VDPG: Binary Vulnerability Detection via Decompiled Pseudo Code and Double-branch GCN

Wenbo Zhang
1024041136
Nanjing, China
1024041136@njupt.edu.cn

***Abstract*—This invention introduces an automated vulnerability detection method using intermediate pseudo-code and graph neural networks. It involves converting the assembly code of a binary program into pseudo-code, identifying sensitive functions, and building a Program Dependence Graph (PDG) with these functions as central nodes to extract data dependence subgraphs. Then, it combines the pre-trained CodeT5-small model and Graph Attention Networks (GAT) to generate graph semantic embedding vectors, and uses the node2vec algorithm for graph structural embedding vectors. Finally, it trains a dual-branch feature analysis model based on Graph Convolutional Networks (GCN) and a classification model using Multi-Layer Perceptrons (MLP). This invention enables the identification of sensitive function vulnerabilities in binary programs without source code, greatly improving the accuracy and efficiency of automated vulnerability mining. It effectively overcomes the limitations of conventional methods and has significant practical value.**

***Index Terms*—binary program; decompile; vulnerability detection; program slicing; GAT; GCN**

I. Introduction

In the process of social development, computer software plays a crucial role. However, since the birth of software systems, the problem of software vulnerabilities has emerged simultaneously. Once hackers exploit these vulnerabilities, it may lead to serious consequences such as data leakage, service interruption, and information tampering. In addition, there are a large number of legacy software in real life. For this type of software, the source code is lost, and it is no longer maintained, making it impossible to directly analyze potential vulnerabilities from the source code [1]. At the same time, most commercial software is not open - sourced and only provides users with binary files. Users cannot easily determine whether these binary programs contain potential vulnerabilities [2]. Therefore, in practical production and life, vulnerability detection of binary programs is necessary.

Although traditional vulnerability discovery methods can uncover some vulnerabilities, they highly rely on manually defined vulnerability patterns, resulting in time-consuming and labor-intensive processes and poor detection performance. Current research shows that deep learning technology can break through the bottlenecks encountered by traditional vulnerability detection technologies in terms of automation and performance. As a result, deep learning-based vulnerability detection technology has become an important and highly regarded research direction in the field of software security.

Currently, deep learning has been widely applied in the field of binary vulnerability detection. However, existing analysis methods, such as BVDetector [3] proposed by Tian et al. and VulDeePecker [4] proposed by Li et al., mostly focus on the assembly representation of decompiled binary programs. This analysis method based on assembly code has obvious drawbacks. The assembly representation is a low - level code representation, which means that the semantic information of the code itself is ignored. At the same time, assembly code contains a large amount of information irrelevant to vulnerabilities, with excessive redundant information, which is not conducive to the identification and detection of vulnerabilities.

BinVulDet [5] proposed by Wang et al. has noticed the impact of coarse-grained analysis of assembly code on vulnerability detection and instead uses pseudo-code to restore and simplify the original binary execution flow. However, this method still treats the extracted code fragments as sequential streams and combines deep learning methods such as convolutional neural networks(CNN) and recurrent neural networks(RNN) to achieve automatic vulnerability discovery. The drawback of sequential stream analysis is that it ignores the structural information of the code and does not make full use of the topological information of the graph, resulting in a relatively high false positive rate of the model.

Contributions. Our contributions are as follows:

(1) To the best of our knowledge, we are the first to propose extracting vulnerability features based on graph-based high-level pseudo-code obtained by decompiled binary programs.
(2) We develop a novel graph data structure by leveraging the rich semantic and structural information of the code.
(3) We propose a graph learning model VDPG to detect binary vulnerability. VDPG utilizes double-branch convolution to adapt the diverse relationships in PDG and adopts features to reduce false alarms.

II. Related work

Existing static binary code vulnerability detection techniques can be roughly divided into code similarity-based methods and vulnerability pattern-based methods [5]–[7]. Code similarity-based methods construct a database of known vulnerability codes in advance and then calculate the similarity between the target code and the known vulnerability code. The

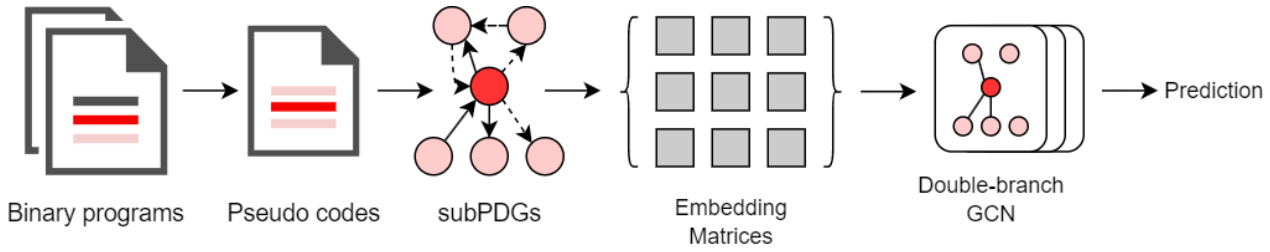

Fig. 1. Overview of VDPG.

target code has the same vulnerability as the known vulnerability code when the similarity reaches a certain threshold. Vulnerability pattern-based methods extract the vulnerability pattern or signature to detect whether the target code contains these features.

A. Code similarity-based methods

Code similarity-based methods can only detect the cloning vulnerability, not the recurring vulnerability. Moreover, this method must build many known vulnerability databases in advance for comparison, and the completeness of the database will also affect the detection results. VDPG can detect both clone and recursion vulnerabilities simultaneously and does not need to build a database of known vulnerabilities [8], [9].

B. Vulnerability pattern-based methods

The methods based on vulnerability patterns extract corresponding vulnerability patterns from existing vulnerability code and then detect target code based on these patterns. However, the patterns and rules are too general, and some features are manually extracted by experts, so these works suffer from high false-positive rates [10], [11]. Also, it takes expensive time to verify the results manually. VDPG automatically extracts vulnerability models by using deep learning methods [8], [12].

III. Design of VDPG

A. System overview

The overall workflow of VDPG is illustrated in Fig. 1. The inputs of VDPG are binary programs, which will be decompiled to obtain pseudo codes with a higher level of semantic information than assembly code. Next, we construct the Program Dependency Graphs (PDGs) of the pseudo code, and take the library/API function call as the point of interest and the library/API function call parameters as variables to construct the slicing criteria. Then, the nodes and edges of subgraph are converted into embedding matrices, which are fed into a graph-based deep neural network double-branch GCN that predicts if the input patches are either security or non-security patches.

B. Decompiling binary codes to pseudo codes

The C-like pseudo-code generated by the decompilation tool IDA Pro through assembly execution flow simulation offers significant advantages over raw assembly code. These advantages are primarily manifested in two aspects:

- **Enhanced Readability.** It enables researchers to comprehend the code logic more readily.
- **Richer Semantic Information.** It facilitates in-depth analysis of code behavior and potential vulnerabilities.

Subsequently, the graph analysis tool Joern extracts a data dependency graph representation from this pseudo-code by parsing its C language syntax.

C. Extracting code gadgets

Collect all classes or functions annotated as vulnerabilities in existing vulnerability reports, along with their usage specifications, to establish function call matching rules. Using vulnerable class invocations or function API calls documented in these reports as the judgment basis, match specific sensitive function calls within the pseudo-code.

Algorithm 1 Extracting subPDGs.

Input: A program $P = f_1, f_2, ..., f_n$; a library/API function set $L = (l_1, l_2, l_3, ..., l_n)$
Output: The set of subPDGs C

```
if some condition is true then
    do some processing
else if some other condition is true then
    do some different processing
else if some even more bizarre condition is met then
    do something else
else
    do the default actions
end if
```

Furthermore, the method for constructing the function call path subgraph comprises the following steps:

- **Precise Node Localization.** Based on the matching rules, accurately locate the sensitive function node.
- **Backward Slice Subgraph Extraction.** Employ a depth-first traversal strategy to extract a subgraph by traversing backward from the identified node position to a specified depth. This subgraph is termed the backward slice subgraph [13].
- **Forward Slice Subgraph Extraction.** Reverse the original graph. Using the same depth-first traversal strategy,

TABLE I
DATASETS FOR EXPERIMENTS

Datasets	#subPDGs	#subPDGs with vulnerability	#subPDGs without vulnerability	#Train	#Test
IDA_1	5014	2103	2911	4011	1003
IDA_55	48310	19172	29138	38648	9662
IDA_CVE	27	27	0	0	0

extract a subgraph by traversing forward from the identified node position to a specified depth. This subgraph is termed the forward slice subgraph [13].

- **Sensitive Function Subgraph Formation.** Concatenate the backward slice subgraph and the forward slice subgraph to form the final sensitive function subgraph.

D. Embedding

The embedding method integrates two key dimensions of graph information to generate comprehensive node embeddings for subgraphs extracted from sensitive functions, as depicted in the original description. This dual-component approach first processes graph semantic information embedding, which focuses on capturing the intrinsic attributes and contextual semantics of nodes within the subgraph. For instance, in the context of program analysis or security-sensitive graphs, semantic information may include node types (e.g., function calls, control flow statements), textual labels, or API names that reflect the functional or operational semantics of nodes. Techniques such as node feature aggregation (e.g., averaging or attention-based pooling of node attributes) or natural language processing-inspired methods (e.g., embedding APIs as text tokens) might be employed to encode this semantic layer.

Simultaneously, the graph structure information embedding component emphasizes the topological relationships and connectivity patterns among nodes. This involves quantifying the graph's architecture, such as node degrees, edge density, path-based features (e.g., shortest paths, meta-paths as in PathSim [1]), or graph-based features like centrality metrics. Structural embeddings may be derived through graph neural networks (GNNs) that propagate information across edges (e.g., Graph Convolutional Networks [7] or Graph Attention Networks), or through unsupervised methods like Node2Vec or GraphSAGE that capture neighborhood context.

By synergizing semantic and structural cues, the embedding method enables downstream tasks like anomaly detection, vulnerability prediction, or inter-subgraph similarity analysis to leverage both explicit node attributes and implicit relational contexts, thereby enhancing the discriminative power of the embeddings for security-critical applications.

Graph Semantic Embedding. In terms of generating the graph semantic information embedding vector, a pre-trained model is utilized to vectorize the node information in the subgraph of sensitive functions. The pre-trained model CodeT5-small converts each element in the node information sequence into its corresponding vector representation, thereby obtaining the vector representation of each node's information in the subgraph. Subsequently, with the aid of the Graph Attention Network (GAT), the attention mechanism with residual connections is introduced to perform information transmission and aggregation operations on the nodes [2]. This updates the vector representation of each node. The formula for calculating the updated vector representation is as follows:

$$v_i{}' = \sigma(\sum_{j \in N_i} a_{ij} \mathbf{W} v_j) + \mathbf{W_{res}} v_i, \tag{1}$$

where a_{ij} is the attention coefficient, which is calculated as:

$$a_{ij} = \frac{LeakyReLU(a^\top [\mathbf{W}h_i || \mathbf{W}h_j])}{\sum_{k \in N_i} LeakyReLU(a^\top [\mathbf{W}h_i || \mathbf{W}h_k])}, \tag{2}$$

where h_i represents the vector representation of the central node, $h_j$represents the vector representation of the target node, h_k represents the vector representation of all neighbor nodes of the node, N_i is the set of neighbors of node i, $\mathbf{W}$ is a shared learnable matrix.

After updating the vectors of each node in the graph, Attention-based Pooling is employed to dynamically capture key nodes. All node vectors are integrated to form the graph semantic embedding vector. The specific calculation formula is as follows:

$$\mathbf{X_s} = \sum_{i=1}^{N} \alpha_i v_i{}'. \tag{3}$$

The attention coefficient α_i is calculated as:

$$\alpha_i = softmax(q^\top tanh(\mathbf{W_attn} v_i{}' + b)), \tag{4}$$

where $q \in \mathbb{R}^d$ is a learnable parameter vector, which is used to measure the importance of each node; $\mathbf{W_attn} \in \mathbb{R}^{\times}$ is a learnable parameter matrix; and $tanh(\cdot)$ is a non-linear activation function, with the specific calculation formula: $tanh(x) = \frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$.

Algorithm 2 Transforming subgraphs to vectors.

Input: A program $P = f_1, f_2, ..., f_n$; a library/API function set $L = (l_1, l_2, l_3, ..., l_n)$
Output: The set of subPDGs C

```
if some condition is true then
    do some processing
else if some other condition is true then
    do some different processing
else if some even more bizarre condition is met then
    do something else
else
    do the default actions
end if
```

Graph structure Embedding. The extraction of graph structure information is achieved through the node2vec embedding method. This method first captures the link information of nodes in the graph via a biased random walk mechanism. The node sequence generated by the random walk is used as the input of the model. After training, the graph structure information embedding vector is obtained. Finally, the graph embedding vector is obtained by concatenating the graph semantic information embedding vector and the graph structure information embedding vector, expressed as: $\mathbf{g} = \mathbf{g_{semantic}}||\mathbf{g_{structure}}$.

Furthermore, for the graph structure information embedding implemented by node2vec, its characteristic lies in using the node sequence generated by random walk as the input of the model, and obtaining the graph structure information embedding vector after training. The objective of this model is to optimize the following function:

$$max_f \sum_{u \in V} logPr(N_S(U)|f(u)). \tag{5}$$

E. Double-branch GCN

The dual-branch GCN was proposed in this paper to address the limitations of traditional GCN models when dealing with the distinct types of edge attributes in program dependence graphs, namely data dependence and control dependence. This novel approach improves the feature extraction capability of the model and enhances its ability to learn vulnerability features. The dual-branch GCN first divides the input graph data into two separate subgraphs based on edge attributes, and constructs their adjacency matrices. Then, a mathematical formula is used to optimize the traditional adjacency matrix to balance the computational bias caused by significant differences in node degrees. Subsequently, the dual branches extract and aggregate node features using graph convolutional formulas. Finally, the features obtained from different branches are combined to form a comprehensive feature representation. The dual-branch GCN demonstrates better performance in handling different types of edge attributes compared to traditional single-branch GCN models.

REFERENCES

[1] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "PathSim: Meta path-based top-K similarity search in heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 992–1003, 2011.

[2] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," 2021.

[3] J. Tian, W. Xing, and Z. Li, "BVDetector: A program slice-based binary code vulnerability intelligent detection system," *Information and Software Technology*, vol. 123, p. 106289, 2020.

[4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, (San Diego, CA), Internet Society, 2018.

[5] Y. Wang, P. Jia, X. Peng, C. Huang, and J. Liu, "BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention," *Computers & Security*, vol. 125, p. 103023, 2023.

[6] C. Batur Şahin and L. Abualigah, "A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection," *Neural Computing and Applications*, vol. 33, no. 20, pp. 14049–14067, 2021.

[7] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2409–2426, 2023.

[8] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 224–236, 2021.

[9] G. Xu, *Frontiers in Cyber Security: Third International Conference, FCS 2020, Tianjin, China, November 15-17, 2020, Proceedings*. No. v.1286 in Communications in Computer and Information Science Ser, Singapore: Springer Singapore Pte. Limited, 2020.

[10] B. Wu, S. Liu, Y. Xiao, Z. Li, J. Sun, and S.-W. Lin, "Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (San Francisco CA USA), pp. 1371–1383, ACM, 2023.

[11] X.-C. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, and Z. Gu, "When Less is Enough: Positive and Unlabeled Learning Model for Vulnerability Detection," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 345–357, 2023.

[12] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, (Virtual Event), pp. 378–389, ACM, 2022.

[13] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.