The Journal of Systems & Software 206 (2023) 111832

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Static vulnerability detection based on class separation☆

Chunyong Zhang, Yang Xin*

Beijing University of Posts and Telecommunications, Beijing, China
National Engineering Lab fo DBR, Beijing, China

A R T I C L E   I N F O

Article history:
Received 12 April 2023
Received in revised form 26 July 2023
Accepted 2 September 2023
Available online 9 September 2023

Keywords:
Vulnerability detection
Code property graph
Jump structure
Graph attention network
Class separation

A B S T R A C T

Software vulnerability detection is a key step to prevent the system from being attacked. However, tens of thousands of codes have brought great challenges to engineers, so we urgently need an automatic and intelligent vulnerability detection method. The existing vulnerability detection model based on deep learning has the problem that it is difficult to separate the features of vulnerable and neutral code. Based on the code data drive, this paper proposes a static vulnerability detection method SDV(**S**tatically **D**etecting **V**ulnerability) for C\C++ programs. SDV is a function-level vulnerability code detection method. This paper uses a code property graph to represent the code and decouples the feature extractor and the classifier. In the graph feature extraction stage, we use Jump Graph Attention Network layers and convolutional pooling layers. Their combination can not only prevent the over-smoothing problem but also separate the sample classes deeply. Finally, on the chrdeb dataset, SDV outperforms state-of-the-art function-level vulnerability detection methods by 52.3%, 15.9%, and 39.6% in Precision, Recall, and F1-Score, respectively. On the real project sard, the number of vulnerabilities detected by SDV is 10.7 times more than Reveal.

1. Introduction

Software vulnerabilities refer to design errors and coding defects that occur in the software life cycle. Software vulnerability detection is actually to detect code vulnerabilities (Wu et al., 2012). According to the report of the USA National Vulnerability Database (NVD) (Booth et al., 2013), the number of vulnerability CVEs in 2021 will be as high as 18,378, including 3,646 high-risk vulnerabilities, 11,767 medium-risk vulnerabilities, and 2,965 low-risk vulnerabilities. These vulnerabilities have led to large-scale data breaches and extortion incidents, resulting in huge economic losses. Therefore, software vulnerability detection has always been critical in preventing network attacks and data loss.

Early vulnerability detection techniques are divided into static analysis methods and dynamic analysis methods according to whether the program runs during the detection process. The static analysis method generally analyzes the file structure, lexical, syntax, data dependencies, and control dependencies of the source code before the program runs, and finally determines whether there are logical problems to find vulnerabilities. Static analysis methods mainly include static taint analysis technology (Ceara et al., 2009), static symbol execution technology (King, 1976), binary file comparison technology (Gao et al., 2008), and manual testing technology (Dukes et al., 2013). Dynamic analysis technology refers to determining whether the code is a vulnerability through the running state, function call, and execution path of the program during the running process of the program. Commonly used dynamic analysis techniques include fuzzing (Peng et al., 2018), dynamic symbolic execution techniques (Xie et al., 2009), and dynamic taint analysis techniques (Ganesh et al., 2009). Static analysis technology formulates rules based on expert knowledge, which leads to the problems of low detection accuracy and high false positive rate, while dynamic analysis technology has problems of low code coverage and high false negatives, and dynamic analysis technology when the program cannot run normally is ineffective.

With the growing of code data, data-driven code vulnerability detection methods have been proposed. Deep learning-based vulnerability detection methods have been proven to be effective and feasible (Russell et al., 2018; Li et al., 2021c; Zou et al., 2019; Li et al., 2021d, 2018; Lin et al., 2019; Liu et al., 2019; Lin et al., 2018; Wei et al., 2021; Chakraborty et al., 2021; Zhou et al., 2019; Ye et al., 2020; Cao et al., 2021). The researcher also focuses on using machine learning methods to collect and preprocess code datasets (Zheng et al., 2021b; Wang et al., 2020), they reduce the steps of manual rules and use machine learning models to mine potential vulnerability features. Many methods still have limitations that lead to poor vulnerability detection performance.

Russell. et al. utilized convolutional neural networks to process input text sequences and perform vulnerability detection

☆ Editor: Leandro L. Minku.
* Corresponding author at: Beijing University of Posts and Telecommunications, Beijing, China.
E-mail address: yangxin@bupt.edu.cn (Y. Xin).

https://doi.org/10.1016/j.jss.2023.111832
0164-1212/© 2023 Elsevier Inc. All rights reserved.

tasks (Russell et al., 2018). The two teams of Li and Lin are committed to converting the code into a token sequence, and then extracting features through LSTM or Bi-LSTM for vulnerability detection (Li et al., 2021c; Zou et al., 2019; Li et al., 2021d, 2018; Lin et al., 2019; Liu et al., 2019; Lin et al., 2018; Wei et al., 2021), which is a natural language processing method. Li et al. convert the code into the form of slices and only use part of the information of the code (Li et al., 2021c; Zou et al., 2019; Li et al., 2021d, 2018). Lin et al. consider the code as text, ignoring the logic and structure of the source code (Lin et al., 2019; Liu et al., 2019; Lin et al., 2018; Wei et al., 2021). Vulnerabilities are sometimes subtle flaws that require feature extraction from multiple dimensions. In addition, wan. et al. conduct a thorough structural analysis of the code from three aspects: attention analysis, word embedding, and syntax tree induction. They found that incorporating the grammatical structure of the code into the pre-training process may contribute to better code representation. Therefore, it is necessary for us to choose to use the graph structure to represent the code (Wan et al., 2022a). This paper uses the code property graph. The code property graph contains data dependencies, control dependencies, and other semantic relationships of the code, providing rich code-behind information for deep learning (Yamaguchi et al., 2014).

In terms of model selection, methods based on natural language processing cannot handle non-sequential features, and gated graph neural networks cannot select important neighbor node information for aggregation (Chakraborty et al., 2021; Zhou et al., 2019; Ye et al., 2020; Cao et al., 2021). Reveal (Chakraborty et al., 2021), Devign (Zhou et al., 2019), Poem (Ye et al., 2020), and BGNN4VD (Cao et al., 2021) have devoted themselves to extracting code features with sequential Gated Graph Neural Network (hereafter, GGNN) (Li et al., 2015) and its variants in vulnerability detection research. Research works such as GSM (Zhang et al., 2022), LinSDV (Hin et al., 2022), and ACGVD (Li et al., 2021a) use Graph Attention Network (hereafter, GAT) (Velickovic et al., 2017) to extract code vulnerability features and distinguish the importance of neighbor nodes.

However, The above vulnerability code detection method only uses the graph neural network model with the sequential structure, which will cause over-smoothing problems. That is, the vulnerability code feature and the neutral code feature tend to be in the same feature space area, and finally the vulnerability and the neutral code feature cannot be separated.

In the case of poor sample data, if the feature extraction model cannot effectively distinguish the features of vulnerable samples from those of neutral samples, the performance of the final vulnerability detection task will become extremely poor. Based on the above problems, we propose a function-level C\C++ language code vulnerability detection method SDV, which does not target some special types of vulnerabilities, but judges whether the function may contain a vulnerability. First, SDV converts the code into a code property graph. The code property graph is a commonly used graph structure representation method of source code. It provides a comprehensive and compact representation of code that includes elements of control flow and semantics. Next, we initialize node features using word embedding methods. The Jump Graph Attention Network aggregates the feature information of multi-hop neighbor nodes into the target node through the attention mechanism and enriches the feature representation of the target node. This mechanism can help graph attention networks to better capture the local and global relationships between nodes, while also preserving the feature information of the previous layer network. The convolutional pooling layer consists of one-dimensional convolutional layers and max-pooling layers. The role of the Convolutional Pooling Network is to directly learn features related to graph-level tasks without node sorting. It works on every graph data with predefined node ordering and connectivity. Therefore, this paper uses the Jump Graph Attention Network and Convolutional Pooling Network when extracting features. This design solves the problem of over-smoothing and the difficulty of separating vulnerable samples and neutral samples. In addition, We adopt a resampling method to ensure the balance of training data and use the balanced data to train a random forest classifier in the classification detection stage (Breiman, 2001). The key innovation of this paper is to combine the Jump Graph Attention Network with convolutional pooling layer as a graph feature extraction model.

In summary, the contributions of this paper in C\C++ language static code vulnerability detection are as follows:

- Effective class separation. We proposes a function-level static code vulnerability detection method SDV. In the feature extraction stage, we propose Jump Graph Attention Network layers and convolutional pooling layers. This method can effectively separate the features of the vulnerability and the neutral code sample.
- Extensive empirical evaluation. We complete the comparison with advanced methods under multiple data sets, and we analyze the reasons why SDV is superior to other methods through silhouette and feature dimensionality reduction visualization.
- Open comparison. We made the code and data available at https://github.com/cherishdd/SDV for certified comparisons.

The rest of this paper is organized as follows. Section 2 describes the work related to vulnerability detection. Section 3 details the framework of the SDV method. Section 4 provides the relevant experimental setup and description. Section 5 is the experimental results and analysis. Section 6 discusses the validity threats and limitations of SDV. Section 7 is the conclusion of this paper.

2. Related work

With the success of machine learning and deep learning in various fields, many researchers have proposed combining deep learning with vulnerability detection for better vulnerability code analysis. This paper divides vulnerability detection methods into two types: methods based on text sequences and methods based on graph structures.

2.1. Vulnerability detection methods based on text sequences

Vulnerability detection methods based on text sequences usually slice the source code according to lexical semantics and search for identifiers, keywords, function names, and operators in the marked code according to the lexical rules in program language analysis. Then, these key characters are converted into Token sequences in the order of natural sentences, embedded in the feature space using word vectors, and then detected by natural language processing.

Starting from function calls, VulDeePecker (Li et al., 2018) converts the code into an intermediate representation with data and control dependency information, the code gadget represents a set of semantically interconnected code sentences and finally selects the Bi-LSTM model to extract feature vectors. μVulDeePecker (Zou et al., 2019) introduced code attention based on Vul- DeePecker, which can focus on the local features of the program and then fuse the global characteristics of the vulnerability through the BiLSTM network. SySeVR (Li et al., 2021c) automatically extracts syntactic and semantic fragments based on the control flow graph, data dependency graph, and program dependency graph. Next, it uses the vector features pre-trained by the code snippet as the input of the bidirectional gated neural network to represent the vulnerability features. However, slicing will cause some information in the source program to be lost, and semantic information will be separated between codes. The research of Lin et al. treats the code as a sequence of text and employs BiLSTM in the feature extraction process. In their study, the multi-domain theory is used to increase the acquisition of vulnerability feature information (Lin et al., 2019, 2018). Wei et al. proposed a context-aware vulnerability detection method, which can capture the contextual dependencies related to source–sink patterns (Wei et al., 2021).

The vulnerability detection method based on sequences does not consider the structural information of the code logically, while the graph structure is the most effective data structure to express the syntax and semantic knowledge of the code. In addition, the above method based on slicing processing will cause the loss of code information and the logical relationship between splitting code statements. It is necessary to retain complete code information to discover vulnerability pattern features.

2.2. Vulnerability detection methods based on graph structures

In general, to obtain the source code's syntactic and semantic structure, the code is first processed into an abstract syntax tree, a data flow graph, a program dependency graph, or a code property graph. Then, the node information is initialized by the word embedding method (Word2vec (Mikolov et al., 2013), Glove (Pennington et al., 2014), or Doc2vec (Le and Mikolov, 2014)), and the initialized node features are employed as the input of the Graph Neural Networks(hereafter, GNN) for graph feature extraction and vulnerability detection.

Devign (Zhou et al., 2019), BGNN4VD (Cao et al., 2021), Reveal (Chakraborty et al., 2021), Poem (Ye et al., 2020), and FUNDED (Wang et al., 2020) construct comprehensive property graphs based on multilateral type relationships, covering data and control dependencies and function calls. They utilize GGNN in the feature extraction stage, and these methods consider multiple edge relationships based on abstract syntax trees. However. Poem incorporates the control data flow graph of the underlying middleware and takes into account structured information at different levels. Devign designs a convolutional layer to aggregate raw information and improves vulnerability classification performance. FUNDED utilizes layer-by-layer high-speed gating units to control the propagation of noisy data, thereby extracting more useful code features. BGNN4VD utilizes a bidirectional gated graph neural network to reverse code information aggregation. Reveal adds sampling methods to balance the dataset and uses representation learning to separate sample categories. In the code property graph, adjacent nodes have crucial relationships, while various neighbor nodes have varying degrees of relevance. GGNN cannot extract the importance of neighbor nodes. In addition, the feature extraction of node information by GGNN with sequential structure will have an over-smoothing problem; that is, the final graph feature vector will be densely overlapped, so the classification performance is not optimal.

Attention-based graph neural network methods are also widely used in vulnerability detection (Zhang et al., 2022; Hin et al., 2022; Li et al., 2021a; Zheng et al., 2021a; Nguyen et al., 2022). Based on the slice property graph, VulSPG introduces a triple attention mechanism of node attention mechanism, statement attention mechanism, and subgraph attention mechanism to improve the aggregation ability of node information. But slicing will cause the loss of code information (Zheng et al., 2021a). ACGVD designs a node-level attention mechanism and a path-level attention mechanism for data control flow graphs to improve classification performance (Li et al., 2021a). GSM (Zhang et al., 2022) is a vulnerability code detection method that leverages Graph Attention Network for graph feature extraction and uses Metric Learning to mine separation vulnerabilities. Although these methods utilize the attention mechanism to consider the importance of neighbor nodes, there is an over-smoothing problem in the feature extraction stage. Therefore, the vulnerability detection performance is poor.

It is worth noting that, unlike the above two methods, these methods consider a value flow path of the vulnerability, trying to understand the potential vulnerability path, this is a more fine-grained approach (Cheng et al., 2022a,b). For example, on the one hand, ContraFlow compiles the source code into LLVM IR, and then inputs it into SVF (Sui and Xue) to parse the value-flow graph; on the other hand, it uses Joern (Anon, 0000a) to extract the AST subtree. ContraFlow trains a value-flow embedding layer model and Value-flow Path Encoder (VPE) by contrastive learning. Finally, ContraFlow uses the attention mechanism layer and linear layer for classification (Cheng et al., 2022b). Flow2Vec builds inter-procedural value flow graphs on IR, which can precisely preserve inter-procedural dependencies. Through the asymmetric transitivity of value flows, Flow2Vec embeds a program's control flow and alias-aware value flows into a low-dimensional vector space. Flow2Vec (Sui et al., 2020) maintains context-sensitive transitivity by filtering out infeasible value flow paths via CFL reachability. This is a more precise inter-procedural code embedding method. This may replace Word- 2vec used in this paper to increase vulnerability detection performance. VGDETECTOR (Cheng et al., 2019) processes the source code into a control flow graph. It considers the control flow in the code and embeds the nodes through Doc2vec. Finally, VGDETECTOR utilizes graph convolutional networks to preserve high-level control flow information of vulnerable programs. However, this paper argues that multi-layer graph convolutional neural networks can also cause over-smoothing problems, which is the problem that this paper aims to solve.

3. SDV methodology

The code property graph integrates all syntactic and semantic information of a piece of code (Yamaguchi et al., 2014). We design SDV, which learns the feature of code property graphs via the Jump Graph Attention Network and the convolutional pooling layer. The SVD method's framework is depicted in Fig. 1, encompassing four stages. Phase I: Node Feature Initialization. This stage transforms a function into a code property graph that possesses comprehensive program semantics and initializes the coding of every node in the graph. Phase II: Graph Feature Extraction. During this stage, the Graph Attention Network is employed to continually aggregate and assimilate the information of neighboring nodes to represent node features. Subsequently, the Convolutional Pooling layer is implemented to further learn the high-dimensional features of the code property graph. The Graph Attention Network layer is the jump structure, which can retain the input information of each layer of nodes. Phase III: Sampling. SDV decouples the graph feature extraction from the classifier and adds a sampling method in the middle to ensure the balance of the training dataset. Phase IV: Vulnerability Detection. In this stage, we utilize the balanced dataset to train a Random Forest classifier and then conduct a function prediction classification. Specifically, 0 denotes neutral code, while 1 represents vulnerable code.

3.1. Formalization of code vulnerability detection

This paper analyzes code vulnerabilities at the functional level, which is a finer-grained detection method than the file. In this

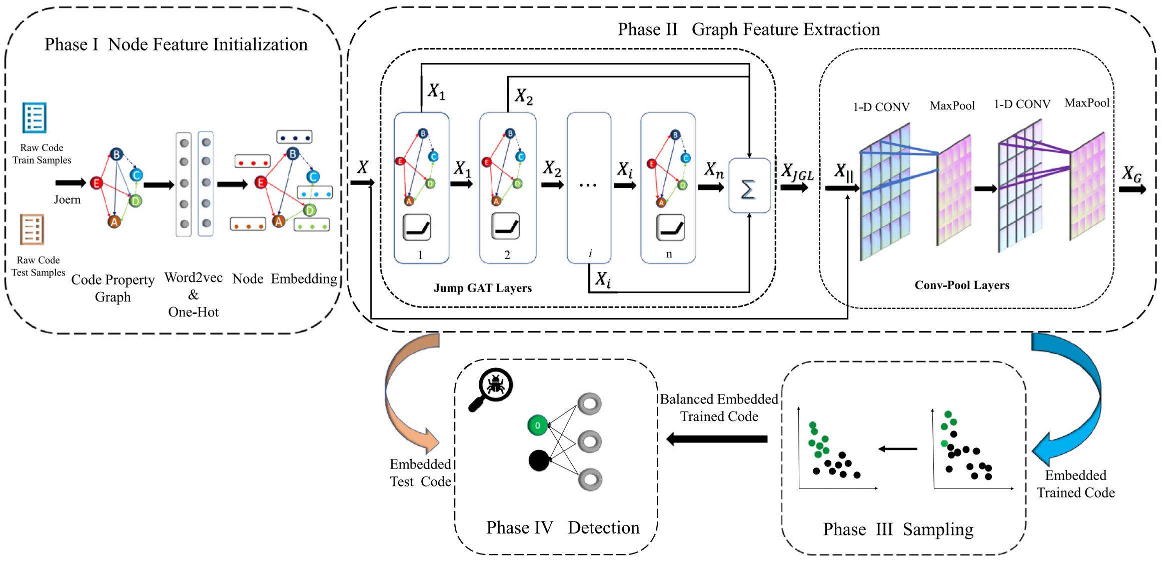


Fig. 1. The Framework of the SDV method.

paper, the problem of code vulnerability detection is regarded as a binary classification problem. Given a code function, it predicts whether it may contains a vulnerable code. We set the code training data to be denoted as $(\mathcal{C}_i, \mathcal{Y}_i)$, where $\mathcal{C}_i$ represents the source code and $\mathcal{Y}_i$ represents the source code label, that is, vulnerability or neutral. This paper decouples the feature extraction stage from the classifier model. Therefore, the goal of the feature extraction stage is to design a vulnerability detection model $\mathcal{F}$ to extract vulnerable code features most efficiently, $\mathcal{F}$ needs to reduce the value of the loss function through continuous iterative learning. Therefore, the objective function of the feature extraction stage can be defined as:

$$min \sum_{i=1}^{n} \mathcal{L}(\mathcal{F}(\mathcal{C}_i, \mathcal{Y}_i)) \tag{1}$$

Where $\mathcal{L}$ represents the loss function.

3.2. Node feature initialization

Vulnerability detection via code property graphs (Yamaguchi et al., 2014) has been proven to be one of the most effective methods (Chakraborty et al., 2021; Zhou et al., 2019; Zhang et al., 2022; Cheng et al., 2021). The code property graph contains various relationships between statements, such as data flow, control flow, definition, use, and dominate. In addition, this paper analyzes a single function, and there is no inter-procedural program analysis. Hence, we do not use the sparse value flow graph (Sui and Xue).

Firstly, we use Joern to parse the source code $\mathcal{C}_i$ into a code property graph $\mathcal{G}(V, E)$, it generates node.csv and edge.csv. Node.csv saves the information of V, which contains node ID (1, 2, 3, etc.), node attributes (ComoundStatement, Identifier CallExpression, ArgumentList, etc.) and Code Statement. Code statements represent entire statements or subsets of statements. Edge.csv saves the information of E, which contain the edge's start and end nodes and the edge's type (REACHES, DOM, DEF, CONTROLS, etc.). Fig. 2 is an example of a buffer overflow vulnerability code, which shows all the property graph's information of the vulnerability code. To prevent the loss of the relationship and semantic information between nodes, we use eight types to connect the nodes in the code property graph instead of fixed ones. Although there will be redundancy, it may become the key information for mining vulnerability features.

Secondly, SDV standardizes the naming of functions or variables, that is, the function names are standardized as FUN0, FUN1, etc., and the variable names are standardized as VAR0, VAR1, etc. For each code property graph, we number variable names or function names starting from 0. After standardization, codes of the same structure may have the same FUN or VAR in similar positions. Standardized preprocessing not only reduces noise, but also prevents vocabulary explosion. Standardized naming does not change the semantic structure of the code (Chakraborty et al., 2021).

Finally, we start to encode node *i*. Node information includes node attributes and code statements or code fragments. The initialization coding rules are as follows, (1) There are 71 kinds of node attributes, which are encoded by One-hot (Mitra et al., 1997) and expressed as $\mathcal{P}_i$, (2) There is a code statement in each node, as shown in Fig. 2, Therefore, Word2vec is used to encode each code statement, which is represented as $\mathcal{S}_i$ and its vector length is set to 100. Therefore, the feature vector $\mathcal{X}_i$ of node *i* is composed of two parts and expressed as:

$$\mathcal{X}_i = \mathcal{P}_i \parallel \mathcal{S}_i \tag{2}$$

Where $\parallel$ represents connection. The feature vector $\mathbb{X}$ of all nodes is defined as:

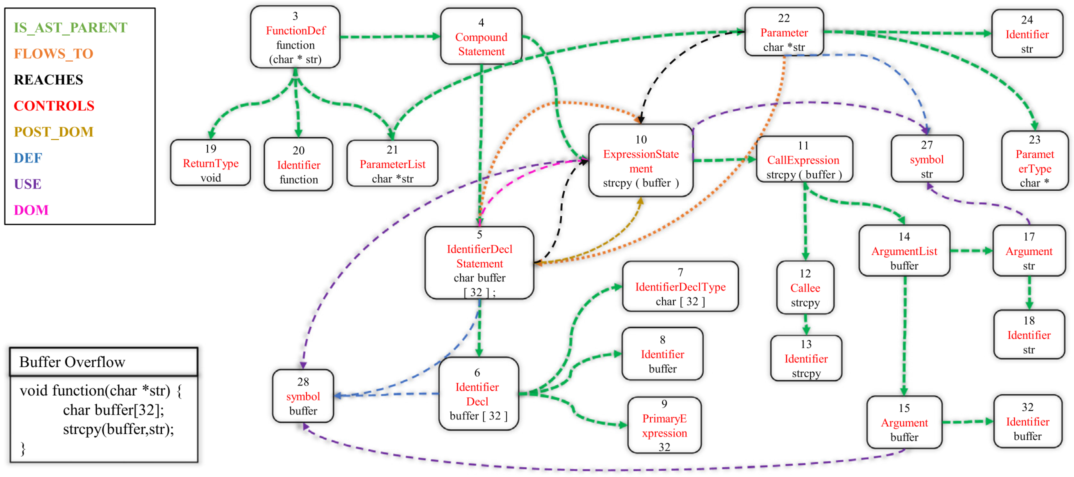
$$\mathbb{X} = (\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_i, \ldots, \mathcal{X}_n) \tag{3}$$

3.3. Graph feature extraction

This stage is divided into two steps. The first step is to input the initialized node vector $\mathbb{X}$ into the Graph Neural Network module, which uses the Jump Graph Attention Network to aggregate and update the node features in the entire graph to obtain the feature vector of each node. The second step is to input the aggregated node feature vector into the convolutional pooling layers for class feature distinction.

3.3.1. Jump graph attention network layer(JGL)

Over-smoothing means that in a multi-layer graph neural network, as the number of layers increases, the characteristics of nodes will become too similar. This loses the personalized information in the original graph structure, which reduces the representation ability and classification performance of the model. To solve the over-smoothing problem of graph neural networks,



```
void function(char *str) {
        char buffer[32];
        strcpy(buffer,str);
}
```

Fig. 2. An example of code property graph for buffer overflow vulnerability.

inspired by Jump Knowledge Network (Xu et al., 2018), we design the Jump Graph Attention Network layer(hereafter, JGL) and use the Relu function to activate node features after each layer of GAT. As shown in Phase II Jump GAT Layers of Fig. 1. The jump structure aggregates the node features of different layers in the multi-layer GNN. This process enables the nodes to obtain the information of local neighbors in the next layer and the global structure information of the longer hops. Its purpose is to integrate node feature information at different levels to maintain the validity of the model. Therefore, the jump structure maintains more original graph structure information and avoids the over-smoothing problem.

Each GAT layer inputs the node vector and the code property graph. There are multiple attention mechanisms α between each node and its neighbors, and α is calculated by a single-layer feed-forward neural network. After each layer of GAT, all nodes will aggregate the information of their neighbor nodes to update according to the attention mechanism. The general steps of GAT for node updating using a multi-head attention mechanism are shown below (Zhang et al., 2022).

The first step uses a linear transformation matrix to map the input features of the node to $\mathbb{R}$.

$$Z_i = W\mathcal{X}_i \tag{4}$$

Where W is the linear transformation matrix and $\mathcal{X}_i$ is the node input feature. The second step uses *LeakyReLU* to provide nonlinear mapping, and calculates the attention score e_{ij} between nodes.

$$e_{ij} = LeakyReLU(\vec{a}^T(Z_i \parallel Z_j)) \tag{5}$$

Where $\vec{a}$ is the learnable weight vector. The third step utilizes the softmax function for normalization to obtain the attention weight α_{ij}.

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k\in\mathcal{N}(i)} exp(e_{ij})} \tag{6}$$

Where $\mathcal{N}(i)$ represents the set of neighbor of node i.

If the GAT layer is in the first or the middle, the node features are updated for the splicing operation.

$$\mathcal{X}_i = \Vert_{k=1}^{K} \Big(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^k W^k \mathcal{X}_j\Big) \tag{7}$$

If the GAT layer is in the last, the node feature update operation is averaging.

$$\mathcal{X}_i = \frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^k W^k \mathcal{X}_j \tag{8}$$

Suppose that the node features after the i-th layer of GAT layer are represented as $\mathbb{X}_i$.

$$\mathbb{X}_i = Relu(GAT(\mathbb{X}_{i-1})) \tag{9}$$

The JGL adds each feature vector $\mathbb{X}_i$. Therefore, the node feature $\mathbb{X}_{JGL}$ obtained after the JGL is expressed as:

$$\mathbb{X}_{JGL} = \mathbb{X}_1 + \mathbb{X}_2 + \cdots + \mathbb{X}_i + \cdots + \mathbb{X}_n \tag{10}$$

3.3.2. *Convolutional pooling layer (CPL)*

After the graph neural network, the features of different categories will be in a state of overlap. Fig. 3(a) shows the spatial distribution of sample features after using GGNN, and Fig. 3(b) shows the spatial distribution of sample features after using GAT. SDV adds two Convolutional Pooling Layers(hereafter, CPL) to extract node feature vectors related to the current graph-level task. CPL contains one-dimensional convolutional layers and max-pooling layers. Each code property graph has predefined node ordering and connection information between nodes, which is encoded in an adjacency matrix. The role of convolutional pooling layers is to directly learn features relevant to graph-level tasks without node ranking. This approach is suitable for cases where each code property graph has a predefined node ordering and connection. With convolutional pooling layers, we can more efficiently learn graph data with predefined node orderings and improve classification performance on graph-level tasks. That is, we utilize convolutional pooling layers to increase the separation of vulnerable and neutral samples.

As shown in Phase II Convolutional Pooling Layers of Fig. 1. CPL splices the original node feature $\mathbb{X}_i$ and the JGL node feature $\mathbb{X}_{JGL}$. The spliced feature $\mathbb{X}_\parallel$ is expressed as:

$$\mathbb{X}_\parallel = \mathbb{X} \parallel \mathbb{X}_{JGL} \tag{11}$$

Next, SDV inputs $\mathbb{X}_\parallel$ to the one-dimensional convolutional layer and max-pooling layer to obtain high-dimensional features of nodes. CPL performs local perception through a one-dimensional convolution layer and aggregates local features at a higher level to obtain global information. The max-pooling layer can compress data and parameters. It also can reduce overfitting and improve the fault tolerance of the model.

Two layers of CPL are used in SDV. After the first layer of CPL, the node characteristics are updated to $\mathbb{X}_{CPL1}$:

$$\mathbb{X}_{CPL1} = MaxPool(Relu(1\_D\_Conv(\mathbb{X}_\parallel))) \tag{12}$$

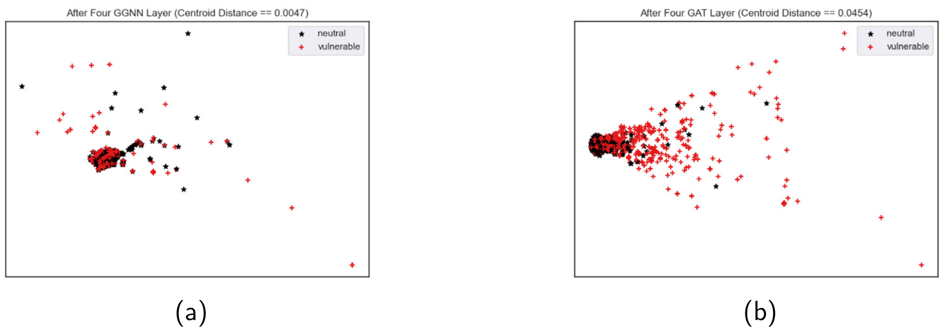


Fig. 3. (a) the spatial distribution of sample features after using GGNN (b) the spatial distribution of sample features after using GAT.

After the second layer of CPL, the node characteristics are updated to $\mathbb{X}_{CPL2}$:

$$\mathbb{X}_{CPL2} = MaxPool(Relu(1\_D\_Conv(\mathbb{X}_{CPL1}))) \tag{13}$$

Where $1\_D\_Conv(\cdot)$ represents a one-dimensional Convolutional layer.

SDV aggregates the vectors of all nodes of the code property graph through a summation function to form a graph feature vector $\mathbb{X}_G$, $\mathbb{X}_G$ is expressed as follows:

$$\mathbb{X}_G = \sum_{v \in V} \mathbb{X}_{CPL2} \tag{14}$$

3.4. Sampling

The proportion of vulnerable and benign codes in the real world is unbalanced. Therefore, there is also an imbalance in the real training dataset. We decoupled feature extraction and classification training, so this paper resamples the extracted dataset features in this stage. The sampling technique uses a combination of over-sampling and under-sampling, SMOTETomek (Zeng et al., 2016). Compared with a single sampling method, SMOTE-Tomek has been proven feasible and effective in vulnerability detection (Zhang et al., 2022).

SMOTETomek is a combination of the over-sampling method Synthetic Minority Oversampling Technique (hereafter, SMOTE) (Chawla et al., 2002) and the under-sampling method Tomek-Links (Tomek, 1976). First, we synthesize new vulnerability class samples using the SMOTE method. SMOTE generates new samples by interpolating between vulnerable samples. Specifically, it selects a fragile sample, then, randomly selects a sample among its nearest k vulnerable class samples, and finally, generates a new vulnerable sample by interpolating between the two samples. The SMOTE method is repeated until the proportion of vulnerable samples and neutral samples is balanced. The SMOTE method can increase the number of vulnerable sample types and alleviate the imbalance of the dataset. Second, we utilize the TomekLinks method to clean up noisy samples. TomekLinks refer to pairs of samples that belong to different classes in a dataset but are very close in distance. These sample pairs may cause the decision boundary of the classifier to be unstable. Therefore, the TomekLinks method mines and removes these sample pairs to reduce the influence of noisy samples. After many times with the TomekLinks method, we end up with a balanced dataset. We implement the sampling method using the sklearn library. The sampling module in SDV is configurable and can be replaced by other balancing methods (eg, MWMOTE Barua et al., 2012, ProWSyn Barua et al., 2013, etc.).

3.5. Detection

After the feature extraction stage, the vulnerability and neutral samples have been basically separated. Furthermore, one of the drawbacks of resampling is overfitting. Therefore, we can choose any regular classifier. Random forest (Breiman, 2001) controls the depth and the size of the Gini index to prevent overfitting. In addition, Random Forest classifiers are often used in natural language processing-based vulnerability detection, and their effectiveness has been proven (Lin et al., 2019, 2018). Therefore, this paper chooses the Random Forest classifier. SDV uses the balanced training data features as input to the Random Forest to train the classifier. Finally, the test dataset features are input into the trained classifier to detect vulnerable and neutral codes.

4. Experimental setup

4.1. Experimental implement

Experiments use a CPU server with Intel(R) Xeon(R) CPU E5-24300 @ 2.20 GHz and 128 GB of RAM. These library functions are mainly used in the experiment: version 3.7 of the python programming language, version 0.3.1 of the C language code parsing tool Joern, version 1.9.0 of the deep learning framework Pytorch, version 0.5.1 of the graph neural network framework DGL and version 0.19.1 Machine learning framework sklearn. For the Jump GAT, the maximum epoch is set to be 1000. We use AutoML (Anon, 0000b) to select parameters from batch size (64, 128, 256), and learning rate (0.001, 0.0001). The hyperparameter settings involved in the network layer of SDV are shown in Table 1.

4.2. Dataset

Recent research suggests that vulnerable code detection models should be evaluated on data that can represent the characteristics of real-world vulnerabilities (Chakraborty et al., 2021). This means that the evaluation requires a sufficient number of code functions extracted from real projects. To ensure the effective testing of each method on the dataset, this paper selects four function-level datasets, as shown in Table 2. The chrdeb and ffmqem will be used for model training. The github and sard datasets will be used for real project testing. In the classification stage, the code under the same data set is divided into training data set and test data set according to 8:2, and it is ensured that both the training set and the test set contain vulnerability samples. The vulnerable and neutral functions and the total number of samples in the training dataset and test are shown in Table 3.

Table 1
Hyperparameter settings for SDV.

Network	Parameter	Value
One-hot	Vector size	71
Word2vec	Window size	10
	Vector size	100
GAT	(Input,Output)-1	(171,100)
	(Input,Output)-2\3	(100,100)
	Learning Rate	0.001
	Number layers	3
	Head_nums	2
	Batch_size	128
	Activation	None
1-D Conv	(Input,Output)	(371,200)
	Outputdim	200
	Kernel-Conv1	3
	Kernel-Conv2	1
	Number layers	2
	Activation	Relu
MaxPool	Kernel-Pool1	3
	Kernel-Pool2	2
	Stride	2
	Number layers	2
Optimier	Adam	
Loss function	Cross Entropy Loss	

Table 2
Dataset description.

Dataset	Vulnerable	Neutral	Source
chrdeb	1658(9.1%)	16508	Reveal (Chakraborty et al., 2021)
ffmqem	11641(45.6%)	13847	Devign (Zhou et al., 2019)
sard	5401(26.4%)	14999	FUNDED (Wang et al., 2020)
github	2408(9.1%)	5490	FUNDED (Wang et al., 2020)

Table 3
Number of samples of training and test dataset.

Dataset	Training		Testing	
	Vulnerable	Neutral	Vulnerable	Neutral
chrdeb	1328	13204	330	3304
= ffmqem	9279	11112	2362	2735

4.3. Evaluation metrics

Static code vulnerability detection is a two-category problem. To prevent the singleness of the evaluation indicators, the two-category evaluation indicators selected are Precision, Recall, F1-Score, and MCC. MCC comprehensively considers TP, TN, FP, and FN, it is a more comprehensive evaluation metric, which can also be used in the case of unbalanced samples. The calculation formula of each indicator is as follows:

$$Precision = \frac{TP}{TP + FP} \tag{15}$$

$$Recall = \frac{TP}{TP + FP} \tag{16}$$

$$F1\text{-}Score = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} \tag{17}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{18}$$

Where TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative.

Moreover, we use t-SNE (der Maaten and Hinton, 2008) to reduce dimensionality to visualize features and use Centroid Distance (Euclidean Distance between two class centers) to quantify separability. We use silhouette (Hove, 1986) to visualize feature vectors and use the silhouette coefficient to quantify the effectiveness of classification. Where the silhouette coefficient is a measure of the similarity of an object to other classes.

4.4. Research question

RQ1: Compared with the function-level methods, how does SDV perform on static code vulnerability detection? **(Advanced Verification)**

RQ2: How does each part of SDV affect the performance of vulnerable code detection? **(Ablation Experiment)**

RQ3:How effective is the detection of SDV in real projects? **(General Verification)**

4.5. Experimental method

RQ1 Advanced Verification

Goal: Verify the performance of SDV and other methods on different indicators, and analyze the reasons for the performance improvement.

Baseline: We compare SDV with the following baselines, VulDeePecker (Li et al., 2018), SySeVR (Li et al., 2021c), which text-based vulnerability detection model, and Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021), which graph-based vulnerability detection model.

Dataset: chrdeb and ffmqem.

Evaluation metric: Precision, Recall, F1-Score, and Centroid Distance.

RQ2 Ablation Experiment

Goal: Analyze the impact of each stage on the performance of vulnerability detection.

Baseline: SDV (W/O JGL) represents that JGL (Jump GAT Layers) is not used in the graph feature extraction stage. SDV (W/O CPL) represents that CPL (Convolutional Pooling Layer) is not used in the graph feature extraction stage. SDV (W/O Sampling) represents that the sampling stage does not use the rebalancing technique SMOTETomek (Zeng et al., 2016).

Dataset: ffmqem.

Evaluation metric: Precision, Recall, F1-Score, and MCC.

RQ3 General Verification

Goal: Verify the performance of SDV in real-world projects.

Baseline: We train on the ffmqem dataset to obtain the optimal models of SDV, Devign, and Reveal. Then we put the github and sard datasets into the model for detection.

Dataset: github, sard, and ffmqem.

Evaluation metric: TP, Precision, Recall, F1-Score, and silhouette coefficient.

5. Results and analysis

5.1. Advanced verification

Numerical analysis. Fig. 4 shows the comparison between SDV and other methods under Precision, Recall, and F1-Score. The performance description of their indicators is as follows:

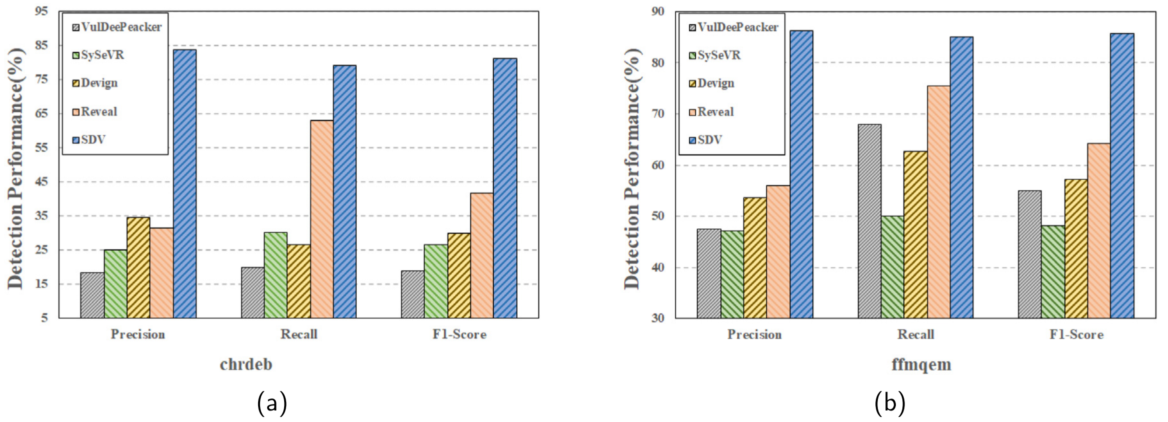


Fig. 4. Precision, Recall, and F1-Score when VulDeePeacker, SySeVR, Devign, Reveal, and SDV detect vulnerabilities under chrdeb and ffmqem.

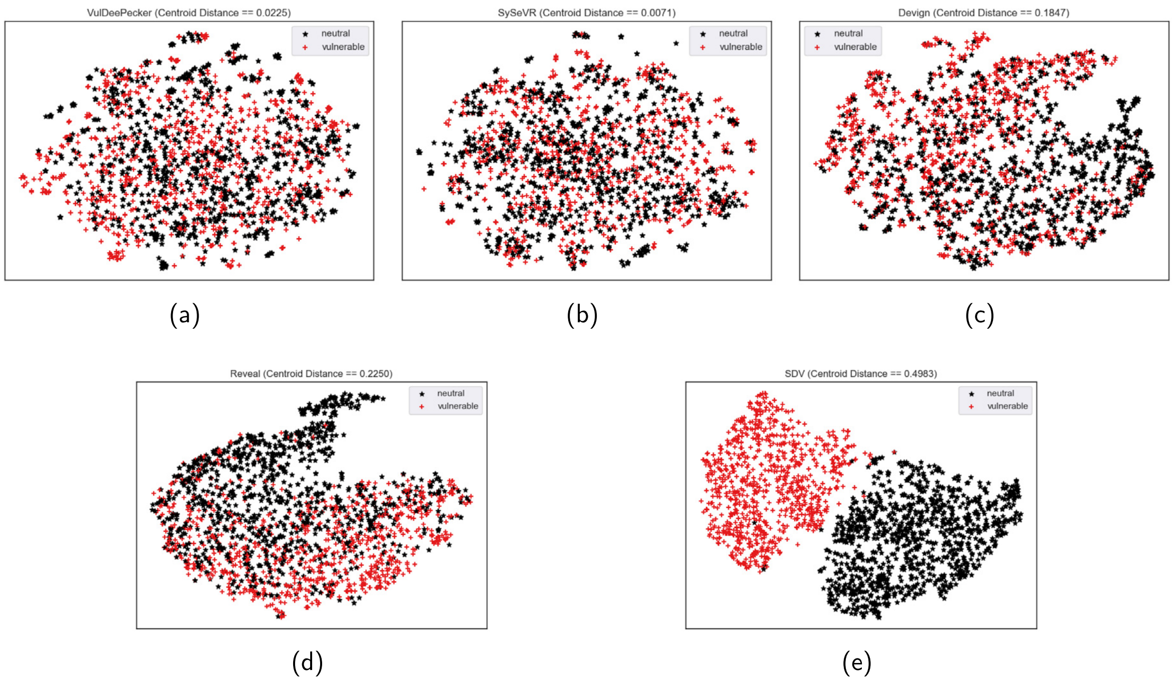


Fig. 5. t-SNE figure, analyzing the separation between vulnerable codes (+) and neutral codes (*).

- chrdeb: SDV reached 83.6%, 79.0%, and 81.2% in Precision, Recall, and F1-Score respectively. The F1-Score of VulDeePeacker (Li et al., 2018), SySeVR (Li et al., 2021c), Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021) is only 18.9%, 26.6%, 29.9%, and 41.7%. The number of True Positives (TP) is smaller than that of SDV.
- ffmqem: SDV has reached more than 85% in Precision, Recall, and F1-Score. Compared with the other four methods, the F1-Score has increased by 30.6%, 37.5%, 28.4%, and 21.3%, respectively.

t-SNE dimensionality reduction visualization. Taking the ffmqem dataset as an example, we use t-SNE dimensionality reduction to visualize the feature distribution of samples under VulDeePeacker, SySeVR, Devign, Reveal, and SDV, as shown in Fig. 5(a)(b)(c)(d)(e), where **red "+"** indicates the vulnerability sample, and the **black "*"** indicates a neutral sample. It can be seen from Fig. 5 that the two types of codes in VulDeePeacker, SySeVR, Devign, and Reveal have a large overlap. The vulnerability code and the neutral code are almost inseparable. However, the sample class separation of SDV is obvious, and the maximum Centroid Distance of SDV is 0.4983, which is about 0.27 and 0.31 higher than Reveal and Devign. Especially, VulDeePeacker and SySeVR cannot reach 0.05. The poor separation also explains the poor performance of the baseline model in vulnerability detection.

Answer To RQ1: Under different datasets, SDV has shown better performance than advanced vulnerability detection methods in multiple indicators.

5.2. Ablation experiment

Numerical analysis. Fig. 6 shows the vulnerability detection performance results of JGL with different layers in the chrdeb dataset. It can be seen from the figure that the best performance is achieved at the third layer, and tends to be stable at the fourth, sixth, and eighth layers. The F1-Score fluctuates around 81%.

Table 4 describes the ablation experiment results under the SDV (W/O JGL), SDV (W/O CPL), SDV (W/O Sampling), and SDV methods under the ffmqem dataset. It can be seen from Table 4

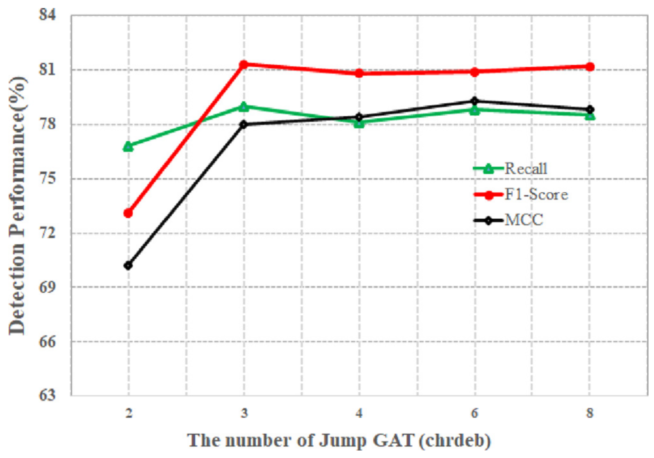


Fig. 6. Vulnerability detection performance of JGL with different layers.

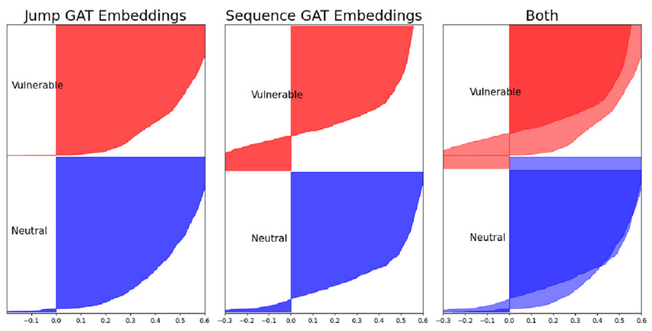


Fig. 7. The silhouette of Jump GAT and Sequence GAT.

that the lack of any module will have a great impact on the final vulnerability detection performance. Especially when the two modules of JGL (Jump GAT Layers) and CPL (Convolutional Pooling Layer) are missing, F1 is relatively reduced by 33.6% and 24.7%, and MCC is also as low as 11% and 12%. These data shows the importance of JGL and CPL modules for vulnerability feature extraction, because the role of JGL is to alleviate the over-smoothing problem, and the role of CPL is to separate vulnerability and neutral samples. Vulnerability samples in the ffmqem dataset accounted for 45.6%. After rebalancing technology, the F1-Score also increased by 6.4%. This illustrates the power of rebalancing techniques for vulnerable code detection methods.

Silhouette analysis. We use the silhouette (Hove, 1986) to show the feature embeddings of Jump GAT and Sequence GAT, as shown in Fig. 7. Specifically, the silhouette coefficient score of Jump GAT is 0.475, and the silhouette coefficient of Sequence GAT is 0.316. These data show that the jump structure has better feature representation than the sequential structure. The jump structure can extract more node feature information.

Answer To RQ2: JGL, CPL, and Sampling play a vital role in the detection performance of generational vulnerability codes. If they are missing, it will cause low performance.

5.3. General verification

Numerical analysis. We trained the best models of Devign, Reveal, and SDV on the ffmqem sample. We tested 28298 codes, of which 7809 were vulnerable. Table 5 shows the vulnerability detection results in real projects. SDV detected a total of 2683 vulnerabilities, while Devign and Reveal only detected 1117 and

Table 4
SDV ablation experiment results.

Method	Pre.	Rec.	F1.	MCC
SDV(W/O JGL)	53.7	50.3	52.0	0.11
SDV(W/O CPL)	50.8	75.9	60.9	0.12
SDV(W/O Sampling)	77.2	81.4	79.2	0.61
SDV	**86.2**	**85.1**	**85.6**	**0.86**

Table 5
SDV ablation experiment results.

Dataset	Method	TP	Pre.	Rec.	F1.
sard	Devign	969	**32.9**	17.9	23.2
sard	Reveal	185	31.7	3.4	6.2
sard	SDV	**1983**	26.0	**36.7**	**30.4**
github	Devign	148	27.5	6.14	10.1
github	Reveal	261	**33.9**	10.8	16.4
github	SDV	**655**	27.8	**27.2**	**27.5**

446 vulnerabilities, respectively. Reveal shows extremely poor performance in the sard dataset, while SDV performs well in real projects, which shows that SDV has better robustness among the three methods. In addition, the calculation parameter of SDV is 862023, the calculation parameter of Devign is 1603011, and the calculation parameter of Reveal is 3537801. Compared with them, SDV has lower calculation overhead.

Silhouette analysis. We use the silhouette to demonstrate the feature embeddings of SDV, Devign, and Reveal. From the "Both" overlap in Fig. 8, we can see that SDV has a wider embedding space, which corresponds to better class separation. Specifically, the silhouette coefficient of SDV is 0.475, the silhouette coefficient of Devign is 0.134, and the silhouette coefficient of Reveal is 0.093. The larger the silhouette coefficient, the higher the matching degree of the same class. Therefore, it shows that SDV has better class separation, and it shows better performance. However, Reveal shows worse performance on the sard dataset, which shows that it is also related to the applied dataset and the robustness of the model.

Finally, we show an example of a vulnerability detected by SDV but not by Devign and Reveal. The left side of Fig. 9 shows the vulnerability example. On line 17, if the value of **nr_data_stripes(map)** returns 1, then
[(i+rot)% map->num_stripes] is 1, it occurs out of bounds write. The right side of Fig. 9 shows part of the code property graph related to the vulnerability. It can be seen that node 87 (out-degree: 12, in-degree: 7) and node 12 (out-degree: 4, in-degree: 6) are very closely related to other nodes. If the information of neighbor nodes cannot be aggregated through the multi-head attention mechanism, this will lead to the lack of relevant semantic or structural relationships. Then, it will cause the model to fail to learn key sentence relationships.

Answer To RQ3: When performing vulnerability detection in real projects, SDV can detect 2683 vulnerabilities out of 7809. Compared with Reveal and Devign, SDV shows better generality and effectiveness.

6. Discussion

6.1. Threats to validity

External validity. One of the main external threats to this paper is the reliability of code data. As with much vulnerability detection research, the usefulness of this vulnerability detection

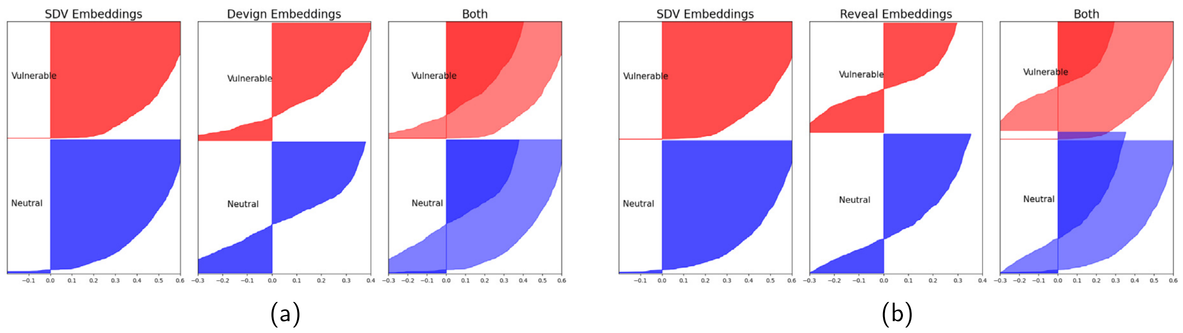
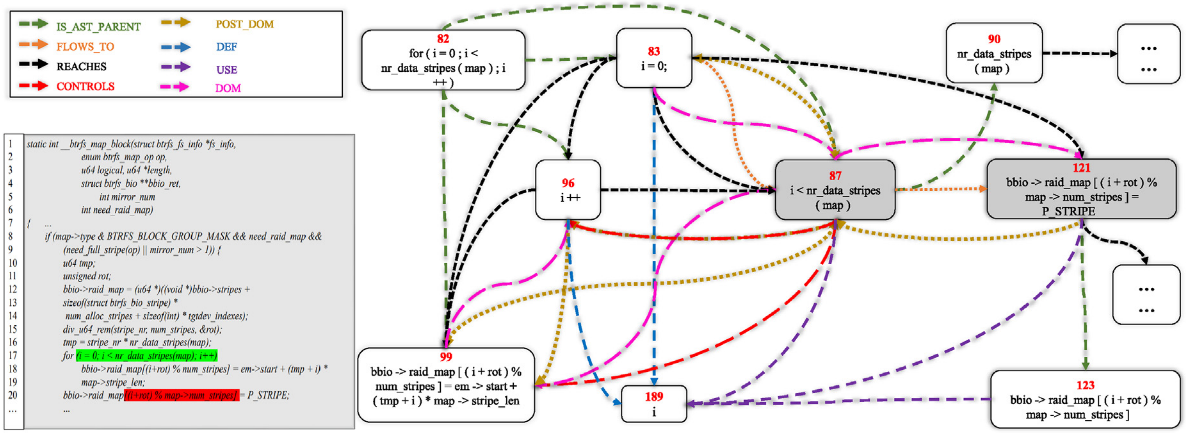


Fig. 8. (a)The silhouette of SDV and Devign. (b) The silhouette of SDV and Reveal.


IS_AST_PARENT
FLOWS_TO
REACHES
CONTROLS
POST_DOM
DEF
USE
DOM
82 for (i = 0 ; i < nr_data_stripes (map) ; i ++)
83 i = 0;
90 nr_data_stripes (map)
96 i++
87 i < nr_data_stripes (map)
121 bbio -> raid_map [(i + rot) % map -> num_stripes] = P_STRIPE
99 bbio -> raid_map [(i + rot) % num_stripes] = em -> start + (tmp + i) * map -> stripe_len
189 i
123 bbio -> raid_map [(i + rot) % map -> num_stripes]

```
static int __btrfs_map_block(struct btrfs_fs_info *fs_info,
            enum btrfs_map_op op,
            u64 logical, u64 *length,
            struct btrfs_bio **bbio_ret,
                int mirror_num
            int need_raid_map)
{     ...
    if (map->type & BTRFS_BLOCK_GROUP_MASK && need_raid_map &&
        (need_full_stripe(op) || mirror_num > 1)) {
        u64 tmp;
        unsigned rot;
        bbio->raid_map = (u64 *)((void *)bbio->stripes +
        sizeof(struct btrfs_bio_stripe) *
         num_alloc_stripes + sizeof(int) * tgtdev_indexes);
        div_u64_rem(stripe_nr, num_stripes, &rot);
        tmp = stripe_nr * nr_data_stripes(map);
        for (i = 0; i < nr_data_stripes(map); i++)
            bbio->raid_map[(i+rot) % num_stripes] = em->start + (tmp + i) *
            map->stripe_len;
        bbio->raid_map[(i+rot) % map->num_stripes] = P_STRIPE;
        ...
```

Fig. 9. An example of a vulnerability detected by SDV but not detected by Devign and Reveal.

tool depends on our use case. Our approach is specific to C\C++ languages and does not apply to projects in other programming languages. On the other hand, the accuracy of the labels can also affect the classification results. Although the labels in the dataset are all labeled by experienced experts, there will inevitably be mislabeled.

Internal validity. There are many uncertainties in the data processing stage and the model training stage. First, the data processing stage is not accurate enough to standardize the code, and there may still be noise codes. This paper is a static code analysis, which lacks information calls between processes, such as data race. Second, our model is not trained end-to-end, and SDV decouples the feature extraction stage from the classification stage, which seems to affect the experimental results. Because the middle of the process may increase uncertainty, such as data resampling. Finally, there is a threat to hyperparameter adjustment. Some parameters are even set directly by experience.

Conclusion validity. Evaluation metrics may threaten the results, but this paper selects 3–4 evaluation metrics for different research questions. These evaluation metrics are commonly used comprehensive indicators in vulnerability detection tasks, which reduce the randomness of evaluation results.

Construct validity. The implementation of the baseline methods threatens the results. Although we reproduce all baseline methods based on Reveal. For their datasets, they report high results, but they show low performance in the dataset of this paper. There are two main reasons for this result. On the one hand, the data processing methods of baseline methods are not accurate. On the other hand, the adaptation and adjustment of model parameters are difficult. There are still optimization problems.

6.2. Limitations

First, SDV is aimed at specific programming languages C and C++. Because SDV cannot build code property graphs in Java, Python, R, and PHP. Therefore, SDV cannot satisfy the vulnerability detection requirements of such languages.

Second, the Jump GAT layer and convolutional pooling layer can well distinguish different categories of features. Although t-SNE and silhouette have been used to explain the reasons for the higher performance of SDV, there is still a lack of reasonable interpretations. These interpretations are related to which nodes play an important role in the feature representation of the vulnerable. We will conduct more explanatory research on code embedding and code property graph explainability (Wan et al., 2022b). This will leverage the interpretability models of graph neural networks. For example, Ivdetect (Li et al., 2021b) has applied GNNExplainer (Ying et al., 2019) to explain the importance of vulnerability codes.

Finally, SDV is a function-level vulnerability code detection method, and this paper lacks the location of the vulnerability path. Coarse-grained reports are not accurate for vulnerability detection. Finding vulnerable program paths or bug trigger paths is crucial for static vulnerability detection (Cheng et al., 2022b). Therefore, we need to extract vulnerability trigger paths based on node relationships. We need node embedding and feature extraction according to the value flow path. During the evaluation, we can use the BTP (bug-triggering path) metric for quantitative evaluation (Cheng et al., 2022a).

7. Conclusion and future work

This paper analyzes the previous work and finds that the vulnerability detection method based on code fragments will lack semantic information, and the vulnerability code detection method based on sequential structure graph neural network has an over-smoothing problem. Therefore, this paper proposes a vulnerability detection method SDV that focuses on the separation of vulnerability and neutral codes. SDV decouples the feature extractor and the classifier. In the feature extraction stage, we use the Jump Graph Attention Network layer and the convolutional pooling layer. The jump structure can prevent over-smoothing problems, and the convolutional pooling layer can separate vulnerability samples and neutral ones. On real project datasets, compared with the most advanced graph-based vulnerability detection methods, SDV can detect 2.4 times more vulnerabilities than Devign and 6 times more than Reveal. In the future, we will focus on more fine-grained statement-level or path-level vulnerability detection. We will introduce vulnerability statement location, path location, and the interpretability of graph neural networks into the research work of vulnerability detection.

CRediT authorship contribution statement

Chunyong Zhang: Completed the experimental content, Manuscript of the paper. **Yang Xin:** Provided the paper ideas, Designed experiments, Checked grammar.

Declaration of competing interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work, there is no professional or other personal interest of any nature or kind in any product, service and/or company that could be construed as influencing the position presented in, or the review of, the manuscript.

Data availability

Data will be made available on request.

Acknowledgments

This paper was funded by the National Natural Science Foundation of China 62233003 and 2020YFB1708602.

References

Anon, 0000. https://github.com/joernio/joern.
Anon, 0000. https://github.com/automl/auto-pytorch.
Barua, Sukarna, Islam, Md.Monirul, Murase, Kazuyuki., 2013. Prowsyn: Proximity weighted synthetic oversampling technique for imbalanced data set learning. In: Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April (2013) 14-17, Proceedings, Part II 17. Springer, pp. 317–328.
Barua, Sukarna, Islam, Md Monirul, Yao, Xin, Murase, Kazuyuki, 2012. Mwmote–majority weighted minority oversampling technique for imbalanced data set learning. IEEE Trans. Knowl. Data Eng. 26 (2), 405–425.
Booth, Harold, Rike, Doug, Witte, Gregory A., et al., 2013. The national vulnerability database (nvd): overview.
Breiman, Leo, 2001. Random forests. Mach. Learn. 45 (1), 5–32.
Cao, Sicong, Sun, Xiaobing, Bo, Lili, Wei, Ying, Li, Bin, 2021. Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136, 106576.
Ceara, Dumitru, Potet, Marie-Laure, Ensimag, G.I, Mounier, Laurent, 2009. Detecting Software Vulnerabilities-Static Taint Analysis. Vérimag-Distributed and Complex System Group, Polytechnic University of Bucharest.
Chakraborty, Saikat, Krishna, Rahul, Ding, Yangruibo, Ray, Baishakhi, 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng..
Chawla, Nitesh V., Bowyer, Kevin W., Hall, Lawrence O., Kegelmeyer, W. Philip, 2002. Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.
Cheng, Xiao, Nie, Xu, Li, Ningke, Wang, Haoyu, Zheng, Zheng, Sui, Yulei, 2022a. How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors. IEEE Trans. Dependable Secure Comput..
Cheng, Xiao, Wang, Haoyu, Hua, Jiayi, Xu, Guoai, Sui, Yulei, 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–33.
Cheng, Xiao, Wang, Haoyu, Hua, Jiayi, Zhang, Miao, Xu, Guoai, Yi, Li, Sui, Yulei, 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In: 2019 24th International Conference on Engineering of Complex Computer Systems. (ICECCS), IEEE, pp. 41–50.
Cheng, Xiao, Zhang, Guanqin, Wang, Haoyu, Sui, Yulei, 2022b. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 519–531.
Dukes, LaShanda, Yuan, Xiaohong, Akowuah, Francis, 2013. A case study on web application security testing with tools and manual testing. In: 2013 Proceedings of IEEE Southeastcon. IEEE, pp. 1–6.
Ganesh, Vijay, Leek, Tim, Rinard, Martin, 2009. Taint-based directed whitebox fuzzing. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 474–484.
Gao, Debin, Reiter, Michael K., Song, Dawn, 2008. Binhunt: Automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security. Springer, pp. 238–255.
Hin, David, Kan, Andrey, Chen, Huaming, Ali Babar, M., 2022. Linevd: statement-level vulnerability detection using graph neural networks. arXiv preprint arXiv:2203.05181.
Hove, Patrick L. Van, 1986. Silhouette-Slice Theorems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE RESEARCH LAB OF ELECTRONICS.
King, James C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.
Le, Quoc, Mikolov, Tomas, 2014. Distributed representations of sentences and documents. In: International conference on machine learning. PMLR, pp. 1188–1196.
Li, Min, Li, Chunfang, Li, Shuailou, Wu, Yanna, Zhang, Boyang, Wen, Yu, 2021a. Acgvd: Vulnerability detection based on comprehensive graph via graph neural network with attention. In: International Conference on Information and Communications Security. Springer, pp. 243–259.
Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, Zemel, Richard, 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.
Li, Yi, Wang, Shaohua, Nguyen, Tien N., 2021b. Vulnerability Detection with Fine-Grained Interpretations. Association for Computing Machinery, New York, NY, USA, pp. 292–303.
Li, Zhen, Zou, Deqing, Xu, Shouhuai, Jin, Hai, Zhu, Yawei, Chen, Zhaoxuan, 2021c. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput..
Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Zhang, Y., Chen, Z., Li, D., 2021d. Vuldeelocator: A deep learning-based system for detecting and locating software vulnerabilities. IEEE Trans. Dependable Secure Comput..
Li, Zhen, Zou, Deqing, Xu, Shouhuai, Ou, Xinyu, Jin, Hai, Wang, Sujuan, Deng, Zhijun, Zhong, Yuyi, 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.
Lin, Guanjun, Zhang, Jun, Luo, Wei, Pan, Lei, Vel, Olivier De, Montague, Paul, Xiang, Yang, 2019. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans. Dependable Secure Comput. 18 (5), 2469–2485.
Lin, Guanjun, Zhang, Jun, Luo, Wei, Pan, Lei, Xiang, Yang, Vel, Olivier De, Montague, Paul, 2018. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inform. 14 (7), 3289–3297.
Liu, Shigang, Lin, Guanjun, Han, Qing-Long, Wen, Sheng, Zhang, Jun, Xiang, Yang, 2019. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans. Fuzzy Syst. 28 (7), 1329–1343.
der Maaten, Laurens Van, Hinton, Geoffrey, 2008. Visualizing data using t-sne. J. Mach. Learn. Res. 9 (11).
Mikolov, Tomas, Chen, Kai, Corrado, Greg, Dean, Jeffrey, 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
Mitra, Subhasish, Avra, L.J., McCluskey, Edward J., 1997. Scan synthesis for one-hot signals. In: Proceedings International Test Conference 1997. IEEE, pp. 714–722.
Nguyen, Van-Anh, Nguyen, Dai Quoc, Nguyen, Van, Le, Trung, Tran, Quan Hung, Phung, Dinh, 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. pp. 178–182.
Peng, Hui, Shoshitaishvili, Yan, Payer, Mathias, 2018. T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy. (SP), IEEE, pp. 697–710.
Pennington, Jeffrey, Socher, Richard, Manning, Christopher D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing. (EMNLP), pp. 1532–1543.

Russell, Rebecca, Kim, Louis, Hamilton, Lei, Lazovich, Tomo, Harer, Jacob, Ozdemir, Onur, Ellingwood, Paul, McConley, Marc, 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.

Sui, Yulei, Cheng, Xiao, Zhang, Guanqin, Wang, Haoyu, 2020. Flow2vec: Value-flow-based precise code embedding. In: Proceedings of the ACM on Programming Languages, 4(OOPSLA):. pp. 1–27.

Sui, Yulei, Xue, Jingling, Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th international conference on compiler construction.

Tomek, Ivan, 1976. Two modifications of cnn.

Velickovic, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Lio, Pietro, Bengio, Yoshua, 2017. Graph attention networks. Stat 1050, 20.

Wan, Yao, Zhao, Wei, Zhang, Hongyu, Sui, Yulei, Xu, Guandong, Jin, Hai, 2022a. What do they capture? a structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2377–2388.

Wan, Yao, Zhao, Wei, Zhang, Hongyu, Sui, Yulei, Xu, Guandong, Jin, Hai, 2022b. What do they capture? a structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2377–2388.

Wang, Huanting, Ye, Guixin, Tang, Zhanyong, Tan, Shin Hwei, Huang, Songfang, Fang, Dingyi, Feng, Yansong, Bian, Lizhong, Wang, Zheng, 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wei, Hongwei, Lin, Guanjun, Li, Lin, Jia, Heming, 2021. A context-aware neural embedding for function-level vulnerability detection. Algorithms 14 (11), 335.

Wu, Shizhong, Guo, Tao, Dong, Guowei, Wang, Jiajie, 2012. Software vulnerability analyses: A road map. J. Tsinghua Univ. Sci. Technol. 52 (10), 1309–1319.

Xie, Tao, Tillmann, Nikolai, Halleux, Jonathan De, Schulte, Wolfram, 2009. Fitness-guided path exploration in dynamic symbolic execution. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE, pp. 359–368.

Xu, Keyulu, Li, Chengtao, Tian, Yonglong, Sonobe, Tomohiro, Kawarabayashi, Ken-ichi, Jegelka, Stefanie, 2018. Representation learning on graphs with jumping knowledge networks. In: International Conference on Machine Learning. PMLR, pp. 5453–5462.

Yamaguchi, Fabian, Golde, Nico, Arp, Daniel, Rieck, Konrad, 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Ye, Guixin, Tang, Zhanyong, Wang, Huanting, Fang, Dingyi, Fang, Jianbin, Huang, Songfang, Wang, Zheng, 2020. Deep program structure modeling through multi-relational graph-based learning. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. pp. 111–123.

Ying, Zhitao, Bourgeois, Dylan, You, Jiaxuan, Zitnik, Marinka, Leskovec, Jure, 2019. Gnnexplainer: Generating explanations for graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Zeng, Min, Zou, Beiji, Wei, Faran, Liu, Xiyao, Wang, Lei, 2016. Effective prediction of three common diseases by combining smote with tomek links technique for imbalanced medical data. In: 2016 IEEE International Conference of Online Analysis and Computing Science. (ICOACS), pp. 225–228.

Zhang, Chunyong, Liu, Bin, Fan, Qi, Xin, Yang, Zhu, Hongliang, 2022. Vulnerability detection with graph attention network and metric learning.

Zheng, Weining, Jiang, Yuan, Su, Xiaohong, 2021a. Vulspg: vulnerability detection based on slice property graph representation learning. arXiv preprint arXiv:2109.02527.

Zheng, Yunhui, Pujar, Saurabh, Lewis, Burn, Buratti, Luca, Epstein, Edward, Yang, Bo, Laredo, Jim, Morari, Alessandro, Su, Zhong, 2021b. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. (ICSE-SEIP), IEEE, pp. 111–120.

Zhou, Yaqin, Liu, Shangqing, Siow, Jingkai, Du, Xiaoning, Liu, Yang, 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Zou, Deqing, Wang, Sujuan, Xu, Shouhuai, Li, Zhen, Jin, Hai, 2019. μ Vuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput. 18 (5), 2224–2236.

Chunyong Zhang is a Ph.D. at Beijing University of Posts and Telecommunications and an engineer at the Disaster Recovery Technology Industry Alliance. His research interests include vulnerability detection, malware homology analysis, and disaster recovery techniques.

Yang Xin received the BE degree from the Department of Electronics, Shandong University in 1999, the MS degree from Shandong University of China in 2002, and the Ph.D. degree from Beijing University of Posts and Telecommunications in 2005. He is currently a professor at Beijing University of Posts and Telecommunications. His research direction is disaster recovery storage, network security, and information security.