Improving Vulnerability Detection with Hybrid Code Graph Representation

Xiangxin Meng
SKLSDE Lab, Beihang University
Beijing, China
mengxx@buaa.edu.cn

Shaoxiao Lu
SKLSDE Lab, Beihang University
Beijing, China
lushaoxiao@buaa.edu.cn

Xu Wang*[†]
SKLSDE Lab, Beihang University
Beijing, China
xuwang@buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University
Beijing, China
liuxd@act.buaa.edu.cn

Chunming Hu*
SKLSDE Lab, Beihang University
Beijing, China
hucm@buaa.edu.cn

***Abstract*—The increasing richness of software applications contributes to the enhanced productivity and convenience in daily life. However, the growing software complexity simultaneously poses significant challenges to software security. As one of the most important solutions, vulnerability detection technology attracts increasing attention. This paper proposes a novel vulnerability detection method HybridNN based on graph neural networks (GNNs). To begin, we simplify the code property graph (CPG) to design a hybrid code graph (HCG) which is better suitable for the deep semantic extraction via GNN models. Subsequently, the datasets consisting of considerable amount of samples including both artificially synthesized and real-world vulnerabilities are constructed. Next, we leverage a GNN model with a hierarchical attention mechanism which is proficient in extracting deep semantics in heterogeneous graphs, and apply it to the newly designed HCG representation. Moreover, we propose UD-Sampling method, which combines up-sampling and down-sampling methods, to balance the distribution of the training samples. Finally, extensive experiments are conducted, showing that HybridNN outperforms all baseline methods.**

***Index Terms*—vulnerability detection, software security, heterogeneous graph representation, graph neural network**

I. Introduction

Since the scale and complexity of software systems continue to grow, more and more software vulnerabilities are introduced, which increases the potential risks in many critical applications [1], such as financial transactions [2], autonomous driving [3], [4], and mission critical systems [5], [6]. The software vulnerabilities can be exploited by attackers to gain unauthorized access, compromise sensitive information and disrupted services [6], leading to severe consequences. In 2014, the vulnerability Heartbleed emerged in the OpenSSL encryption library, enabling attackers to read sensitive information from affected servers [7]. In 2021, more than 35,000 Java packages were impacted by the log4j vulnerability, which allows attackers to perform remote code execution by exploiting the insecure JNDI lookups feature [8].

* Zhongguancun Laboratory, Beijing, P.R.China
[†] Corresponding author: Xu Wang, xuwang@buaa.edu.cn.

In response to the challenges mentioned above, the studies on software vulnerability detection attract more attentions, which are essential for guaranteeing the security of software systems [9]–[24]. Early vulnerability detection methods mainly include static analysis and dynamic analysis techniques. Static analysis such as rule-based analysis [25], [26] and symbolic execution [27] analyzes the source/binary code without execution to extract specific patterns of vulnerable code. It operates at a high speed, but it is prone to generate false positives and struggles to detect complex vulnerabilities [15]. Dynamic analysis is based on program execution, where fuzz testing is most frequently used [28], [29]. It can help detect vulnerabilities such as memory leaks, null pointer issues, and access control problems. However, it requires a large number of test cases and spends considerable execution time, which may also not cover enough execution paths.

Early methods heavily rely on the hand-crafted patterns designed by experts, which are difficult to be applied to all vulnerabilities. The introduction of deep learning technology has brought new possibilities to vulnerability detection tasks. For instance, VulDeePecker [16] incorporates source code and data/control dependencies into the program slices, and utilizes a BiLSTM (Bidirectional Long Short-Term Memory) [30], [31] based model to perform binary classifications. SySeVR [17] enriches the semantic representations by leveraging PDG (Program Dependence Graph) [32] and employs a BiGRU (Bidirectional Gated Recurrent Unit) [33] model to mine deep semantics. Russell et al. [18] labels the source code and adopts a CNN (Convolutional Neural Network) model, which also leverages ensemble learning [34] and random forest [35] techniques to further improve the performance. Recent studies employ GNN (Graph Neural network) model for this task and have achieved better performance. Devign [19] utilizes CPG (Code Property Graph) [36] as the code representation and leverages an GGNN (Graph-based Global Neural Network) [37] model to extract deep semantics, Compared with Devign, ReVeal [20] utilizes an up-sampling approach to sample the pooling vectors generated by the GGNN model to balance the distributions of the training samples. Though achieving some improvements, existing GNN-based methods still face two main constraints: (1) They are ineffective in handling information propagation for heterogeneous graphs, that is, the weights of different types of edges should not be the same. (2) The number of edge types in the recently used graph representations increases, making more information available for model training. However, it makes the data sparsity problem more severe, where the edge types that contain few samples may hinder the overall effective learning.

In this paper, we propose a novel vulnerability detection method HybridNN, which is based on GNN architectures. Specifically, we first design a heterogeneous graph representation called Hybrid Code Graph (HCG in the following part), which is simplified from CPG and consists of three types of edges. It captures code information from syntax tree structures, control flows, and data/control dependencies. This graph is designed to mitigate the impact of data sparsity on model training. Afterwards, a GNN model with a hierarchical attention mechanism [38] is employed to extract the deep semantic features from HCG, which has the ability to capture the different impacts from different edge types on the target node. Furthermore, to balance the distributions of training samples, we propose a method called UD-Sampling to sample the pooling vectors generated by the already trained GNN model, which combines the advantages of up-sampling and down-sampling methods. Finally, the processed balanced samples are leveraged to retrain the model for further enhancement.

We conduct extensive experiments to validate the performance of our method on both artificially synthesized and real-world vulnerabilities. We compare our method against five vulnerability detection techniques, including three methods that adopt recurrent or convolutional neural networks and two more recent methods that leverage GNN models. The experimental results demonstrate that HybridNN outperforms all other methods under all used datasets, proving the effectiveness of our method.

The main contributions of this paper are as follows:

- We propose an information-simplified graph representation HCG, which helps alleviate the problem of insufficient training caused by data sparsity. It is highly compatible with GNN models, enabling better information extraction and information fusion.
- We propose HybridNN, a novel GNN-based vulnerability detection method, which applies a GNN model with a hierarchical attention mechanism to the newly designed heterogeneous graph representation HCG. The UD-Sampling method is designed to balance the pooling vectors generated by the GNN model, and the processed balanced samples are leveraged to retrain the model to further improve the performance.
- We conduct extensive experiments on both artificially synthesized and real-world vulnerability datasets to demonstrate the effectiveness of HybridNN in vulnerability detection tasks.

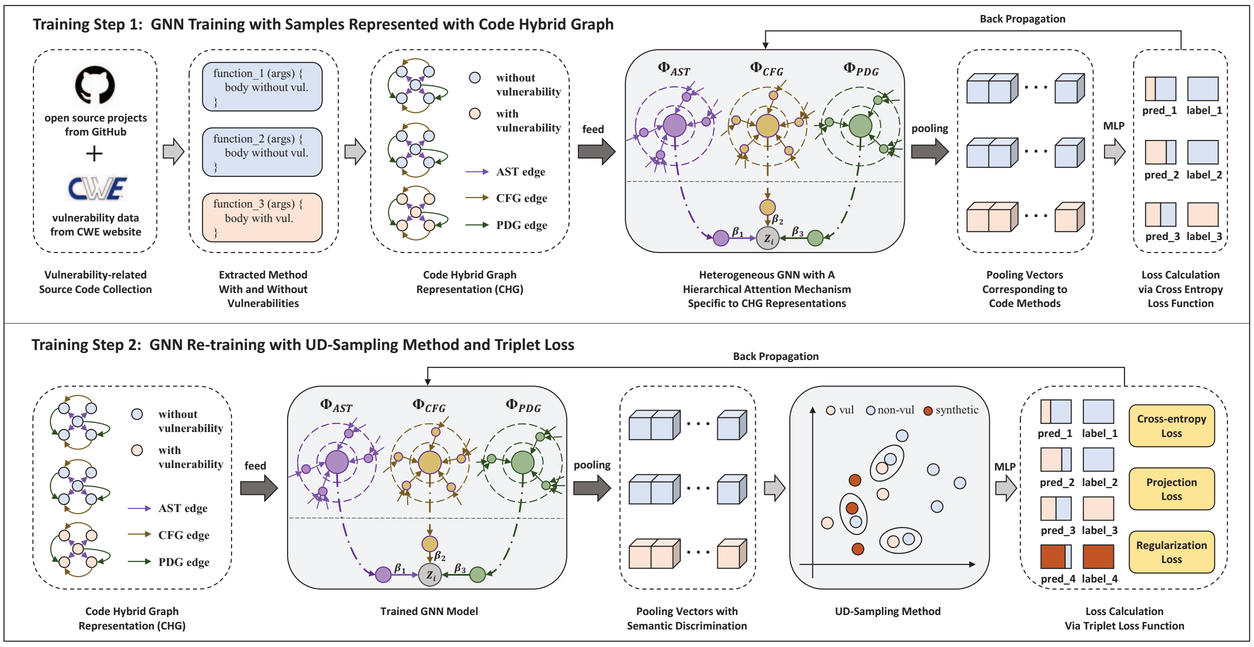
II. Proposed Approach

In this section, we introduce the detailed architectural of HybridNN. As shown in Figure 1, HybridNN consists of two rounds of training phases. In the first round, a sufficient amount of vulnerability data is collected, and a novel graph representation HCG is employed (Section II-A). Afterwards, a GNN model with a hierarchical attention mechanism which is suitable for handling heterogeneous graphs is utilized for deep semantic extraction, and a subsequent MLP (Multi-Layer Perceptron) architecture is connected for vulnerability prediction (Section II-B). In the second round, the input data is fed back into the trained GNN module to gain the corresponding pooling vectors. Furthermore, the UD-Sampling method is designed to balance the distributions of the samples in different classes by combining the up-sampling and down-sampling techniques (Section II-C).

A. Hybrid Code Graph Design and Dataset Construction

In recent years, many vulnerability detection studies have extracted features to improve the performance from different code representations [16]–[20]. AST (Abstract Syntax Tree) provides the parsed code elements with syntax relations, which is able to extract richer code semantics. Unfortunately, AST does not contain control flow and data flow information, limiting the performance on scenarios where the analysis of conditional expressions or variable propagation are required. CFG (Control Flow Graph) provides the analysis of the control structures with the triggered statements. However, since data flow analysis is not included, it is difficult to give convincing detection results for the vulnerabilities caused by the improper code element types and values. Afterwards, PDG (Program Dependency Graph) is composed of two sub-graphs DDG (Data Dependency Graph) and CDG (Control Dependency Graph), where the control flows and syntactic structures are not focused on. In recent years, some studies consider the CPG (Code Property Graph) representation. It leverages AST as the skeleton, and integrates information from CFG and PDG. Therefore, it contains richer syntactic and semantic features compared with the former ones. Nevertheless, it aggravates the problem of data sparsity, that is, the number of specific types of edges is too low to guarantee the adequate model training. Simultaneously, it also negatively impacts the feature learning for other edge types due to the concurrent information fusion via GNNs.

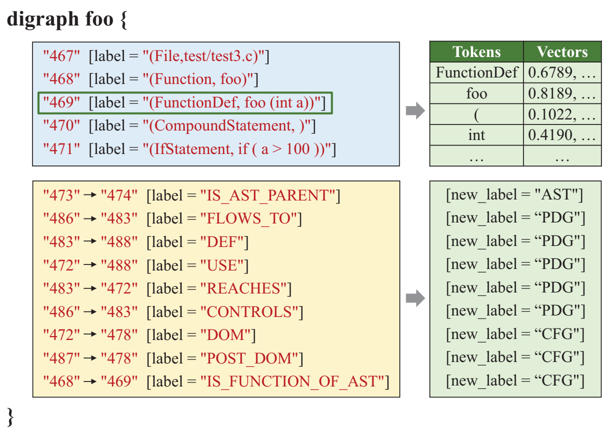
To address the above problems, we propose a novel code representation method called HCG (Hybrid Code Graph), which is simplified from the CPG representation. In total, CPG consists of 12 edge types as follows: *IS_AST_PARENT* connects AST nodes to their child nodes, while *IS_CLASS_OF* links the class members to the corresponding class nodes; *FLOWS_TO* connects two basic blocks with a control dependency, while *REACHES* is used for the data dependency; *DEF* connects the definition node in AST with the newly created symbol node in PDG, while *USE* gives the call relations; *CONTROLS* describe the control relations in PDG; *DECLARES* connects the declaration statement with all its



Training Step 1: GNN Training with Samples Represented with Code Hybrid Graph
Back Propagation
open source projects from GitHub
+
vulnerability data from CWE website
function_1 (args) { body without vul. }
function_2 (args) { body without vul. }
function_3 (args) { body with vul. }
without vulnerability
with vulnerability
AST edge
CFG edge
PDG edge
feed
Φ_{AST}
Φ_{CFG}
Φ_{PDG}
β_1
β_2
β_3
z_i
pooling
MLP
pred_1 label_1
pred_2 label_2
pred_3 label_3
Vulnerability-related Source Code Collection
Extracted Method With and Without Vulnerabilities
Code Hybrid Graph Representation (CHG)
Heterogeneous GNN with A Hierarchical Attention Mechanism Specific to CHG Representations
Pooling Vectors Corresponding to Code Methods
Loss Calculation via Cross Entropy Loss Function
Training Step 2: GNN Re-training with UD-Sampling Method and Triplet Loss
Back Propagation
without vulnerability
with vulnerability
AST edge
CFG edge
PDG edge
feed
pooling
vul
non-vul
synthetic
MLP
pred_1 label_1
pred_2 label_2
pred_3 label_3
pred_4 label_4
Cross-entropy Loss
Projection Loss
Regularization Loss
Code Hybrid Graph Representation (CHG)
Trained GNN Model
Pooling Vectors with Semantic Discrimination
UD-Sampling Method
Loss Calculation Via Triplet Loss Function

Fig. 1. An overview of HybridNN.

declaration members; *DOM* indicates the dominance relationship in CFG, while *POST_DOM* gives an opposite relation; *IS_FUNCTION_OF_AST* and *IS_FUNCTION_OF_CFG* provide connections between the function node and the possible AST or CFG node. We divide these edge types into three groups according to their functionalities. Specifically, the first 2 edge types are related to the AST representation, so they are divided into the AST group; the middle 6 edge types are related to data dependencies and control dependencies, so they are divided into the PDG group; and the last 4 types are related to the declarations and uses of the control flows, which are expected to be put into the CFG group.



```
digraph foo {
    "467"  [label = "(File,test/test3.c)"]
    "468"  [label = "(Function, foo)"]
    "469"  [label = "(FunctionDef, foo (int a))"]
    "470"  [label = "(CompoundStatement, )"]
    "471"  [label = "(IfStatement, if ( a > 100 ))"]

    "473" → "474"  [label = "IS_AST_PARENT"]
    "486" → "483"  [label = "FLOWS_TO"]
    "483" → "488"  [label = "DEF"]
    "472" → "488"  [label = "USE"]
    "483" → "472"  [label = "REACHES"]
    "486" → "483"  [label = "CONTROLS"]
    "472" → "478"  [label = "DOM"]
    "487" → "478"  [label = "POST_DOM"]
    "468" → "469"  [label = "IS_FUNCTION_OF_AST"]
}
```

Tokens	Vectors
FunctionDef	0.6789, …
foo	0.8189, …
(	0.1022, …
int	0.4190, …
…	…

```
[new_label = "AST"]
[new_label = “PDG"]
[new_label = “PDG"]
[new_label = “PDG"]
[new_label = “PDG"]
[new_label = “PDG"]
[new_label = “CFG"]
[new_label = “CFG"]
[new_label = “CFG"]
```

Fig. 2. An example to demonstrate the design of HCG.

The dataset used in this paper comes from two sources, one is the CWE website [1], and 4 vulnerability categories are employed in this paper that are either used in previous work [16] (i.e., CWE119 Memory-Buffer-Errors, CWE399 Resource-Management-Errors) or ranked as the most dangerous ones in the official website (i.e., CWE020 Input-Validation-Errors, CWE200 Sensitive-Information-Exposure-Errors) [2]. The other source is the dataset provided by previous studies, including *"FFmpeg+Qemu"* dataset from Devign [19] and *"Chromium+Debian"* dataset from ReVeal [20].

For data pre-processing, we first use an out-of-the-box static analysis tool Joern [3] to get the corresponding CPG representation for each sample from the datasets. As seen in Figure 2, the left side represents a CPG generated by Joern for the method *foo*. Specifically, the upper blue box indicates the nodes in the CPG, while the lower yellow box indicates the edges. To convert the graph node information into the numerical representation, we use NLTK [39] to split the *label content* (e.g., *"FunctionDef, foo (int a)"* in Figure 2) into a sequence of tokens, and then transform each token into a vector representation via word2vec [40]. The initial vector of the graph node is set as the average value of the token vectors in it. And for graph edge information, based on the grouping criteria mentioned above, we convert the original 12 edge types in the CPG into 3 edge types in the newly designed HCG. In this way, the source code is transformed into the graph embedding representation. Finally, we store the nodes, edges and the corresponding vulnerable or non-vulnerable labels as binary serialized files that are easy to read by the GNN models

[1]https://cwe.mitre.org/
[2]https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
[3]https://github.com/joernio/joern

described in the next Section. There are 301,198 collected samples in total, and the detailed statistics of the constructed datasets are shown in Table I.

TABLE I
THE STATISTICS OF DATASETS, WHERE VUL. MEANS VULNERABILITY

Datasets	# Samples with Vul.	# Samples without Vul.	# Samples in total
FFmpeg+Qemu	11608	13825	25433
Chromium+Debian	1658	16511	18169
CWE020	35565	84394	119959
CWE119	19636	39526	59162
CWE399	21824	55713	77537
CWE200	252	686	938

B. Attention Graph Neural Network for Hybrid Code Graphs

The traditional graph neural network model GCN [41] has achieved good results in a series of tasks, proving that combining local graph structure information with node features can help achieve good performance on tasks such as node classification. But it has two obvious disadvantages: (1) GCN cannot handle dynamic graph information, while training and testing can only be performed on the same graph; (2) GCN learns the information of each neighboring node on average, and cannot give them different weights, which limits the further improvement of model performance. In order to solve the above problems, this paper uses a GNN based on the attention mechanism. Specifically, assuming that the graph is represented as $G = (V, E)$, where V represents the set of nodes, and E represents the set of edges. Each node v_i in V is initialized as $h_i \in R^F$ by word2vec [40], where F indicates the dimension of the vector.

Afterwards, we use the shared weight matrix W to map the initial features of the nodes into high-dimensional representations. In order to consider the weights of edges, we calculate the correlation score (e.g., c_{ij}) for each of them (e.g., the edge e_{ij} between the nodes v_i and v_j) with the help of a single-layer feed-forward neural network f:

$$c_{ij} = f([Wh_i||Wh_j]) \tag{1}$$

where $[\bullet||\bullet]$ represents the vector concatenation operation. For each node, we adopt softmax activation function to generate the normalized weights for all its edges based on the obtained correlation scores:

$$\alpha_{ij} = \frac{exp(c_{ij})}{\sum_{t\in N_i} exp(e_{it})} \tag{2}$$

where N_i indicates the neighboring nodes of v_i. Next, the vector representation of v_i is updated by the following equation:

$$\acute{h_i} = \sigma\left(\sum_{t\in N_i} \alpha_{it} W h_t\right) \tag{3}$$

where σ indicates the selected activation function and $\acute{h_i}$ is the updated vector representation of v_i, which is generated by combining the weighted information of neighboring nodes.

To further improve the ability to aggregate the neighboring node information, a multi-head attention mechanism is leveraged, and the detailed calculation is as follows:

$$\acute{h_i} = ||_{k=1}^{K} \sigma\left(\sum_{t\in N_i} \alpha_{it}^{k} W^k h_t\right) \tag{4}$$

where $||$ represents the vector concatenation operation, K is set as the number of attention heads, α_{it}^{k} represents the attention score of the k^{th} head, W^k represents the corresponding trainable weight matrix. However, the dimension of the node vector increases rapidly as the number of attention heads rises, which may hinder the overall calculation process. To solve this problem, average pooling is a frequent used solution:

$$\acute{h_i} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{t\in N_i} \alpha_{it}^{k} W^k h_t\right) \tag{5}$$

However, it may lead to the problem of information loss, so it is necessary to choose an appropriate way in practice.

However, the GNN model constructed above does not consider the impact of the different edge types on information propagation and updates, which can also result in information loss. Since the graph data constructed in Section II-A contains three types of edges (i.e., AST, PDG and CFG), it is necessary to construct an advanced GNN model that is able to effectively operate on heterogeneous graphs. We leverage a GNN model HAN [38] to accomplish this task, which extracts heterogeneous information by designing a two-level attention mechanism. Specifically, the heterogeneous graph should be modeled by meta paths. A meta path Φ is defined as a special path connecting two entities, e.g., $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \ldots \xrightarrow{R_l} A_{l+1}$. Among them, A_i represents a node while R_i represents an edge. Since HCG representation contains 1 type of nodes and 3 types of edges, there are three corresponding meta paths. The core idea of the model is to calculate the information generated by different meta paths separately, which is then weighted and summed to update the vector representation of the current node. In our task, the first step is to extract the weighted information from the neighboring nodes of each meta path. The Equations 1-5 can be reused to conduct the calculation, where N_i should be replaced with N_i^{Φ}. The latter symbol represents the neighboring nodes only connected via the meta path Φ, rather than those connected by all edges. Assuming that the vector concatenation handling shown in Equation 4 is adopted, z_i^{Φ} is defined as the weighted information extracted from the neighboring nodes on the meta path Φ:

$$z_i^{\Phi} = ||_{k=1}^{K} \sigma\left(\sum_{t\in N_i^{\phi}} \alpha_{it}^{k\Phi} W^k h_t\right) \tag{6}$$

The above calculation process leverages a node-level attention mechanism. Given a set of meta paths $\{\Phi_1, \Phi_2, \ldots, \Phi_P\}$, after the calculation process, a node embedding group with specific semantics can be constructed as $\{Z_{\phi_1}, Z_{\phi_2}, \ldots, Z_{\phi_P}\}$.

After obtaining the weighted information at the node level, the impacts of different meta paths are calculated with another

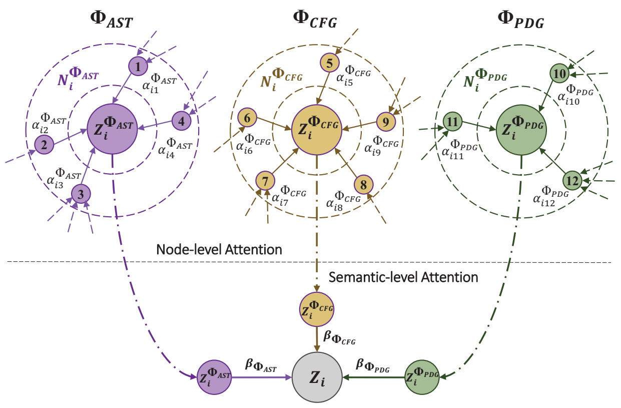


Fig. 3. GNN with a hierarchical attention mechanism specific to HCG.

attention mechanism at the semantic level. Specifically, the semantic embedding is first transformed by a dense layer with a tanh activation function. Then the trainable semantic attention vector q is leveraged to calculate the importance of each meta path. Since each node may have multiple neighboring nodes under a specific meta path, the all relevant calculated results should be averaged. The calculation process is as follows:

$$w_{\Phi_p} = \frac{1}{|S_{\Phi_p}|} \sum_{j \in S_{\Phi_p}} q^T \cdot tanh(W_s \cdot z_j^{\Phi_p} + b) \tag{7}$$

where W_s is the weight matrix (the subscript s indicates *semantic-level*), and b is the bias term. In addition, S_{Φ_p} represents the neighboring nodes under the meta path Φ_p. Note that, the above parameters are shared for all meta paths and node vector representations. After the weighted score w_{Φ_p} of each meta path is obtained, it is expected to be further normalized by the softmax activation function:

$$\beta_{\Phi_p} = \frac{exp(w_{\Phi_p})}{\sum_{p=1}^{P} exp(w_{\Phi_p})} \tag{8}$$

This can be interpreted as the contribution of the meta path Φ_p to the vulnerability detection task. The higher β_{Φ_p} is, the more important Φ_p is. The final representation z_i of the node v_i is the weighted sum by considering all meta paths:

$$z_i = \sum_{p=1}^{P} \beta_{\Phi_p} \cdot z_i^{\Phi_p} \tag{9}$$

Finally, with the assistance of the node-level and semantic-level attention mechanisms, the richer semantic information from the heterogeneous graph is able to be extracted.

C. Design of UD-Sampling Method

In real-world development, the number of code without vulnerabilities generally far exceeds those with vulnerabilities. For instance, in *Chromium+Debian* dataset, the ratio reaches about 10:1. This would make the model more prone to predict the unseen code snippets as non-vulnerable, so as to reduce the task performance. Sampling method is an important way to solve this problem, which mainly includes up-sampling and down-sampling methods. In vulnerability detection task, up-sampling and down-sampling methods are expected for vulnerable and non-vulnerable ones, respectively. However, some disadvantages cannot be neglected, such as the information loss issue caused by down-sampling method and data noise problem caused by up-sampling method.

SMOTE [42] is an up-sampling method that synthesizes approximate samples for the minority category based on the original vector representations. In Figure 4-(a), black dots represent the vulnerable code while white dots are for non-vulnerable code. As shown in Figure 4-(b), SMOTE synthesizes new samples (i.e., the blue dots) by weighted average the vector representations of two neighboring dots in the same category. However, there are some possibilities that the newly synthesized samples are closer to the existing samples in the opposite category, which would make it harder to find the classification boundary (shown in Figure 4-(b)). TomekLink [43] is a down-sampling method that removes samples in a pairwise manner. A pair of samples will be removed if they satisfy the following conditions: (1) each member of the pair is the nearest neighbor to the other one; (2) the two members are from different categories. After processing, the remaining samples in different categories are far enough apart and easier to be classified. However, this may lead to information loss. Furthermore, after the processing of TomekLink, the number of vulnerable samples are still less than those of non-vulnerable samples.

To fuse the advantages of both methods, we propose UD-Sampling. Specifically, we first leverage SMOTE [42] to synthesize samples for the minority category until the number is the same as the majority category, and then use Tomek-Link [43] to remove sample pairs. This brings three benefits: (1) with TomekLink processing, some of the low-quality samples synthesized by SMOTE have greater possibilities to be removed due to their excessive similarity with the opposite category; (2) not all removed samples parsed by TomekLink are from the original data, but part of them are from the synthesized ones, reducing the risk of information loss; (3) after processing, the numbers of samples in different categories are the same, which facilitates model training. As seen in Figure 4-(c) and 4-(d), after UD-Sampling processing, a clear boundary between two categories is detected.

Although the UD-Sampling method partially increase the gap between different classes, effectively distinguishing positive and negative samples remains challenging. Therefore, we consider the triplet loss function that is also used in the previous studies [44], [45] to further enhance the discrimination between different classes from three aspects: (1) cross entropy loss used as regular loss calculation for classification task, (2) projection loss to enhance the affinity between vectors within the same class while reduce it from different classes, (3) regularization term to penalize significant fluctuations in the vectors. As illustrated in Figure 1, UD-Sampling method and triplet loss are only employed in the second round of

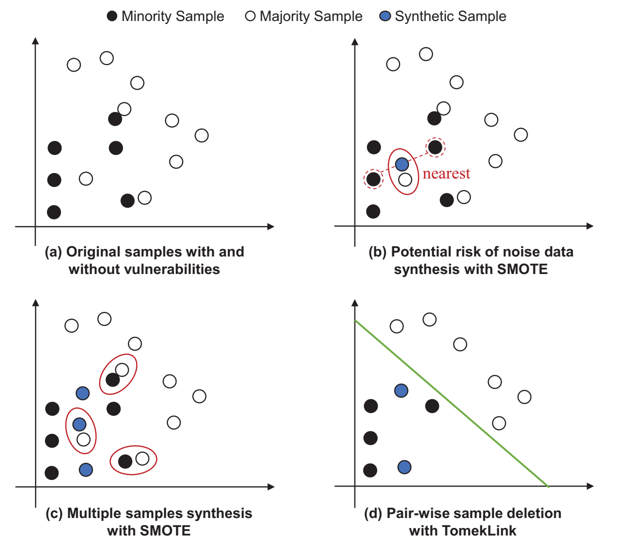

● Minority Sample
○ Majority Sample
● Synthetic Sample
nearest
(a) Original samples with and without vulnerabilities
(b) Potential risk of noise data synthesis with SMOTE
(c) Multiple samples synthesis with SMOTE
(d) Pair-wise sample deletion with TomekLink

Fig. 4. UD-Sampling method designed to balance the sample distributions in different classes, where the majority samples represent the non-vulnerable code, and the minority samples represent the vulnerable code.

training. Specifically, we first train the end-to-end neural model, after which the model has acquired initial knowledge for vulnerability detection. Subsequently, the input data is fed back into the trained GNN module to obtain the corresponding pooling vectors, which are then sampled by UD-Sampling method to make the sample distribution more balanced. The triplet loss function is used in the back propagation step to optimize the model parameters.

III. Evaluation

A. Benchmark Dataset

In Section II-A, we give a detailed description of the construction process of the datasets, and the statistics are shown in Table I. Consistent with the previous studies, each of the six datasets is split into 3 parts including training, validation, and test sets in a ratio of 8:1:1. Among them, the 4 CWE datasets are artificially synthesized, which only contain limited types of vulnerability information and the distribution of the samples is more balanced. The remaining 2 datasets consist of the vulnerable and non-vulnerable source code in real-world software developments. Thus, we use these two sources of datasets to validate the performance of the proposed method from a wider perspective.

B. Evaluation Metrics

Three widely-used evaluation metrics in vulnerability detection studies are also leveraged in our experiments:

Precision: Precision is the proportion of true positive predictions out of all positive predictions made by the model. It is calculated as $Precision = TP/(TP + FP)$, where TP is the number of true positive predictions and FP is the number of false positive predictions.

Recall: Recall is the proportion of actual positive samples that are correctly identified as positive by the model. It is calculated as $Recall = TP/(TP + FN)$, where FN is the number of false negative predictions.

F1 score: F1 score combines both precision and recall into a single metric which is calculated as $F1\ score = 2 * (Precision * Recall)/(Precision + Recall)$. F1 score ranges from 0 to 1, with a higher value indicating better model performance.

C. Experimental Settings

The experiments are performed under the Ubuntu 20.04 operating system. The used languages, tools, frameworks and their corresponding versions are as follows: Python version 3.8, PyTorch version 1.8, DGL [46] version 0.7.2 (used to construct GNN models). In addition, Joern version 0.2.5[4] is leveraged to generate CPGs, which relies on the Java 1.8 environment. In terms of hardware for model training, NVIDIA Tesla V100 GPUs with 32GB memory are leveraged.

TABLE II
Settings of the key modules for experiments, where dim. represents dimension

Modules	Parameters	Values
word2vec [40]	window size	5
	vector dimension	100
GNN Model	input dimension	100
	pooling vector dim.	128
	learning rate	1e-4
	GNN layers	3
	attention heads	2,2,1
	dropout rate	0.2
Triplet Loss	MLP layers	3
	α	0.5
	β	0.2
	γ	1e-3

Table II shows the detailed parameter settings used in HybridNN at different stages. Note that, the parameters α, β and γ are used for adjusting the weights among three parts in the triplet loss function [44], [45]. In addition, during model training, we adopt the widely-used Adam optimizer, set the batch size as 512, and the number of training epochs as 500. Moreover, the F1 score is used as a guide for the performance determination, and when a higher F1 score occurs, the current model parameters are preserved. On the other hand, when the F1 score does not increase and the number of the consumed epochs is greater than 50, the training step will be stopped. Note that, all the hyper-parameters used in the supervised training phases (i.e., except for the token vector pre-training phase with word2vec [40]) are determined by the grid search algorithm based on the validation set.

D. Results and Discussion

RQ1: How does HybridNN perform in vulnerability detection task? To evaluate the performance of HybridNN, we

[4]https://github.com/VulDetProject/ReVeal/tree/master/code-slicer/joern

compare it with 5 baseline methods, the detailed descriptions of which can be seen in Section V. Since previous studies such as VulDeePecker [16] have already demonstrated the significant performance improvement compared to traditional pattern-based and code similarity-based methods, we only leverage these learning-based methods with better performance as our baseline methods. First, Table III shows the experimental results of all 6 vulnerability detection methods in 4 CWE datasets, where the values represent the corresponding F1 scores (given that the F1 scores are relatively close to 1.0, the detailed results of Precision and Recall metrics are not released). It can be clearly seen that whether the early-proposed method VulDeePecker [16] or the recently-proposed methods such as ReVeal [20] have achieved remarkable performance on these datasets due to the limited types and more balanced distributions of the synthesized vulnerabilities. Even so, HybridNN can still outperform all baseline models, proving the effectiveness of our method on synthesized vulnerability detection tasks.

Compared with the performance on artificially synthesized vulnerabilities, the effectiveness of the model on vulnerabilities from real-world software development attracts more attention of researchers and developers. As shown in Table IV, HybridNN achieves the best results on both datasets, proving the effectiveness of our method. However, no matter on the dataset ”FFmpeg+Qemu” or the dataset ”Chromium+Debian”, the F1 score of each method decreases rapidly compared with that on the 4 CWE datasets mentioned above, indicating that it is more difficult to detect real-world vulnerabilities compared with the artificially synthesized ones. Moreover, by relating the design of each method to its corresponding detection performance, it can be found that the performance of the methods without leveraging graph representations are the worst (including VulDeePecker [16] and Russell [18]), while the performance of the methods which adopt graph representations for static analysis is significantly better than the former (including SySeVR [17], Devign [19], ReVeal [20] and HybridNN). Therefore, the experimental results show that converting the program source code into a graph representation with more structured syntactic and semantic information is beneficial for mining the deeper information for vulnerability detection tasks.

TABLE III
EXPERIMENTAL RESULTS ON CWE DATASETS

Techniques	CWE020	CWE119	CWE399	CWE200
VulDeePecker [16]	0.92	0.80	0.90	0.91
SySeVR [17]	0.93	0.92	0.92	0.93
Russell [18]	0.93	0.90	0.91	0.93
Devign [19]	0.94	0.93	0.92	0.93
ReVeal [20]	0.94	0.93	0.93	0.94
HybridNN	**0.95**	**0.94**	**0.95**	**0.95**

Among the four approaches that utilize graph representation, SySeVR employs PDG analysis and utilizes BiGRU models for vulnerability detection. Although it incorporates graph representation, the performance is not as competitive as the other three methods due to the absence of more effective GNN models for mining deeper graph information. Devign adopts a combination of GGNN [37] and TextCNN [47] models to delve into the information embedded in the CFG and PDG, and gains a significant improvement on performance compared with SySeVR. Similarly, ReVeal also employs the GGNN model in conjunction with CPGs. It performs classification prediction on the pooling vectors generated by GGNN model. In contrast to Devign, ReVeal applies the SMOTE method to up-sample the data in the minority class, effectively balancing the sample distributions across different categories. Furthermore, it strengthens the classification capability of the model through a triplet loss function. The experimental results show that ReVeal outperforms Devign, thereby validating the effectiveness of its proposed methodology.

TABLE IV
EXPERIMENTAL RESULTS ON TWO REAL-WORLD VULNERABILITY DATASETS, WHERE PRE. REPRESENTS THE PRECISION METRIC

Techniques	FFmpeg+Qemu			Chromium+Debian		
	Pre.	Recall	F1	Pre.	Recall	F1
VulDeePecker [16]	0.49	0.27	0.35	0.19	0.14	0.17
SySeVR [17]	0.50	0.66	0.56	0.24	0.42	0.31
Russell [18]	0.55	0.41	0.45	0.26	0.12	0.16
Devign [19]	0.52	0.63	0.57	0.33	0.32	0.32
ReVeal [20]	0.55	0.73	0.62	0.31	**0.58**	0.40
HybridNN	**0.58**	**0.73**	**0.65**	**0.38**	0.52	**0.44**

This paper proposes a novel code graph representation called HCG, which merges homogeneous type edges for the combination of 3 existing graphs (i.e., AST, PDG and CFG). It utilizes a heterogeneous GNN with a hierarchical attention mechanism to explore deep syntactic connections and semantic information in the graph. Additionally, a new sampling method called UD-Sampling is introduced, which implements an iterative process of up-sampling and down-sampling to reduce the quantity difference between different categories while preserving essential information. As shown in Table IV, the performance of HybridNN outperforms all baselines. Furthermore, we collect results from 10 sets of experimental runs and leverage t-test to assess the significance of differences between the performance of HybridNN and baselines, the results of which indicate that HybridNN performs significantly better than all baselines on F1 score metric ($p\text{-}value < 0.05$). Since neither ”FFmpeg+Qemu” [5] nor ”Chromium+Debian” [6] datasets provides detailed information such as the type or root cause of the vulnerability corresponding to each issued sample, it is difficult to further evaluate the more fine-grained performance of HybridNN on different vulnerability types, which is expected to be improved in the future.

RQ2: Is the newly designed HCG representation proposed in this paper more suitable for vulnerability detection tasks?

We propose a novel graph representation called HCG for

[5]https://sites.google.com/view/devign
[6]https://bit.ly/3bX30ai

code analysis, which integrates information from three types of graphs, including AST, CFG and PDG. In this RQ, we will discuss the necessity of using HCG for vulnerability detection tasks. Table V shows the experimental results obtained by training models on different graph representations. The experiments are conducted without employing any sampling methods. It can be observed that both on the dataset ”FFmpeg+Qemu” and ”Chromium+Debian”, the model variant with HCG achieves the highest F1 scores (i.e., an increase of 4 and 6 percentage points, respectively). The results demonstrates that compared to the three traditional graph representations, HCG is able to provide richer syntactic and semantic information on vulnerability detection tasks.

TABLE V
COMPARATIVE ANALYSIS WITH DIFFERENT GRAPH REPRESENTATIONS

Datasets	Graph Representations	Precision	Recall	F1 Score
FFmpeg Qemu	AST	**0.59**	0.57	0.57
	CFG	0.57	0.61	0.59
	PDG	0.56	0.62	0.59
	HCG	0.55	**0.74**	**0.63**
Chromium Debian	AST	0.34	0.25	0.29
	CFG	0.34	0.35	0.34
	PDG	0.34	0.28	0.31
	HCG	**0.39**	**0.35**	**0.37**

On the other hand, due to the more imbalanced ratio of positive and negative samples in the dataset ”Chromium+Debian” compared to ”FFmpeg+Qemu” (i.e., 0.10 and 0.84 calculated from Table I, respectively), the task is more difficult in ”Chromium+Debian” than the other. This is the reason why all model variants yield lower F1 scores on this dataset. Generally, in most of the real-world software development scenarios, only a small portion of the code contains vulnerabilities, hence the challenge of dealing with the imbalanced data distribution is a common issue in this research area. Therefore, on dataset ”Chromium+Debian”, the fact that the model variant with HCG outperforms all other model variants by more than 5 percentage points on all three metrics provides strong evidence for the effectiveness of our method in performing vulnerability detection tasks in such realistic and challenging scenarios with extremely imbalanced samples.

RQ3: How do different GNN models affect the experimental results?

In this RQ, we examine the impact of different GNN models on the experimental results. We explore three types of GNN models including GGNN [37], RGCN [48] and HAN [38] (the model leveraged in this paper). Following each GNN module, we append an MLP layer with softmax activation function to perform binary classification predictions. The number of layers of each GNN module is set as 3. Moreover, The cross-entropy loss function is adopted to measure the discrepancy between predicted values and ground truth labels. Note that, the experiments at this part are conducted on the dataset ”Chromium+Debian”. As shown in Table VI, HAN [38] gains a higher F1 score compared to GGNN [37] and RGCN [48]. This result demonstrates that using different weights for different edge types during the propagation and updating of heterogeneous graph information can help capture the contributions of different edges in specific tasks. It is evident that this approach is beneficial for vulnerability detection tasks as well. On the other hand, it is difficult for the traditional GNN models to achieve such effectiveness, since they do not differentiate the information in heterogeneous graphs.

TABLE VI
COMPARATIVE ANALYSIS WITH DIFFERENT GNN MODELS

Techniques	Precision	Recall	F1 Score
GGNN [37]	0.33	0.32	0.32
RGCN [48]	0.25	**0.57**	0.35
HAN [38]	**0.39**	0.35	**0.37**

RQ4: What is the impact of different sampling methods on the detection results?

In this RQ, we discuss the impact of different sampling methods on the vulnerability detection results. We construct three experimental variants including no sampling method, SMOTE [42] and UD-Sampling. The experiments at this part are also conducted on the dataset ”Chromium+Debian”. The results, presented in Table VII, show that without using any sampling method the model gains higher precision but lower recall. This is due to the imbalanced distribution of samples across different categories in the dataset, leading the trained model to predominantly predict input cases as non-vulnerable code. In contrast, the SMOTE method synthesizes artificial samples that contain vulnerabilities, equalizing the sample sizes across different categories and helping the model to avoid naive non-vulnerable predictions, which improves the recall to some extent. However, the synthesized samples introduced by SMOTE may also introduce noise, which may affect the performance on precision. Simultaneously, the proposed UD-Sampling method in this paper can effectively combine the advantages of up-sampling and down-sampling methods. On the one hand, it employs up-sampling (i.e., SMOTE [42] method) to synthesize positive samples, balancing their quantity with negative samples. On the other hand, it utilizes down-sampling (i.e., TomekLink [43] method) to remove pairs of positive and negative samples that are too similar in the terms of vector representations. This helps maintain balanced proportions of categories while removing samples that are more likely to contain noise. As a result, it enhances the recall metric while almost maintaining the precision performance, leading to the improvement of F1 score. This experiment proves the effectiveness of the proposed UD-Sampling method in vulnerability detection tasks.

IV. THREATS AND LIMITATIONS

One threat to external validity is that only the C/C++ language is considered in this paper. In fact, HybridNN is not tied to a specific programming language. During the pre-processing stage, we transform the source code into a CPG representation.

TABLE VII
COMPARATIVE ANALYSIS WITH DIFFERENT SAMPLING METHODS

Techniques	Precision	Recall	F1 Score
No Sampling	**0.39**	0.35	0.37
SMOTE [42]	0.31	**0.62**	0.41
UD-Sampling	0.38	0.52	**0.44**

Therefore, it is possible to construct the corresponding graphs via static analysis tools specific to other languages. Another threat is the benchmarks used for validation. To ensure the experimental results robustly reflecting the performance of HybridNN, we collect 301,198 samples from both artificially synthesized and real-world vulnerabilities. Finally, HybridNN outperforms all baselines on all datasets, proving the effectiveness of our method. However, it remains challenging to determine the performance on other datasets, which is planed to be discussed in the future.

One threat to internal validity is the robustness of our implemented code. This primarily refers to the code for the construction and training of the deep neural networks, as various complex steps such as module combinations and numerical computations are involved. To mitigate this risk, this part of code is implemented based on the well-established frameworks PyTorch and DGL. Additionally, for the preprocessing scripts, we also utilize mature third-party tools, such as Joern to generate CPGs. These attempts minimize the potential issues and enhance the reliability of our method.

V. Related Work

A. Code Representation for Vulnerability Detection

The choice of how to transform the source code into a suitable form as the model input is a crucial consideration in method design. Some studies treat the source code as a token sequence and employ natural language processing techniques for vulnerability detection [16], [18], [24]. However, this representation is prone to losing the rich hierarchical structural information. Subsequent approaches have attempted to incorporate representations such as AST, CFG and PDG, which capture more complex relationships among code elements from syntactic structure, control flow, and element dependency perspectives [17], [20], [22], [23]. The more recently proposed CPG [19], [36] integrates these representations into a single graph, and it has been demonstrated to enhance the method performance [20]. Additionally, some studies combine the aforementioned representation methods to capture comprehensive semantics at different levels [12], [21].

B. Deep Learning-based Vulnerability Detection

The introduction of deep learning techniques has significantly improved the performance of vulnerability detection tasks. VulDeePecker [16] incorporates source code and data/control dependencies into the program slices, and then utilizes a BiLSTM-based neural network model to perform binary classification. SySeVR [17] utilizes program slicing to analyze the program dependency graph generated from the source code, and supports the integration of sliced information from custom function calls. BiGRU-based model is employed to extract code information and predict vulnerabilities. Russell [18] labels the source code into the corresponding matrix. It employs a CNN model combined with ensemble learning strategy, and leverages a random forest classifier to facilitate vulnerability detection. Devign [19] converts the program source code into a code property graph while considering data/control dependency relationships. It incorporates the GGNN [37] to extract syntactic and semantic information encoded in the code. Finally, it connects a TextCNN model to perform vulnerability detection tasks. ReVeal [20] converts the source code into a code property graph and utilizes the GGNN model to extract graph embedding vectors. It employs the SMOTE sampling method to balance the sample distribution.

VI. Conclusion

In this paper, we propose a novel vulnerability detection method HybridNN, based on graph neural networks. It leverages a newly designed graph representation HCG, specifically tailored for training GNNs. HybridNN utilizes a GNN model with a hierarchical attention mechanism to capture the varying impacts of different edge types and neighboring nodes on the node representation. Afterwards, information propagation is performed to update node representations iteratively. Furthermore, to address the issue of imbalanced sample distributions, we introduce a novel sampling method UD-Sampling, which combines the advantages of up-sampling and down-sampling techniques. This approach performs sampling on the pooling vectors generated by the trained GNN model. Finally, extensive experimental are conducted, and the experimental results demonstrate the superior performance of our method on both artificially synthesized and real-world vulnerabilities.

Data availability: The datasets used in this paper are publicly available at: **https://github.com/mxx1219/HybridNN**.

VII. Acknowledgement

This work was supported partly by National Key Research and Development Program of China (No. 2022YFB4502003), partly by National Natural Science Foundation of China (No. 62072017) and Huawei. This work is also partly supported by State Key Laboratory of Software Development Environment.

References

[1] N. S. Harzevili, A. B. Belle, J. Wang, S. Wang, Z. Ming, N. Nagappan *et al.*, "A survey on automated software vulnerability detection using machine learning and deep learning," *arXiv preprint arXiv:2306.11673*, 2023.

[2] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, "Neucheck: A more practical ethereum smart contract security analysis tool," *Software: Practice and Experience*, vol. 51, no. 10, pp. 2065–2084, 2021.

[3] C. Gao, G. Wang, W. Shi, Z. Wang, and Y. Chen, "Autonomous driving security: State of the art and challenges," *IEEE Internet of Things Journal*, vol. 9, no. 10, pp. 7572–7595, 2021.

[4] Q. Luo, Y. Cao, J. Liu, and A. Benslimane, "Localization and navigation in autonomous driving: Threats and countermeasures," *IEEE Wireless Communications*, vol. 26, no. 4, pp. 38–45, 2019.

[5] K. Goseva-Popstojanova and J. Tyo, "Experience report: security vulnerability profiles of mission critical software: empirical analysis of security related bug reports," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 152–163.

[6] N. S. Harzevili, J. Shin, J. Wang, and S. Wang, “Characterizing and understanding software security vulnerabilities in machine learning libraries,” *arXiv preprint arXiv:2203.06502*, 2022.

[7] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, “The matter of heartbleed,” in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[8] J. Wetter and N. Ringland, “Understanding the impact of apache log4j vulnerability,” *Retrieved January*, vol. 11, p. 2022, 2021.

[9] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, “Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1456–1468.

[10] S. Jeon and H. K. Kim, “Autovas: An automated vulnerability analysis system with a deep learning approach,” *Computers & Security*, vol. 106, p. 102308, 2021.

[11] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, “Software vulnerability discovery via learning multi-domain knowledge bases,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2019.

[12] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[13] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, “Combining graph-based learning with automated data collection for code vulnerability detection,” *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[14] N. Ziems and S. Wu, “Security vulnerability detection using deep learning natural language processing,” in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2021, pp. 1–6.

[15] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “μvuldeepecker: A deep learning-based system for multiclass vulnerability detection,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.

[16] 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,” *arXiv preprint arXiv:1801.01681*, 2018.

[17] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[18] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.

[20] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet,” *IEEE Transactions on Software Engineering*, 2021.

[21] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, “Reentrancy vulnerability detection and localization: A deep learning based two-phase approach,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[22] Y. Li, S. Wang, and T. N. Nguyen, “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*, 2021, pp. 292–303.

[23] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, “Vulcnn: An image-inspired scalable vulnerability detection system,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2365–2376.

[24] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[25] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, “Bugs as deviant behavior: A general approach to inferring errors in systems code,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[26] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.

[27] D. A. Ramos and D. Engler, “{Under-Constrained} symbolic execution: Correctness checking for real code,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[28] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[29] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing.” in *NDSS*, 2018.

[30] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[33] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.

[34] D. Opitz and R. Maclin, “Popular ensemble methods: An empirical study,” *Journal of artificial intelligence research*, vol. 11, pp. 169–198, 1999.

[35] L. Breiman, “Random forests,” *Machine learning*, vol. 45, pp. 5–32, 2001.

[36] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[37] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” *arXiv preprint arXiv:1511.05493*, 2015.

[38] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, “Heterogeneous graph attention network,” in *The world wide web conference*, 2019, pp. 2022–2032.

[39] E. Loper and S. Bird, “Nltk: The natural language toolkit,” *arXiv preprint cs/0205028*, 2002.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[41] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[42] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[43] I. Tomek, “Two modifications of cnn,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-6, no. 11, pp. 769–772, 1976.

[44] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[45] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, “Metric learning for adversarial robustness,” *Advances in neural information processing systems*, vol. 32, 2019.

[46] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai *et al.*, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.

[47] Y. Kim, “Convolutional neural networks for sentence classification,” *arXiv preprint arXiv:1408.5882*, 2014.

[48] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 2018, pp. 593–607.