Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Guard2Vul: Vulnerability Detection viaGradient-based Adversarial TrainingEnhanced Graph Learning

Hao Shen
Nantong University
Xiaolin Ju
Ju.xl@ntu.edu.cn
Nantong University
Wenjie Li
Anhui Normal University
Xiang Chen
Nantong University
Guang Yang
Nanjing University of Aeronautics and Astronautics

Research Article

Keywords: Vulnerability detection, Graph convolutional neural network, Abstract syntax tree, Gradient-based adversarial training

Posted Date: January 10th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-3840091/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Guard2Vul: Vulnerability Detection via Gradient-based Adversarial Training Enhanced Graph Learning

Hao Shen[1], Xiaolin Ju[1*], Wenjie Li[2], Xiang Chen[1*] and Guang Yang[3]

[1*]School of Information Science and Technology, Nantong University, Nantong, 226019, Jiangsu, China.
[2*]School of Computer and Information, Anhui Normal University, Wuhu, 241000, Anhui, China.
[3*]School of Information Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, Jiangsu, China.

*Corresponding author(s). E-mail(s): ju.xl@ntu.edu.cn; xchencs@ntu.edu.cn;
Contributing authors: shenhyc@gmail.com; wjl@ahnu.edu.cn; novelyg@outlook.com;

Abstract

Modern software security is challenged by vulnerabilities, which can lead to severe consequences, including loss of information, property, and privacy disclosure. Recently, graph-based deep learning methods have been proven as a promising solution for vulnerability detection at functional granularity. However, previous studies still face challenges, such as structural imbalances, complex connections in abstract syntax trees of functions, and insufficient training data. In this study, we propose Guard2Vul, which combines a graph neural network with a residual network to jointly capture deeper semantic and structural features, specifically, source code and node dependencies. Moreover, we utilize GraphSMOTE and DGD, a dropout-enhanced gradient adversarial training technique, to conduct data augmentation and automatically improve normalized stability. To demonstrate the effectiveness of Guard2Vul, we

evaluate its performance on four experimental subjects by different programming languages, such as C/C++ and Java. To show the competitiveness of Guard2Vul, we consider five Deep Learning-based vulnerability detection approaches (i.e., TokenCNN, Sysevr, VulDeePecker, Devign, and Reveal) as baselines. The results indicate that Guard2Vul outperforms these baselines, achieving at least 13.4%, 28.9%, 6.2%, and 12.1% higher F1-measure on four experimental subjects. Finally, we perform ablation experiments to demonstrate the effectiveness of our customized components, namely enhanced graph representation learning and the gradient-based adversarial training method, in Guard2Vul.

Keywords: Vulnerability detection, Graph convolutional neural network, Abstract syntax tree, Gradient-based adversarial training

1 Introduction

Lurking software vulnerabilities, such as the reentry vulnerabilities in the decentralized autonomous organization contract [1] and the exploitation of the National Security Agency's Eternal Blue vulnerabilities [2], pose a significant security risk to cyberspace. According to Özkan's analysis [3], the National Institute of Standards and Technology (NIST) has identified more than twice the amount of vulnerabilities in recent years than it did over a decade ago, with the total number of common vulnerabilities exceeding 160,000.

Vulnerability detection aims to identify potential security weaknesses in software systems by analyzing the source code or the program's behavior during execution [4]. As modern software becomes increasingly complex [5], manual detection of vulnerabilities is costly. Automated techniques offer a more practical approach to vulnerability detection.

Previous software vulnerability detection approaches can be divided into two categories: conventional approaches and machine learning (ML)-based approaches [6]. Conventional detection approaches rely on predefined vulnerability rules or features specified by experts [7]. These approaches depend heavily on expert knowledge, and detecting new vulnerabilities is challenging. However, functional logic in software projects is often complex and subject to modifications, posing a challenge in accurately describing it with established rules. Consequently, conventional detection methods struggle to achieve low false positives and false negatives. In response to this challenge, researchers have proposed machine learning (ML)-based approaches [8]. These approaches eliminate the need for experts to define vulnerability rules and can effectively capture the program semantics necessary for detecting potential software vulnerabilities. Previous approaches, such as those employed by Russell et al. [9] and Li et al. [10], treated source code as token sequences and utilized deep learning (DL) techniques to extract syntactic and semantic vulnerability features. However, source code exhibits a greater degree of logical and structured syntax compared to natural language. Consequently, treating source code as

natural language leads to subpar performance. Neural networks for processing source code need to learn non-flattened sequential feature representations to overcome this limitation.

Recently, Zhou et al. [11] proposed the Devign method to address the needs of vulnerability prediction. Devign leverages the Gated Graph Neural Network (GGNN) technique to acquire a comprehensive understanding of code semantics for graph-level classification. Additionally, it incorporates a novel Conv module to extract valuable features from the learned rich node representations, enabling effective graph-level classification. Similarly, Chakraborty et al. [8] introduced the Reveal method, which utilizes the gated graph recurrent network model [12] to extract code features and conduct graph-level classification. Both studies observed that Devign and Reveal outperformed other non-graph methods in terms of F1-measure, highlighting the effectiveness of graph neural networks in code processing. However, Devign and Reveal still suffer from the following three limitations:

(1) The class imbalance in code graph information. Devign uses a balanced dataset and does not handle imbalanced situations. Reveal handles imbalanced code by using SMOTE [13] to add samples for a few classes. However, when code is parsed into a graph structure, SMOTE may lose the structural information of the graph, leading to inconsistencies between the learned information and the actual situation, thus reducing prediction performance.

(2) The representation of code dependencies and semantics. Devign and Reveal rely on GNN-based models to generate vector representations for their graph models during the prediction process. However, GNN models often struggle to learn complex source code dependencies and semantics [14]. This limitation may result in less meaningful vector representations and inaccurate predictions.

(3) The generalization ability of GNN. During the training process of Devign and Reveal, the combination of various graph structures and node features is not fully observed. Although GNN models have significant advantages in learning code structure semantics compared to CNN or RNN, they may struggle when encountering new graph structures or node features that differ from the training set. This can lead to inaccurate predictions, potentially causing the model to miss or misreport vulnerabilities in practical applications.

Therefore, we address the above issues from three perspectives. **First**, we introduce the GraphSMOTE [15] technique to alleviate the class imbalance issue of graph structure data, which extends the traditional SMOTE algorithm to preserve the structural information of the graph, ensuring that the generated samples are consistent with the original graph distribution. **Second**, we propose an approach Guard2Vul, which combines GraphSAGE [16] enhanced by residual neural network [17] and a channel attention mechanism (Aesnet). By leveraging the power of residual connections and channel attention, Guard2Vul can effectively capture deep dependencies and code semantics, enabling more

meaningful vector representations and accurate predictions. **Third**, we augment the data through dropout-enhanced gradient adversarial training DGD to improve the model's generalization ability. This technique helps the model adapt to new graph structures and node features that differ from the training set, improving the accuracy and reliability of vulnerability detection.

We conducted experiments by comparing Guard2Vul with five state-of-the-art baseline methods, namely TokenCNN[9], SySeVR [18], VulDeePecker [19], Devign[11], and Reveal [8]. Then, we evaluated their performance on three C/C++ open-source datasets and one Java open-source dataset. The experimental results show that Guard2Vul can improve the performance of F1 measurement by an average of 6.2%∼28.9%.

The main contributions of this paper are as follows:

- We propose a novel approach, Guard2Vul, to extract deep semantic information from source code and augment semantic features for effective vulnerability detection.
- We design a dropout-enhanced gradient adversarial training method to improve the data representation and augment semantic features.
- We compare Guard2Vul with five state-of-the-art DL-based approaches and found that it outperforms them by at least 13.4%, 28.9%, 6.2%, and 12.1% in F1-measure on one Java project and three C/C++ projects.
- We share Guard2Vul for to encourage future studies on vulnerability detection[1].

The rest of this paper is organized as follows. Section 2 describes the background of this research topic. Section 3 presents the proposed model with detailed description. Section 4 describes the design of research questions and experimental methods, evaluation metrics. Section 5 evaluates the performance of Guard2Vul, including result comparison and analysis. Section 6 discussed the threats to the validity of our work. Finally, we summarized the paper in the last section.

2 Background and Research Movitation

Software vulnerabilities are defects or vulnerable code in software implementation caused by the design or encoding of the software system [20]. In software security, vulnerabilities may cause internal system crashes or allow attackers to control the system by crossing permission boundaries within the computer system. Therefore, two automatic vulnerability detection methods have been proposed based on program analysis (PA) and ML/DL-based techniques [6].

PA-based approaches mainly rely on static analysis, dynamic analysis, or hybrid analysis to detect vulnerabilities [21]. However, these traditional methods are limited by high false positives and computational costs, etc. ML/DL-based methods include a) vulnerability prediction model based on

[1] https://github.com/VulDetection/Guard2Vul

software metrics, b) anomaly detection, and c) vulnerable code pattern recognition [22]. With the increasing interest in machine learning methods for software vulnerability detection, primarily the breakthrough of deep learning methods, some studies have discussed the potential of deep learning in vulnerability detection.

2.1 Graph-based Vulnerability Detection

The traditional vulnerability detection approach based on deep learning involves detecting differences in expert metrics [23–25]. However, code metrics may sometimes be identical, leading to misjudgment. To address this issue, many DL-based approaches have been proposed to extract the semantic features of the code and learn the vulnerability patterns from the historical code data [9, 11, 18, 26, 27]. For instance, Russell et al. [9] extracted code features based on RNN to capture longer tag dependencies and combine them with a random forest (RF) classifier to detect vulnerabilities. Similarly, Xu et al. [28] utilized Context-LSTM (CLSTM) to merge code context features into the model for vulnerability detection.

Although DL-based approaches for vulnerability detection have proven effective, they still have limitations as they only learn code features sequentially without considering the code's structural attributes. Li et al. [19] analyzed the code for different types of graph attributes and then extracted the graph attribute features through an RNN model. However, this approach still flatly studied code features without using structural information. Recently, many studies have utilized graph neural networks (GNNs) to learn the structural features of code. For example, Cao et al. [29] extracted the syntax and semantic information of source code through ASTs, control flow graphs (CFGs), and data flow graphs (DFGs), and constructed bidirectional GNN (BGNN) to learn different features between vulnerable and non-vulnerable code. Similarly, Wu et al. [30] modeled the software function as a simplified code attribute graph (SCPG) and used GNNs and multi-layer perceptrons (MLPs) to learn graph representations.

2.2 Data Augmentation in Deep Learning

Although many DL-based approaches are proposed to learn structural and semantic features, there is still room for improvement. Many data augmentation approaches have been proposed to improve model detection performance. For example, Wen et al. [31] applied an edge-aware graph convolution network module to integrate heterogeneous edge information into node representation and a kernel scale representation module to effectively capture the relationship between remote graph nodes and improve detection ability. Similarly, Tang et al. [32] identified the vulnerability pattern accurately by retaining the path semantics in the gadget and building a convolutional neural network (CNN) with a multi-layer attention mechanism to learn from the code flexible length.

However, these above approaches have made limited progress. Recently, Ganin et al. [33] proposed an adversary training model that can improve robustness and performance by attacking and defending the model. This model has been applied in many research areas. For instance, Abdelaty et al. [34] utilized generative adversarial networks (GANs) to generate anti-DDoS samples and improve intrusion detection capability. Similarly, Hasegawa et al. [35] constructed a robust model for adversarial examples and modified the samples generated by the attack through the learning gate.

2.3 Research Movitation

Researchers have proposed some DL-based vulnerability detection methods currently. However, current methods cannot learn deeper semantic information to detect complex vulnerabilities. Therefore, some fine-grained graph-based vulnerability prediction methods have recently been proposed, achieving better results than token-based methods. However, these vulnerability detection methods still have three limitations.

Limitation ①: They are suboptimal in handling class imbalance in code graph information. Data imbalance remains an open challenge [36] in software vulnerability detection, as flawed samples are far fewer than clean samples, making it difficult to learn accurate classification models. As is well known, when the data ratio is highly imbalanced, the classifier favors the majority class. Most of the existing graph-based methods do not consider or use simple SMOTE oversampling, ignoring the structural information of the topology map. The synthesized samples generated by GraphSMOTE can better preserve the topological structure of the original dataset, as it uses a graphical structure-based method to construct new samples, resulting in a more realistic and trustworthy synthesized sample.

Limitation ②: Their GNN architecture still cannot effectively capture meaningful deep dependencies and semantics of source code. The existing DL-based methods rely on GNN or RNN-based [37] architectures to generate code representations, which can encounter problems when dealing with complex sequences. The RNN model processes the sequence token by token, where the model considers the context vector of the previous token for long-term dependencies when processing each token. However, given a complex sequence (such as a sequence of 500 tokens), the context vector is complicated to capture enough complex dependencies. However, graph-based models typically only contain one or a few loop layers, which are not as deep as ResNet. Therefore, these limitations may reduce the significance of the generated feature representations, further negatively affecting the accuracy of vulnerability prediction models.

Limitation ③: Their generalization ability is limited. GNN may not adapt well to unknown vulnerability types or variants. When learning feature representations, it may be limited by existing training data and known vulnerability types, resulting in limited detection ability for unknown vulnerabilities. This may reduce the accuracy and reliability of the model when facing new

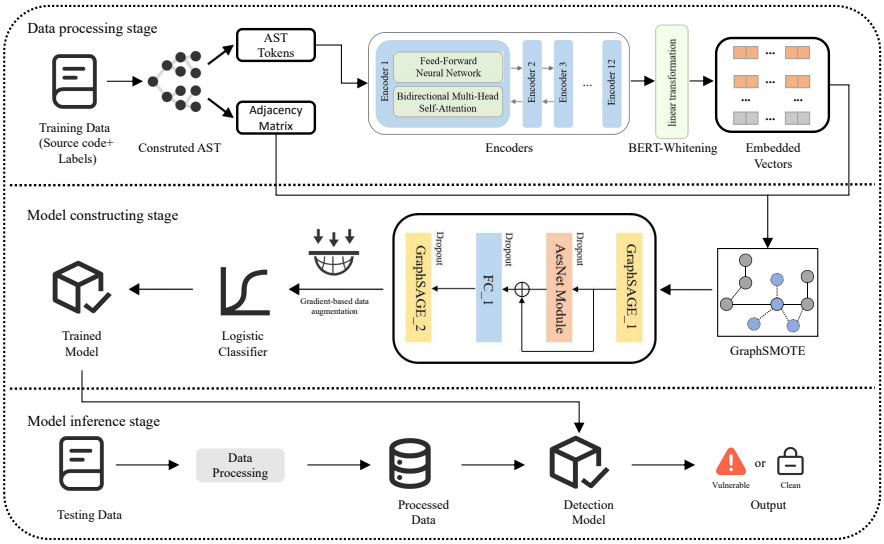


Fig. 1 Overview of Guard2Vul

types of vulnerabilities. It is sensitive to noise and outliers in input data and lacks robustness. In actual vulnerability detection scenarios, there may be various noise and abnormal situations in the code, which requires the model to have a certain degree of robustness to cope with these situations.

3 Our Approach

In this section, we present the design rationale and critical techniques of our Guard2Vul, which addresses three limitations of current DL-based methods. Firstly, we do not use project-specific training data models but instead the pre-trained model CodeBERT to generate vector representations of the source code. The pre-trained model CodeBERT [38] is pre-trained on CodeSearchNet (20GB code corpus), which can generate more accurate vector representations. Secondly, we did not use the basic GNN model to generate code representations. Instead, we utilized ResNet (AesNet)[17] with a channel attention mechanism to improve the GraphSAGE architecture, which can use residual operations to capture complex dependencies in long sequences. Thirdly, we have added the dropout-enhanced gradient adversarial training method DGD, which increases the noise and outliers in the input data, enabling the model to learn more extensive and diverse feature representations, thereby better capturing the hidden and mutated features of vulnerabilities.

We design the Guard2Vul method as a three-step method: data processing stage, Model constructing stage, and model inference stage. Fig. 1 shows the overall architecture of the Guard2Vul method.

3.1 Data Processing Stage

We parse the codes into ASTs to obtain the code semantic features. Then, we utilize CodeBERT to transform the code semantic features into vectors. Later, we utilize BERT-whiting [39] to reduce the dimension. Finally, we obtain the vector representation corresponding to each vulnerable or secure code node. We describe the details of each part in the following.

Firstly, we extract the source code from the collected vulnerability files, i.e., error files with commented vulnerability information. After collecting the source code, we utilize the open-source tool Javalang[2] and Clang[3] to parse the Java and C/C++ source code and build ASTs. Javalang and Clang are primarily designed to parse function code and may be unable to parse extraneous content within source files, which can impede data processing. To address this issue, we consider the aspect of processing time consumption and remove files where parsing errors occur. Each AST has a set of node collections corresponding to the source code. The generated tokens will differ from the source code, such as MemberReference, MethodDeclaration, and BasicType. However, we can still find the specific location of the tokens in the source code according to the AST syntax rules. We traverse the graph composed of ASTs to obtain an adjacency matrix with graph structure information. The AST can be treated as a special undirected graph, so the adjacency matrix obtained is symmetric on the diagonal. We set the value in the adjacency matrix to 0 or 1, depending on whether the code corresponding to the marker is vulnerable.

Then, we utilize word embedding to convert the tokens corresponding to the source code obtained by traversing the AST into a fixed-length vector. Specifically, we use a popular approach, CodeBERT [38], to accomplish word embedding. CodeBERT is a pre-trained model based on a multi-layer bidirectional transformer encoder structure that achieves the best performance on code retrieval tasks. We can convert each code token into its corresponding vector representation with a fixed dimension of 768 by the CodeBERT model. Like Son et al. [40], we use the first line of each token representation as the semantic representation of the whole token.

With the above processing, we can transform the vulnerability-related codes in the dataset into a set of row vectors $\{\mathbf{X_i}\}_{i=1}^{N}$, where N denotes the number of rows. We further process the semantic vectors through the BERT-whitening operation to reduce the model's overhead and obtain better performance. Firstly, BERT-whitening performs the following linear transformation on the row vector $\{\mathbf{X_i}\}_{i=1}^{N}$:

$$\tilde{X}_i = (X_i - \mu)\, W. \tag{1}$$

[2]https://github.com/c2nes/javalang
[3]https://github.com/llvm-mirror/clang

To set the mean value to 0 and the covariance matrix to the unit matrix, we can solve for μ using equation (2):

$$\mu = \frac{1}{N}\sum_{i=1}^{N} X_i. \tag{2}$$

We record the covariance of the original data to solve W as (3):

$$\sum = \frac{1}{N}\sum_{i=1}^{N} (X_i - \mu)^T (X_i - \mu), \tag{3}$$

where $\sum$ is a positive definite symmetric matrix, which can be decomposed into orthogonal matrix U and positive diagonal matrix Λ by SVD as (4):

$$\sum = U\Lambda U^T. \tag{4}$$

Then, W from equation (1) can be calculated as (5):

$$W = \sqrt{\Lambda^{-1}}U. \tag{5}$$

Finally, we can get the semantic vector $\{\tilde{X}_i\}_{i=1}^{N}$ corresponding to $\{X_i\}_{i=1}^{N}$.

3.2 Model Constructing Stage

Firstly, we perform class imbalance processing through GraphSMOTE oversampling. Then, we construct the Guard2Vul model to learn the code semantic features. Later, we construct an adversarial training model to augment the data. Finally, we utilize the Logistic classifier to distinguish vulnerable code. We describe the details of each part in the following.

Because vulnerable code statements are far less than non-vulnerable statements in programs. The prediction results will be outstanding when such data are trained directly because the model is biased to predict the results of most categories. We need to balance the data to avoid this incorrect prediction. People usually assume that the proportion of each category is almost the same in the classification problem. In real life, this kind of balance rarely exists. In many cases, the sample data of one or several categories will be significantly lower than that of others. Assuming that the ratio of the two types of samples is as high as 10:1, the model only needs to predict that the new samples are of the majority class, and its accuracy will be very high. However, this result is meaningless for the current research scenario. To address this problem, We applied GraphSMOTE to balance vulnerable and secure code in the training set, allowing the model to learn each feature type more effectively.

We extract feature information by parsing the source code into AST node tokens and node relationships as the input of Guard2Vul. We employ a residual neural network (ResNet), a neural network model that applies identity

functions. The ResNet has a more robust training effect on deeper neural networks under the same optimization conditions. However, the complexity of the model is higher, and the deep neural network will have severe over-fitting problems and gradient disappearance/explosion. He et al. [17] proposed residual networks solve this problem effectively. It introduces a forward feedback shortcut connection between input and output called identity mapping. It can be expressed as (6):

$$y = \sigma\left(F\left(x, W\right) + x\right), \tag{6}$$

where y represents the output of the residual block, $\sigma(\cdot)$ represents the activation function, $F(\cdot)$ represents the residual function, x represents the input, and W represents all the weights in the residual block.

Moreover, a channel attention mechanism has been added to ResNet, improving the model's ability to emphasize informative channels in the feature selectively. This mechanism helps to enhance the representation power of ResNet by dynamically recalibrating the importance of different channels based on their relevance to the task at hand. By incorporating channel attention, ResNet can better capture and utilize channel-wise dependencies, improving performance in vulnerability detection tasks. We will refer to ResNet by adding a channel attention mechanism as AesNet. The output of AesNet can be updated as (7):

$$y = \sigma\left(F\left(\mathrm{CA}\left(x\right) \cdot x, W\right) + x\right), \tag{7}$$

where $\mathrm{CA}(\cdot)$ denotes the channel attention mechanism. The channel attention mechanism can be represented as (8):

$$\mathrm{CA}\left(x\right) = \sigma\left(\mathrm{MLP}\left(\mathrm{AvgPool}\left(x\right)\right)\right), \tag{8}$$

where AvgPool denotes average pooling and MLP denotes a multi-layer perception. The output of the average pooling operation is fed to the multi-layer perceptron, which consists of one or more fully connected layers with a nonlinear activation function, such as *ReLU*. The output of the MLP is then passed through a sigmoid activation function to obtain the channel attention weights, which are element-wise multiplied by the input feature map x.

The ResNet module is composed of four CNN layers with two dropout layers, which are shown as (9):

$$\begin{cases} h_1 = \mathrm{CNN}_1\left(x\right) \\ h_2 = \mathrm{Dropout}\left(h_1\right) \\ h_3 = \mathrm{CA}\left(\mathrm{CNN}_2\left(h_2\right)\right) \\ h_4 = \mathrm{Dropout}\left(h_3\right) \\ h_5 = \mathrm{CA}\left(\mathrm{CNN}_3\left(h_4\right)\right) \\ h_6 = \mathrm{CA}\left(h_1 + h_5\right) \\ y = \mathrm{CNN}_4\left(h_6\right) \end{cases}, \tag{9}$$

where $Dropout(\cdot)$ is the dropout layer, and x and y are the inputs and outputs in AesNet, respectively.

We embed ResNet in two GraphSAGE layers and apply the FC layer to alleviate the resulting instability caused by residual connection. Our model can be defined as (10):

$$\begin{cases} f_1 = \mathrm{SAGE}_1\left(x, A\right) \\ f_2 = \mathrm{AesNet}\left(\mathrm{Dropout}\left(f_1\right)\right) \\ f_3 = \mathrm{FC}\left(f_1 + f_2\right) \\ \hat{y} = \mathrm{SAGE}_2\left(\mathrm{Dropout}\left(f_3\right), A\right) \end{cases}, \tag{10}$$

where x and $\hat{y}$ are input and output, A represents the graph's topology, and $SAGE_i$ represents the i-th $SAGE$ layer. We utilize GraphSAGE to learn the code shallow features and AesNet to learn more code detailed features. Finally, we utilize stochastic gradient descent (SGD) [41] to obtain the representation of the given code.

Dropout is a widely used regularization technique in neural networks, which can prevent overfitting by randomly discarding elements. Inspired by Wager et al. [42], we incorporated dropout regularization into the PGD adversarial training process to enhance the robustness of the network against adversarial attacks. Specifically, we add a dropout layer after the input layer and before each hidden neural network layer. During training, we randomly discard a certain proportion of input and hidden units in each forward transfer. For the input embedded vector x, DGD aims to add disturbances to it and generate K adversarial examples as (11):

$$\tilde{x} = x + r, \tag{11}$$

where r is the disturbance generated by DGD, which can be calculated as (12):

$$r = \epsilon \cdot sign(\nabla_x J(\theta, x, y)) \odot mask, \tag{12}$$

where ϵ is the maximum size of the disturbance, $sign(\cdot)$ is a symbolic function, $\nabla_x J(\theta, x, y)$ is the gradient of the loss function J with respect to the input x and the target label y, and $mask$ is a mask matrix for the Dropout operation, which randomly selects 0 or 1 on each element. The mask matrix for the dropout operation is randomly generated during each forward propagation. The loss function of dropout regularization can be calculated as (13):

$$loss = \frac{1}{N} \sum_{i=1}^{N} L(\theta, \tilde{x_i}, y_i) + \lambda \sum_{i=1}^{n_{layer}} W_{iF}^2, \tag{13}$$

where N is the number of samples, θ is the model parameter, $\tilde{x_i}$ is the sample after adding the disturbance, and y_i is the tag corresponding to the sample, λ is the regularization parameter, n_{layer} is the number of layers of the neural network, W_i is the weight parameter for layer i.

During each DGD adversarial training process iteration, we first generate confrontation samples by applying PGD attacks to the current model with

dropout regularization. Then, we combine original and adversarial loss, using both original and adversarial examples to train the network.

After learning the code feature, we use a classification model to distinguish between vulnerable and non-vulnerable codes. The weights and biases in the logistic classifier are pre-determined and used to train the model using labeled training files and predict the code in the test set.

3.3 Model Inference Stage

This stage is used to evaluate the performance of the trained model. The given software code (i.e., vulnerable and clean code) is initially processed and converted into its corresponding vector representation. Next, the vector representation of the code is fed into the trained model to detect any vulnerabilities. Finally, the model provides information on whether there are any vulnerabilities.

Although a model is trained, its performance is often uncertain. Therefore, to ensure that our model can effectively detect vulnerabilities, it must be evaluated by accurately identifying vulnerable code.

4 Empirical Study

We evaluate the effectiveness of Guard2Vul in vulnerability detection tasks by answering the following research questions (RQs):

RQ1: How effective is Guard2Vul in detecting vulnerabilities compared to state-of-the-art DL-based vulnerability detection approaches?

RQ2: How does enhanced graph representation learning contribute to the performance of Guard2Vul?

RQ3: How does the dropout-enhanced gradient adversarial training method contribute to the performance of Guard2Vul?

4.1 Experimental Subjects

Java and C/C++ have been among the top three programming languages through various software industry metrics for the past decade, according to the TIOBE index [4]. Many current vulnerability detection approaches are for C/C++ projects, so we collected vulnerabilities in Java code to compensate for the disadvantage in this area. We collected different vulnerability codes from SARD, a sample vulnerability dataset maintained by NIST. We utilized vulnerability types (i.e., CWE-IDs) as our search criteria. We collect code covering ten types of common vulnerabilities in previous studies (including CWE-78, -89, -90, -129, -134, -191, -400, -404, -476, -643) for model training and evaluation. We collected 25,782 vulnerable functions and 30,648 non-vulnerable functions in total. We also utilized a recently open-sourced MSR, Devign, and Reveal datasets [8, 11, 43, 44]. These three datasets have been widely used in previous research. The Table 1 summarizes characteristics of four datasets.

[4] https://www.tiobe.com/tiobe-index/

Table 1 Characteristics of vulnerability dataset

Dataset	Language	#Vul	#Non-vul
SARD	Java	25,782	30,648
MSR	C/C++	10,547	168,752
Devign	C/C++	10,067	12,294
Reveal	C/C++	1,664	16,505

4.2 Evaluation Metrics

By following previous studies [9, 45], we choose four widely used metrics: AUC, F1-measure, recall, and precision, to evaluate the performance of Guard2Vul and the baseline models. AUC represents the area enclosed with the coordinate axis under the ROC curve and indicates the probability that the detected non-vulnerable code ranks ahead of the detected vulnerable code. F1-measure represents a derived effectiveness measure that measures the summed average of precision and recall. The recall represents the percentage of detected vulnerable codes among the real vulnerable codes. Precision represents the proportion of real vulnerabilities in the detected vulnerable code. F1-measure is utilized to measure the stability of Guard2Vul, and AUC is utilized to evaluate its discriminatory ability.

4.3 Baselines

To evaluate the performance of Guard2Vul, we select four state-of-the-art approaches as baselines. These approaches have been widely used as baselines in recent work [46, 47] and have been proven effective in detecting vulnerable code.

- TokenCNN [9] converts the source code into tokens through lexical analysis. Then, CNN and RF algorithms are applied to achieve the best classification results.
- Sysevr [18] expands the vulnerability syntax features to accommodate semantic information induced by data and control dependency. It has been shown that BGRU is more effective than CNNs.
- VulDeepecker [19] converts the source code of the training set into a code gadget code set, then marks the sample, converts the sample into a processable feature vector, inputs the feature vector into the BLSTM network, obtains the model parameters, and tests the test set to determine whether there are vulnerabilities in the program.
- Devign [11] introduces a novel Convolutional module that can efficiently extract valuable features from rich node representations learned for graph-level classification.

- Reveal [8] proposes a two-phase approach to code vulnerability detection, consisting of feature extraction and training. They utilize a GGNN to extract features.

4.4 Implementation

We extract the structural and semantic features from the source code by parsing it and then use CodeBERT to embed the resulting words. We construct an adversarial training model named DGD to enhance the robustness and augment the data. During the training process, Guard2Vul receives code semantic features and their corresponding tags (i.e., vulnerable and non-vulnerable). Our model undergoes repeated training to optimize the training loss. During the testing process, new code semantic representation vectors and their corresponding tags are inputted into the model, and the model outputs the detection results. The performance of our model is evaluated by calculating various indicators such as AUC and F1-measure based on the obtained results. To conduct our experiments, we split the dataset into training and testing sets, with a ratio of 7:3. Each experiment is repeated 25 times, and the final evaluation result is obtained by taking the average value, which helps to reduce the deviation caused by random data. Since open-source code for TokenCNN and Devign was unavailable, we re-implemented this method by following each step outlined in the original text.

We implemented Guard2Vul with Python using PyTorch. The hyper-parameter settings for our experiment can be found in Table 2. To optimize the loss, we used the SGD algorithm. Our approach was executed on machines running 64-bit Windows 10, with an Intel Core i5-9300H CPU @ 2.40GHz, and an NVIDIA GeForce GTX 4090.

Table 2 Hyper-Parameter settings in Guard2Vul

Category	Hyper-parameters	Value
Deep Learning Part	Dropout rate	0.1
	Learning rate	0.001
	Optimizer	SGD
	Epoch	100
Data Processing Part	BERT-whitening length	128
	K	3
	DGD dropout-1 rate	0.1
	DGD dropout-2 rate	0.3

5 Result Analysis

5.1 RQ1: How effective is Guard2Vul in detecting vulnerabilities compared to state-of-the-art DL-based vulnerability detection approaches?

RQ1 aims to investigate whether a software vulnerability detection system based on GNN with ResNet improvement and data augmentation is effective. To answer RQ1, we compared it with some effective baseline approaches for the same task. We selected the most effective DL-based models from previous studies as baselines.

We conducted experiments using Guard2Vul and five baseline approaches on MSR, Devign, Reveal and the collected SARD datasets. Table 3 shows the

Table 3 Comparison with DL-based approaches

Dataset	Approach	AUC	Recall	Precision	F1-measure
SARD	TokenCNN	0.582	0.582	0.572	0.577
	Sysevr	0.591	0.632	0.525	0.573
	VulDeePecker	0.549	0.402	0.589	0.478
	Devign	0.585	0.573	0.605	0.589
	Reveal	0.596	0.542	0.651	0.592
	Guard2Vul	**0.635**	**0.657**	**0.685**	**0.671**
MSR	TokenCNN	0.501	0.270	0.149	0.192
	Sysevr	0.503	0.141	0.309	0.193
	VulDeePecker	0.501	0.128	**0.384**	0.192
	Devign	0.509	0.160	0.306	0.210
	Reveal	0.512	0.340	0.172	0.229
	Guard2Vul	**0.517**	**0.369**	0.246	**0.295**
Devign	TokenCNN	0.526	0.407	0.548	0.467
	Sysevr	0.531	0.588	0.461	0.517
	VulDeePecker	0.528	0.326	0.461	0.381
	Devign	0.538	0.647	0.525	0.580
	Reveal	0.525	**0.707**	0.555	0.622
	Guard2Vul	**0.606**	0.679	**0.643**	**0.661**
Reveal	TokenCNN	0.506	0.527	0.162	0.248
	Sysevr	0.511	0.249	0.401	0.307
	VulDeePecker	0.514	0.131	0.211	0.162
	Devign	0.502	0.367	0.316	0.339
	Reveal	0.527	**0.611**	0.316	0.416
	Guard2Vul	**0.560**	0.499	**0.438**	**0.467**

experimental results. Guard2Vul outperforms the AUC, F1-measure, recall, and precision than the baselines. Specifically, VulDeePecker only implemented 40.2% of recalls on the SARD dataset. In contrast, Guard2Vul's recall rate is as high as 65.7%, more than 50% (1.63x) higher. On the MSR dataset, Guard2Vul's recall is also much higher than VulDeePecker(1.88x). Poor recall rates indicate that VuldeePecker is unable to detect vulnerabilities accurately. The possible reason for this is that VuldeePecker only considers data flow and ignores control flow information. On the four datasets, Guard2Vulcan

outperform baselines by at least 6.2% and maximum by 28.9% on F1-measure. Although Devign and Reveal also process code in the form of graphs, their ability to obtain deeper code semantic features may be limited due to model constraints. ResNet can compensate for the limitations of GNN in this domain, resulting in more precise prediction performance.

Next, we examine the difference between the techniques listed in Table 3 using the Wilcoxon rank-sum test to calculate the p-values [48]. In our empirical study, we want to evaluate the difference between them. The significance level is set at 0.05, which means that if the corrected p-values are less than 0.05, there is a significant difference between the two compared vectors. We compared Guard2Vul with five software vulnerability detection (SVD) techniques: TokenCNN, Sysevr, VulDeePecker, Devign and Reveal. All corrected p-values obtained are all 0.021, which is far less than 0.05, meaning that Guard2Vul significantly differs from SVD techniques. The results indicate that compared to the baseline method, the method with deep semantic assistance has significant improvements.

In general, Guard2Vul can achieve an improvement of at least 13.4%, 28.9%, 6.2% and 12.1% on F1-measure, respectively, compared with other DL-based approaches in the four datasets. This improvement is due to the ability of our approach to learn better and utilize the critical syntax and semantics of programs with the help of deep code semantic features.

Summary for RQ1: By utilizing data augmentation and fully leveraging deep features, Guard2Vul outperforms state-of-the-art DL-based approaches regarding vulnerability detection performance.

5.2 RQ2: How does enhanced graph representation learning contribute to the performance of Guard2Vul?

RQ2 aims to investigate the impact of AesNet on the performance of the Guard2Vul model. To this end, we experimented with the improved and original model in four datasets. We respectively take the four metrics AUC, recall, precision, and F1-measure as evaluation indicators. We set the K, *dropout*-1, and *dropout*-2 in DGD to 3, 0.1, and 0.3, respectively. To simplify our description, we have given three notations to represent four models, namely Guard2Vul w/o AesNet, Guard2Vul with ResNet, and Guard2Vul with AesNet, whose detailed descriptions are given as follows:

- Guard2Vul w/o AesNet means that using our framework without AesNet.
- Guard2Vul with ResNet means that using our framework with ResNet.
- Guard2Vul with AesNet means that using our framework with AesNet.

The experimental results are shown in Table 4.

Comparative experiments on all datasets have shown that using AesNet-improved models can achieve the best results. We found that the results

improved by AesNet showed a more significant improvement compared to the original ResNet improved model. Improvements of 7.3%, 42.9%, 10.1%, and 17.8% were achieved on F1-measure on four datasets, respectively. However, the ResNet improved model showed much less improvement on F1-measure, with only a 3.0% improvement in Design.

Table 4 Experimental results of AesNet

dataset	method	AUC	Recall	Precision	F1-measure
SARD	Guard2Vul w/o AesNet	0.583	0.649	0.602	0.625
	Guard2Vul with ResNet	0.635	0.664	0.67	0.667
	Guard2Vul with AesNet	0.635	0.657	0.685	0.671
MSR	Guard2Vul w/o AesNet	0.500	0.261	0.171	0.207
	Guard2Vul with ResNet	0.513	0.376	0.203	0.264
	Guard2Vul with AesNet	0.517	0.369	0.246	0.295
Devign	Guard2Vul w/o AesNet	0.572	0.670	0.557	0.600
	Guard2Vul with ResNet	0.590	0.677	0.609	0.641
	Guard2Vul with AesNet	0.606	0.679	0.643	0.661
Reveal	Guard2Vul w/o AesNet	0.529	0.397	0.395	0.396
	Guard2Vul with ResNet	0.546	471	0.408	0.437
	Guard2Vul with AesNet	0.560	0.499	0.438	0.467

Moreover, we found that the improvement in the MSR dataset is higher than that of other datasets, which may be because MSR is from the real world, and deeper code semantic information helps detect more practical vulnerabilities. The vulnerability pattern of a synthesized dataset is simpler than MSR. Compared with approaches without AesNet, Guard2Vul can learn more about software vulnerability patterns.

Summary for RQ2: Using AesNet can significantly improve the performance of Guard2Vul as it can effectively capture the deep features of vulnerable code.

5.3 RQ3: How does the dropout-enhanced gradient adversarial training method contribute to the performance of Guard2Vul?

RQ3 aims to investigate the impact of DGD on the performance of vulnerability detection tasks. We used three adversarial training approaches to add disturbance data to the model and conducted experiments on four datasets separately. We set the K, *dropout*-1, and *dropout*-2 in DGD to 3, 0.1, and 0.3, respectively. The AUC, recall, precision, and F1-measure are evaluation metrics. We also discussed whether *dropout* is set to the same value.

Based on the experimental results of RQ2, we utilized the AesNet improved model to experiment in this section. To simplify our discussion, we given

Table 5 Experimental results of Data Augmentation

Dataset	Approach	AUC	Recall	Precision	F1-measure
SARD	Guard2Vul w/o DGD	0.567	0.632	0.592	0.611
	Guard2Vul with FGM	0.609	0.606	0.657	0.630
	Guard2Vul with PGD	0.588	0.656	0.604	0.629
	Guard2Vul with DGDs	0.627	0.664	0.670	0.667
	Guard2Vul with DGD	0.635	0.657	0.685	0.671
MSR	Guard2Vul w/o DGD	0.501	0.224	0.198	0.210
	Guard2Vul with FGM	0.509	0.277	0.224	0.248
	Guard2Vul with PGD	0.511	0.261	0.218	0.238
	Guard2Vul with DGDs	0.510	0.357	0.220	0.272
	Guard2Vul with DGD	0.517	0.369	0.246	0.295
Devign	Guard2Vul w/o DGD	0.554	0.629	0.545	0.584
	Guard2Vul with FGM	0.577	0.673	0.580	0.623
	Guard2Vul with PGD	0.582	0.674	0.579	0.623
	Guard2Vul with DGDs	0.590	0.677	0.609	0.641
	Guard2Vul with DGD	0.606	0.679	0.643	0.661
Reveal	Guard2Vul w/o DGD	0.504	0.380	0.389	0.384
	Guard2Vul with FGM	0.526	0.442	0.412	0.426
	Guard2Vul with PGD	0.529	0.453	0.410	0.430
	Guard2Vul with DGDs	0.553	0.435	0.447	0.441
	Guard2Vul with DGD	0.560	0.499	0.438	0.467

four notations to represent four models, namely Guard2Vul w/o DGD, Guard2Vul with FGM, Guard2Vul with PGD, Guard2Vul with DGDs and Guard2Vul with DGD, whose detailed descriptions are given as follows:

- Guard2Vul w/o DGD means that using our framework without adversarial training models.
- Guard2Vul with FGM means that using our framework with fgm adversarial training models.
- Guard2Vul with PGD means that using our framework with pgd adversarial training models.
- Guard2Vul with DGDs means that using our framework with DGDs adversarial training models with the same *dropout*.
- Guard2Vul with DGD means that using our framework with DGDs adversarial training models.

The experimental results are shown in Table 5. The results on four datasets show that the DGDs and DGD models can achieve significantly better results than other adversarial training models. The model seems advantageous for the C/C++ dataset (MSR). This indicates that different programming languages affect the model differently by generating adversarial examples.

Moreover, we also observed that when the *dropout* rates of the two layers in DGD are different, better F1-measure will be obtained on the four datasets than when the *dropout* rates of the two layers are the same. This may be because different *dropout* rates can introduce more randomness, thereby increasing the robustness of the model. When the *dropout* rates of the two layers are different, each layer's neurons have a different probability of being

retained, which can reduce the risk of overfitting and improve the model's generalization ability. Different *dropout* rates can also help the model learn more features, thereby improving performance.

Compared with approaches without data augmentation, Guard2Vul can improve F1-measure by 9.7%, 40.3%, 13.1%, and 21.5% in the four datasets. Compared with other data augmentation models, DGD has also achieved significant advantages. It is proved that the proposed DGD model can improve robustness and performance by adding disturbances to the model with the generated augmentation examples.

Summary for RQ3: Using DGD can effectively improve the performance of Guard2Vul by adding diverse adversarial samples to the original training data.

6 Discussion

6.1 What is the Influence of Hyper-parameters on the Performance of Guard2Vul?

We aim to investigate the impact of model hyper-parameters on the performance of the current task's Guard2Vul model. DGD with different hyper-parameters will generate various adversarial examples, which may affect the performance. For this purpose, we conducted a comparative test on the different settings of K and *dropout*. We selected the most commonly used F1-measure in software vulnerability detection as an evaluation indicator.

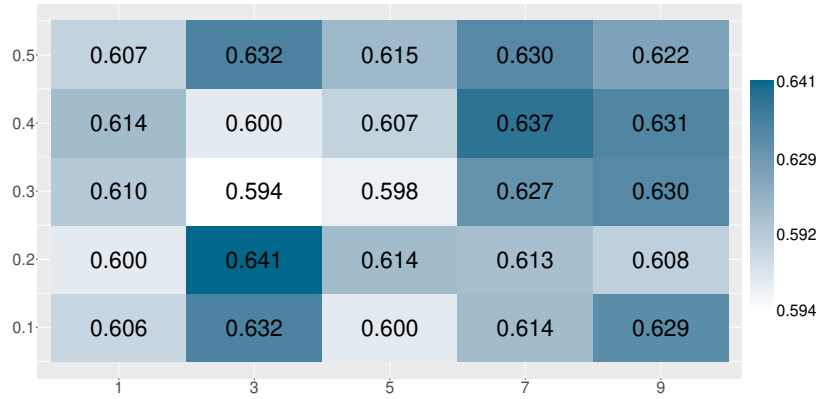


Fig. 2 Experimental results of K and *dropout* in Devign

We have discussed the K value in DGD and now set the two layers of *dropout* to the same rate. We observed the results of F1-measure on the Devign dataset, which was most commonly used in previous studies. The Fig. 2 shows that the model achieves the best results when the rate of K is 3 and the

dropout is set to 0.2. This may involve performing three gradient updates in each iteration, which can increase the disturbance level of the adversarial sample and make it more challenging. A larger K rate can increase the difficulty of adversarial training, forcing the model to learn the features of adversarial samples better. When the *dropout* takes a lower rate, the model retains more neurons, which may lead to overfitting. When the *dropout* takes a higher rate, the model will randomly discard more neurons, which may lead to underfitting. At a *dropout* rate of 0.2, Guard2Vul can moderately reduce overfitting risk while maintaining sufficient neurons to participate in training, thus achieving the best performance.

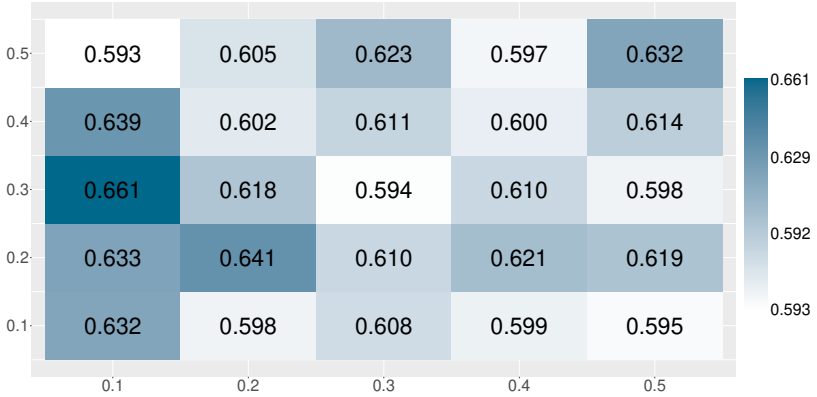


Fig. 3 Experimental results of dropouts in Devign

We use K as 3 in the above discussion to analyze the settings for different *dropout* rates. The Fig. 3 shows that when the *dropout* rates of the two layers in DGD are different, better F1-measure will be obtained on the four datasets than when the *dropout* rates of the two layers are the same. This may be because the Guard2Vul can better balance the issues of overfitting and underfitting during the training process. When the *dropout* of the first layer is set to 0.1, the model retains more neurons, which helps to extract richer features. When the *dropout* of the second layer is set to 0.3, the model will randomly discard a portion of neurons, which helps to reduce overfitting. This different *dropout* rate can control the complexity and generalization ability of the model at different levels, making the model better adapt to training and testing data.

6.2 Why does Guard2Vul Work

We have identified the following two advantages of Guard2Vul, which can explain its effectiveness in code vulnerability detection.

(1) The ability to capture information from complex diagrams of code structure. The proposed AesNet improved GraphSAGE model method helps to learn the structural information of code deep structure graphs. ResNet

is a deep convolutional neural network that can extract deeper feature representations by stacking multiple Residual blocks. In addition, the channel attention mechanism adds an attention module to each residual block, which adaptively adjusts the information in the feature map by learning the weights of each channel. The GNN model usually only contains one or a few loop layers, and the depth is not as deep as ResNet. Therefore, GraphSAGE with AesNet can better capture complex features in graph data. As shown in Table 4, the AesNet module has significantly improved to Guard2Vul. The addition of the AesNet module in Guard2Vul resulted significant performance improvement in vulnerability detection, as shown in Table 3, on four datasets, the average F1-measure was 52.4%.

(2) The ability to handle multiple types of code. We have designed a DGD module for model data augmentation. It can perturb the input data slightly in each iteration, making it difficult for the model to be disturbed by adversarial samples. Dropout regularization can reduce the overfitting risk of the model and help it better adapt to different input data. Moreover, introducing dropout regularization into neural networks can reduce the model's input dependence and improve robustness. DGD introduces adversarial samples to force the model to pay more attention to the boundary and abnormal situations of the input space during the learning process, thereby improving the model's generalization ability. In contrast, the original model may be more prone to overfitting the training data and have weaker generalization ability. The Table 5 shows that the DGD module has significant advantages over all other adversarial training methods, with an average improvement of 10.8% in F1-measure.

We also conducted experiments on an open-source Java dataset to verify the generalization and universality of the model. As shown in Table 3, our method is also at least 13.4% higher than other baselines in F1-measure on the SARD dataset.

6.3 Threats to Validity

The primary external threat to our approach is its generalizability. Although the SARD dataset we collected is synthetic and lacks vulnerability data from the real world, we ensured a fair comparison with other methods by utilizing the real-world dataset of Fan et al. [43]. Our experiment is focused on vulnerabilities in C/C++ and Java programs. It's important to note that the results may not be reproducible when applied to more complex vulnerabilities or languages, such as Python. However, our method has the potential to be extended to other languages and vulnerabilities.

Internal validity in our experiment relates to two factors. The first is related to parameter settings, as different parameters can lead to vastly different results. We followed the previous method to mitigate this and used the same parameters while setting the learning rate to 0.001. The second threat is related to the implementation of the baseline. To compare with existing DL-based vulnerability detection methods, we re-implemented TokenCNN based on the

steps in the original text since it is a closed source code. We made every effort to establish and adjust development parameters on our dataset. It's important to note that our model is only built under the PyTorch framework because it has higher programmability than Tensorflow.

Construct validity in our experiment relates to three factors. The first factor is the word embedding model we use. Our method uses CodeBERT as a pre-training model to embed code tokens. While other models may produce better results, our current findings are positive. In the future, we plan to explore other options for further optimization. Additionally, we currently only consider code structural features generated by AST. However, other technologies, such as Program Dependency Graphs, have also proven effective. Moreover, different GNN models may have varying degrees of impact. GraphSAGE has been shown to outperform GCN in many tasks, and we are making every effort to adjust the model parameters to achieve the best performance.

7 Conclusion

We propose Guard2Vul, a software vulnerability detection approach based on GNN. Guard2Vul detects vulnerable functions using the residual connection network to improve GCN. This allows us to retain high-level program semantics to learn implicit vulnerability patterns. We construct the DGD adversarial training approach to enhance the data and improve the robustness of Guard2Vul. The experimental results show the effectiveness of our approach by comparing it with four state-of-the-art advanced DL-based approaches.

In the future, we want to detect specific types of vulnerabilities separately, such as buffer error vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399). Furthermore, we intend to integrate certain interpretable models to enhance the understandability of the results.

References

[1] Massacci, F., Ngo, C.N., Nie, J., Venturi, D., Williams, J.: The seconomics (security-economics) vulnerabilities of decentralized autonomous organizations (transcript of discussion). In: Security Protocols XXV: 25th International Workshop, Cambridge, UK, March 20–22, 2017, pp. 180–185 (2017)

[2] Christensen, K.K., Liebetrau, T.: A new role for 'the public'? exploring cyber security controversies in the case of wannacry. Intelligence and National Security **34**(3), 395–408 (2019)

[3] Özkan, S.: Cve details: The ultimate security vulnerability database. Verfügbar unter: https://www.cvedetails.com/. **25** (2019)

[4] Steenhoek, B., Rahman, M.M., Jiles, R., Le, W.: An empirical study of deep learning models for vulnerability detection. arXiv preprint

arXiv:2212.08109 (2022)

[5] Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. Proceedings of the IEEE **108**(10), 1825–1848 (2020)

[6] Hanif, H., Nasir, M.H.N.M., Ab Razak, M.F., Firdaus, A., Anuar, N.B.: The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. Journal of Network and Computer Applications **179**, 103009 (2021)

[7] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S&P'06), pp. 6–263 (2006). IEEE

[8] Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet. IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2021)

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

[10] Li, X., Wang, L., Xin, Y., Yang, Y., Tang, Q., Chen, Y.: Automated software vulnerability detection based on hybrid neural network. Applied Sciences **11**(7), 3201 (2021)

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

[12] Groh, F., Ruppert, L., Wieschollek, P., Lensch, H.P.: Ggnn: Graph-based gpu nearest neighbor search. IEEE Transactions on Big Data **9**(1), 267–279 (2022)

[13] Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)

[14] Huang, B., Carley, K.M.: Residual or gate? towards deeper graph neural networks for inductive graph representation learning

[15] Zhao, T., Zhang, X., Wang, S.: Graphsmote: Imbalanced node classification on graphs with graph neural networks. In: Proceedings of the 14th

ACM International Conference on Web Search and Data Mining, pp. 833–841 (2021)

[16] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)

[17] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

[18] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing **19**(4), 2244–2258 (2021)

[19] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: VulDeePecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)

[20] Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: A survey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security, pp. 152–156 (2012). IEEE

[21] Shahriar, H., Zulkernine, M.: Mitigating program security vulnerabilities: Approaches and challenges. ACM Computing Surveys (CSUR) **44**(3), 1–46 (2012)

[22] Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR) **50**(4), 1–36 (2017)

[23] Zagane, M., Abdi, M.K., Alenezi, M.: Deep learning for software vulnerabilities detection using code metrics. IEEE Access **8**, 74562–74570 (2020)

[24] Medeiros, N., Ivaki, N., Costa, P., Vieira, M.: Vulnerable code detection using software metrics and machine learning. IEEE Access **8**, 219174–219198 (2020)

[25] Jiarpakdee, J., Tantithamthavorn, C., Treude, C.: Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 92–103 (2018). IEEE Computer Society

[26] Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the

Foundations of Software Engineering, pp. 292–303 (2021)

[27] Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H.: Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Transactions on Dependable and Secure Computing **19**(4), 2821–2837 (2021)

[28] Xu, A., Dai, T., Chen, H., Ming, Z., Li, W.: Vulnerability detection for source code using contextual lstm. In: 2018 5th International Conference on Systems and Informatics (ICSAI), pp. 1225–1230 (2018). IEEE

[29] Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Information and Software Technology **136**, 106576 (2021)

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

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

[32] Tang, Z., Hu, Q., Hu, Y., Kuang, W., Chen, J.: Sevuldet: A semantics-enhanced learnable vulnerability detector. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 150–162 (2022). IEEE

[33] Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., Lempitsky, V.: Domain-adversarial training of neural networks. Domain Adaptation in Computer Vision Applications, 189 (2017)

[34] Abdelaty, M., Scott-Hayward, S., Doriguzzi-Corin, R., Siracusa, D.: Gadot: Gan-based adversarial training for robust ddos attack detection. In: 2021 IEEE Conference on Communications and Network Security (CNS), pp. 119–127 (2021). IEEE

[35] Hasegawa, K., Hidano, S., Nozawa, K., Kiyomoto, S., Togawa, N.: R-htdetector: Robust hardware-trojan detection based on adversarial training. IEEE Transactions on Computers **72**(2), 333–345 (2022)

[36] Pelayo, L., Dick, S.: Applying novel resampling strategies to software defect prediction. In: NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society, pp. 69–72 (2007). IEEE

[37] Şahin, C.B.: Dcw-rnn: Improving class level metrics for software vulnerability detection using artificial immune system with clock-work recurrent neural network. In: 2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), pp. 1–8 (2021). IEEE

[38] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., *et al.*: Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1536–1547. Association for Computational Linguistics, Online (2020)

[39] Su, J., Cao, J., Liu, W., Ou, Y.: Whitening sentence representations for better semantics and faster retrieval. arXiv preprint arXiv:2103.15316 (2021)

[40] Son, J., Hahn, J., Seo, H., Han, Y.-S.: Boosting code summarization by embedding code structures. In: Proceedings of the 29th International Conference on Computational Linguistics, pp. 5966–5977 (2022)

[41] Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z.: Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security **16**, 1943–1958 (2021)

[42] Wager, S., Wang, S., Liang, P.S.: Dropout training as adaptive regularization. Advances in neural information processing systems **26** (2013)

[43] Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 508–512. Association for Computing Machinery, New York, NY, USA (2020)

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

[45] Ghaffarian, S.M., Shahriari, H.R.: Neural software vulnerability analysis using rich intermediate graph representations of programs. Information Sciences **553**, 189–207 (2021)

[46] Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(3), 1–33 (2021)

[47] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C.: Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks. In:

Proceedings of the 44th International Conference on Software Engineering, pp. 1456–1468 (2022)

[48] Cuzick, J.: A wilcoxon-type test for trend. Statistics in medicine **4**(1), 87–90 (1985)