhang, Y., Hu, H., Tan, J., Leng, Q., Chang, C., 2020. Efficient defense decision-making approach for multistep attacks based on the attack graph and game theory. Math. Probl. Eng. 2020, 1–12. http://dx.doi.org/10.1155/2020/9302619.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. ArXiv preprint arXiv:2102.04664.

Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H., 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22. Springer, pp. 229–246.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. ArXiv preprint arXiv:1301.3781.

National Institute of Standards and Technology, 2017. Juliet java 1.3 test suite. (Accessed 01 April 2025). https://samate.nist.gov/SARD/test-suites/111.

Nia, M.A., Bahrak, B., Kargahi, M., Fabian, B., 2019. Detecting new generations of threats using attribute-based attack graphs. IET Inf. Secur. 13 (4), 293–303. http://dx.doi.org/10.1049/iet-ifs.2018.5409.

Pang, Y., Xue, X., Namin, A.S., 2015. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In: 2015 IEEE 14th International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 543–548.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 757–762.

Schmidt, M., Fung, G., Rosales, R., 2007. Fast optimization methods for l1 regularization: A comparative study and two new approaches. In: Machine Learning: ECML 2007: 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007. Proceedings 18. Springer, pp. 286–297.

Shankar, U., Talwar, K., Foster, J.S., Wagner, D., 2001. Detecting format string vulnerabilities with type qualifiers. In: 10th USENIX Security Symposium. USENIX Security 01.

Shar, L.K., Briand, L.C., Tan, H.B.K., 2014. Web application vulnerability prediction using hybrid program analysis and machine learning. IEEE Trans. Dependable Secur. Comput. 12 (6), 688–707.

Shin, Y., Williams, L., 2013. Can traditional fault prediction models be used for vulnerability prediction? Empir. Softw. Eng. 18, 25–59.

Siow, J.K., Liu, S., Xie, X., Meng, G., Liu, Y., 2022. Learning program semantics with code representations: An empirical study. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 554–565.

Steenhoek, B., Rahman, M.M., Jiles, R., Le, W., 2023. An empirical study of deep learning models for vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, pp. 2237–2248. http://dx.doi.org/10.1109/ICSE48619.2023.00188.

Tang, W., Tang, M., Ban, M., Zhao, Z., Feng, M., 2023a. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. J. Syst. Softw. 199, 111623.

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 Syst. Appl. 238, 122216.

Tang, G., Yang, L., Zhang, L., Cao, W., Meng, L., He, H., Kuang, H., Yang, F., Wang, H., 2023b. An attention-based automatic vulnerability detection approach with GGNN. Int. J. Mach. Learn. Cybern. 14 (9), 3113–3127.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks. ArXiv preprint arXiv:1710.10903.

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., Yu, P.S., 2019. Heterogeneous graph attention network. In: The World Wide Web Conference. pp. 2022–2032.

Wang, W., Nguyen, T.N., Wang, S., Li, Y., Zhang, J., Yadavally, A., 2023a. DeepVD: Toward class-separation features for neural network vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2249–2261.

Wang, J., Xiao, H., Zhong, S., Xiao, Y., 2023b. DeepVulSeeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism. Future Gener. Comput. Syst. 148, 15–26.

Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L., 2022. VUDENC: vulnerability detection with deep learning on a natural codebase for python. Inf. Softw. Technol. 144, 106809.

Wei, B., Li, Y., Li, G., Xia, X., Jin, Z., 2020. Retrieve and refine: exemplar-based neural comment generation. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 349–360.

Wen, X.-C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. ArXiv preprint arXiv:2302.04675.

Wen, S., Haghighi, M.S., Chen, C., Xiang, Y., Zhou, W., Jia, W., 2014. A sword with two edges: Propagation studies on both positive and negative information in online social networks. IEEE Trans. Comput. 64 (3), 640–653.

Wu, Y., Lu, J., Zhang, Y., Jin, S., 2021. Vulnerability detection in c/c++ source code with graph representation learning. In: 2021 IEEE 11th Annual Computing and Communication Workshop and Conference. CCWC, IEEE, pp. 1519–1524.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y., 2020. A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. 32 (1), 4–24.

Wu, F., Wang, J., Liu, J., Wang, W., 2017. Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications. ICCC, IEEE, pp. 1298–1302.

Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H., 2022. VulCNN: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2365–2376.

Xu, Z., Chen, B., Chandramohan, M., Liu, Y., Song, F., 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 462–472.

Xu, J., Wang, F., Ai, J., 2020. Defect prediction with semantics and context features of codes based on graph representation learning. IEEE Trans. Reliab. 70 (2), 613–625.

Yamaguchi, F., 2017. Pattern-based methods for vulnerability discovery. It- Inf. Technol. 59 (2), 101–106.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K., 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 499–510.

Yang, X., 2020. An overview of the attention mechanisms in computer vision. J. Phys.: Conf. Ser. 1693 (1), 012173.

Yu, K., Lin, L., Alazab, M., Tan, L., Gu, B., 2020. Deep learning-based traffic safety solution for a mixture of autonomous and manual vehicles in a 5G-enabled intelligent transportation system. IEEE Trans. Intell. Transp. Syst. 22 (7), 4337–4347.

Yu, C., Yang, G., Chen, X., Liu, K., Zhou, Y., 2022. BashExplainer: Retrieval-augmented bash code comment generation based on fine-tuned CodeBERT. In: 2022 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 82–93.

Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X., 2020. Retrieval-based neural source code summarization. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. ICSE, IEEE, pp. 1385–1397.

Zhao, J., Guo, S., Mu, D., 2021. DouBiGRU-A: software defect detection algorithm based on attention mechanism and double BiGRU. Comput. Secur. 111, 102459.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M., 2020. Graph neural networks: A review of methods and applications. AI Open 1, 57–81.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.