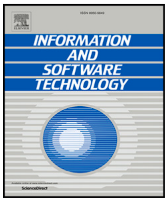
Information and Software Technology 136 (2021) 106576

Contents lists available at ScienceDirect

Information and Software Technology

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



BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection

Sicong Cao [a], Xiaobing Sun [a,b,*], Lili Bo [a], Ying Wei [a], Bin Li [a]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Vulnerability detection
Bidirectional Graph Neural-Network
Code representation

ABSTRACT

Context: Previous studies have shown that existing deep learning-based approaches can significantly improve the performance of vulnerability detection. They represent code in various forms and mine vulnerability features with deep learning models. However, the differences of code representation forms and deep learning models make various approaches still have some limitations. In practice, their false-positive rate (FPR) and false-negative rate (FNR) are still high.
Objective: To address the limitations of existing deep learning-based vulnerability detection approaches, we propose *BGNN4VD* (Bidirectional Graph Neural Network for Vulnerability Detection), a vulnerability detection approach by constructing a Bidirectional Graph Neural-Network (BGNN).
Method: In Phase 1, we extract the syntax and semantic information of source code through abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG). Then in Phase 2, we use vectorized source code as input to Bidirectional Graph Neural-Network (BGNN). In Phase 3, we learn the different features between vulnerable code and non-vulnerable code by introducing backward edges on the basis of traditional Graph Neural-Network (GNN). Finally in Phase 4, a Convolutional Neural-Network (CNN) is used to further extract features and detect vulnerabilities through a classifier.
Results: We evaluate *BGNN4VD* on four popular C/C++ projects from NVD and GitHub, and compare it with four state-of-the-art (*Flawfinder*, *RATS*, *SySeVR*, and *VUDDY*) vulnerab ility detection approaches. Experiment results show that, when compared these baselines, *BGNN4VD* achieves 4.9%, 11.0%, and 8.4% improvement in F1-measure, accuracy and precision, respectively.
Conclusion: The proposed *BGNN4VD* achieves a higher precision and accuracy than the state-of-the-art methods. In addition, when applied on the latest vulnerabilities reported by CVE, *BGNN4VD* can still achieve a precision at 45.1%, which demonstrates the feasibility of *BGNN4VD* in practical application.

1. Introduction

Software vulnerabilities can be exploited by hackers to conduct cyber attacks and cause enormous losses [1,2]. Take the DAO[1] (Decentralized Autonomous Organization) event for example, the hackers exploit the reentrancy bug of *The DAO* contract to steal 3.6 million Ether (Cryptocurrency of Ethereum). In recent years, the number of software vulnerabilities has been increasing rapidly. The vulnerabilities reported by Common Vulnerabilities and Exposures (CVE[2]) show that, before 2016, the number of software vulnerabilities was around 6000 per year. Since 2017, the number of software vulnerabilities has risen sharply to around 15,000. In spite of the effort made by academia and industry to improve software quality, vulnerabilities are still a difficult problem to solve.

Some empirical results show that there is correlation between vulnerabilities and bugs, but it is weak [3]. Conceptually, vulnerabilities are different from bugs. Vulnerabilities represent abusive functionality, but bugs represent wrong or insufficient functionality. Vulnerability detection is a task that tries to find software vulnerabilities by auditing the software code or analyzing the execution process of software [4–12]. It requires understanding and reasoning about program semantics. Therefore, approaches used for detecting bugs cannot be used for vulnerability detection directly. At present, vulnerability detection approaches are mainly based on source code or bytecode [13–19].

* Corresponding author at: School of Information Engineering, Yangzhou University, Yangzhou, China.
E-mail addresses: xbsun@yzu.edu.cn (X. Sun), lb@yzu.edu.cn (B. Li).

[1] http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[2] http://cve.miter.org/.

https://doi.org/10.1016/j.infsof.2021.106576
Received 31 October 2020; Received in revised form 13 March 2021; Accepted 15 March 2021
Available online 20 March 2021

These approaches can be divided into two types: metric-based (or rule-based) and pattern-based approaches. Metric-based approaches, which treat vulnerabilities as a special kind of bug, are inspired by bug detection [20–26]. They are based on some metrics (e.g., code churn) that are often defined by human experts. These approaches have attained a limited success because they cannot avoid the intense labor of human experts on feature extraction and it is impractical to characterize all vulnerabilities with hand-crafted features. Rather than relying on human experts, pattern-based approaches leverage machine learning techniques to automatically learn vulnerability patterns. Deep learning (DL), with the great power to deal with large volume of software code and vulnerability data, is recently introduced in vulnerability detection [27–32]. These vulnerability detection approaches have a common procedure: First, they transform the code into the defined representation forms (e.g., tokens) [4,6] which are suitable for learning the syntax and semantic information to extract the features related to vulnerabilities. Then, the features are vectorized by encoding models to enhance the representation ability of features. Finally, the labels for detection results are calculated by a deep learning model. However, different forms of code representation in these approaches only retain partial information (either syntax or semantics), which cannot cover various vulnerabilities and limit the effect of detection.

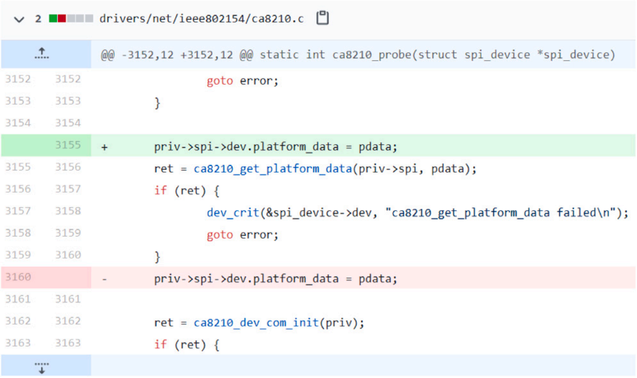
Furthermore, since source code is more logical and structural than natural languages, neural networks used for training need to be able to handle the non-sequential feature representation. Even though many deep learning models are used in vulnerability detection, there are still some limitations: Firstly, Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) are sequence-based model and cannot handle the non-sequential feature of graphs in the code representation. Secondly, Graph Neural Networks (GNN) can only handle single graph without attributes and a part of them (e.g., Gate Graph Neural Network (GGNN) [33]) cannot accommodate the information of incoming edges and outgoing edges into a node at the same time.

To cope with above two challenges, in this paper, we propose a bidirectional graph neural network-based approach for vulnerability detection. We consider multiple code representations which can accommodate sufficient syntax and semantic information of vulnerabilities. Firstly, we transform source code into different graphs which can reflect syntax and semantic features (e.g., Control Flow Graph). Then we vectorize these nodes in graphs as iput to our Bidirectional Graph Neural Network (BGNN) for training. Finally, we extract features through a convolutional neural network and use a classifier to detect vulnerabilities.

The main contributions of this paper are as follows:

- We propose *BGNN4VD* (Bidirectional Graph Neural Network for Vulnerability Detection), a vulnerability detection approach by constructing a Bidirectional Graph Neural-Network (BGNN). We extract the features from BGNN followed by a CNN to identify whether the program function is vulnerable.
- We manually collected 2149 vulnerabilities and 3867 vulnerable functions from four popular C/C++ open source projects (i.e., Linux Kernel, FFmpeg, Wireshark and Libav) to construct our dataset. Compared with the state-of-the-art detectors, *BGNN4VD* achieves 4.9%, 11.0%, and 8.4% improvement in F1-measure, accuracy and precision, respectively.

The rest of the paper is organized as follows. In Section 2, we describe the motivation of constructing BGNN to implement our work and the technical background of *BGNN4VD*. Section 3 explains the detailed design of our approach. Section 4 introduces the implementation of our approach. Section 5 is the systematic experiment and analysis of the results. In Section 6, we analyze the threats to validity of our approaches and experiments. Section 7 is the related work. Section 8 concludes the paper and makes an explanation of the future work.



```
@@ -3152,12 +3152,12 @@ static int ca8210_probe(struct spi_device *spi_device)
3152  3152              goto error;
3153  3153      }
3154  3154
      3155  +   priv->spi->dev.platform_data = pdata;
3155  3156      ret = ca8210_get_platform_data(priv->spi, pdata);
3156  3157      if (ret) {
3157  3158              dev_crit(&spi_device->dev, "ca8210_get_platform_data failed\n");
3158  3159              goto error;
3159  3160      }
3160        -   priv->spi->dev.platform_data = pdata;
3161  3161
3162  3162      ret = ca8210_dev_com_init(priv);
3163  3163      if (ret) {
```

Fig. 1. Motivating example.

2. Motivation and background

In this section, we describe the motivation of constructing BGNN to implement our work and the technical background of *BGNN4VD*.

2.1. Motivation

We use a real-world vulnerability occurred in Linux Kernel to illustrate the reason why we construct BGNN.

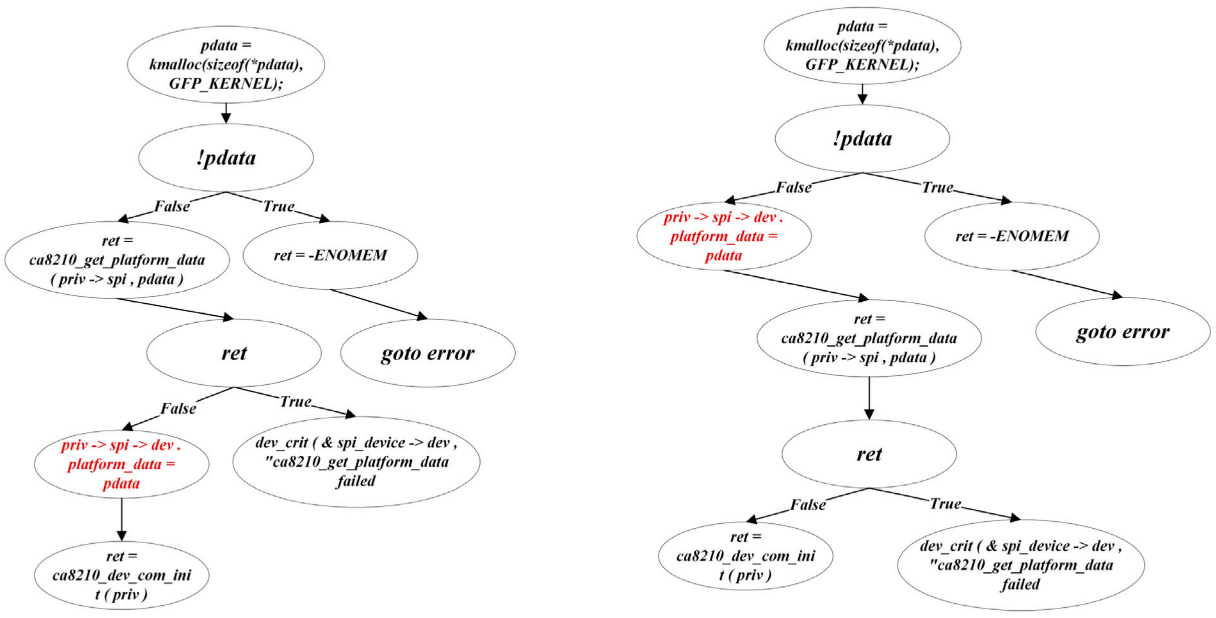
CVE-2019-19075[3] is a memory leak vulnerability which allows attackers to cause a denial of service. Its *diff* file is shown in Fig. 1. The vulnerability occurred in function *ca8210_probe* due to missing release of memory after effective lifetime: the failure of `ca8210_get_platform_data` may lead `pdata` cannot be released. It allows attackers to cause a denial of service (memory consumption) by triggering `ca8210_get_platform_data` failures. To fix this vulnerability, the allocated `pdata` needs to be assigned to `spi_device→dev.platform_data` before calling `ca8210_get_platform_data` (line 3155).

From this example, we can obtain the following two observations:

Observation 1. This vulnerability cannot be detected by using single code graph such as Abstract Syntax Tree (AST) because it can only reflect the syntax problems. Also, the flow direction of `pdata` in vulnerable function and patched function is completely consistent. But their control flows are different. As shown in Fig. 2, we can see that in vulnerable function, the vulnerable statement `priv→spi→dev.platform_data = pdata` executes after `if (ret)` {…}, but in patched function, the vulnerable statement executes before `ret = ca8210_get_platform_data(priv→spi, pdata)`. Therefore, it is necessary to consider various code graphs which reflect syntax and semantic gaps between vulnerable statements and non-vulnerable statements.

Observation 2. The neural network used to train the detection model should be able to accommodate rich context information. As shown in Fig. 1, the core question of this vulnerability is whether the allocated `pdata` is assigned before calling `ret = ca8210_get_platform_data(priv→spi,pdata)`. An ideal detection model should accommodate the information about the statements that appear before and after the vulnerable statements. Compared with widely used Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), Graph Neural Network (GNN) can effectively handle the non-sequential features of graph-based data. However, since the communication between nodes in the existing GNNs is mostly unidirectional, the context information will be partially masked. Hence, how to aggregate information of different edges from various graphs and make

[3] https://nvd.nist.gov/vuln/detail/CVE-2019-19075.


pdata = kmalloc(sizeof(*pdata), GFP_KERNEL);
!pdata
False
True
ret = ca8210_get_platform_data (priv -> spi , pdata)
ret = -ENOMEM
ret
goto error
False
True
priv -> spi -> dev . platform_data = pdata
dev_crit (& spi_device -> dev , "ca8210_get_platform_data failed
ret = ca8210_dev_com_ini t (priv)
pdata = kmalloc(sizeof(*pdata), GFP_KERNEL);
!pdata
False
True
priv -> spi -> dev . platform_data = pdata
ret = -ENOMEM
ret = ca8210_get_platform_data (priv -> spi , pdata)
goto error
ret
False
True
ret = ca8210_dev_com_ini t (priv)
dev_crit (& spi_device -> dev , "ca8210_get_platform_data failed

(a) Control flow of vulnerable function

(b) Control flow of patched function

Fig. 2. Different control flows of motivating example.

a distinction between forward information and backward information are needed to be tackled.

From Observation 1, we combine AST, CFG and DFG, and propose Code Composite Graph (CCG) for capturing various syntax and semantic information of source code. From Observation 2, we propose Bidirectional Graph Neural Network (BGNN) that aggregates the information from all graphs at each step during the iterative learning process and pass information to neighbors through both forward edges and backward edges. With bidirectional edges, *BGNN4VD* can accommodate more information about the statements that appear before and after the vulnerable statements.

2.2. Background

Code representation. Code representation is often used in static analysis and other tasks that need to infer semantics relationships between statements. A good code representation can extract features from source code considering multiple aspects and express the vulnerability characteristics comprehensively. It is a pre-processing step that builds a "bridge" between a program and its vector representation, which is the actual input to deep learning-based model for vulnerability detection. Commonly, code representation techniques exploit graph structure to represent the syntax and semantics characteristics. There are Abstract Syntax Trees (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG). AST [7,34] is an intermediate representation in the compilation process, and the syntax information of the function is stored through the tree. CFG and DFG are abstract representations of a program. CFG [35,36] represents the possible flow of all basic block execution within a process in the form of a graph and also reflects the real-time execution of a process. DFG graphically expresses the logical functions of a program and depicts the process of data flow in the program from the perspective of data transmission and processing. The definitions of AST, CFG and DFG are as follows:

Definition 1 (*Abstract Syntax Tree (AST)*)**.** A graph $G_A^i = (V_A^i, E_A^i)$ where i is the ist function f_i in a program $P = f_1, f_2, \dots, f_\eta$, V_A^i is a set of leaf nodes, and E_A^i is a set of directed edges with each edge links a pair of parent–child nodes.

Definition 2 (*Control Flow Graph (CFG)*)**.** A graph $G_C^i = (V_C^i, E_C^i)$ where V_C^i is a set of nodes, and E_C^i is a set of directed edges with each edge indicate the flow of control.

Definition 3 (*Data Flow Graph (DFG)*)**.** A graph $G_D^i = (V_D^i, E_D^i)$ where V_C^i is a set of nodes, and E_C^i is a set of directed edges with each edge indicate the access or modification of variables.

From the definitions above, we can see that AST, CFG and DFG contain abundant syntax and semantic information of programs. In this paper, we combine AST, CFG and DFG as input to construct a bidirectional graph neural network, so as to extract features that can represent the code more precisely.

Gated Graph Neural Networks. Our work builds on Gated Graph Neural Networks (GGNN) [33], which introduces a learnable parameter W for different types of edges, so that we can handle various graphs that traditional GNN cannot. We summarize how GGNN works here.

For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, $\mathcal{V}$ is a set of nodes, $\mathcal{E} = (\mathcal{E}_1, \dots, \mathcal{E}_K)$ is a list of directed edge sets where K is the number of edge types and $\boldsymbol{X}$ are node annotations (i.e., node labels used as inputs). GGNN annotates each $v \in \mathcal{V}$ with a real-valued vector $\boldsymbol{x}^{(v)} \in \mathbb{R}^D$ representing the features of the node.

GGNN associates each node v with a state vector $\boldsymbol{h}^{(v)}$ initialized from the node initial representation $\boldsymbol{x}^{(v)}$ which represents the characteristic information of the node itself. The state vector of each node will be updated in each time step for iteration and becomes feature vector we used for classification after iterations. The sizes of the state vector and feature vector are typically the same, but we can use larger state vectors through padding of node features. To propagate information throughout the whole graph, message contained by type k are sent from each v to its neighbors, where each type of message $\boldsymbol{m}_k^{(v)}$ is computed from its current state vector $\boldsymbol{h}^{(v)}$ through an arbitrary function. By computing messages for all graph edges at the same time, the state vectors of all nodes can be updated at the same time. In particular, a new state for a node v is computed by aggregating all incoming messages as $\widetilde{\boldsymbol{m}}^{(v)} = g(\boldsymbol{m}_k^{(v)})$, in which there is an edge of type k from u to v. Here, g is an aggregation function. Given the aggregated message $\widetilde{\boldsymbol{m}}^{(v)}$ and the current state vector $\boldsymbol{h}^{(v)}$ of node v, the state of the next time step $\boldsymbol{h}'^{(v)}$ is computed as $\boldsymbol{h}'^{(v)} = GRU(\widetilde{\boldsymbol{m}}^{(v)}, \boldsymbol{h}^{(v)})$, where GRU is the recurrent cell function of gated recurrent unit (GRU). The dynamics

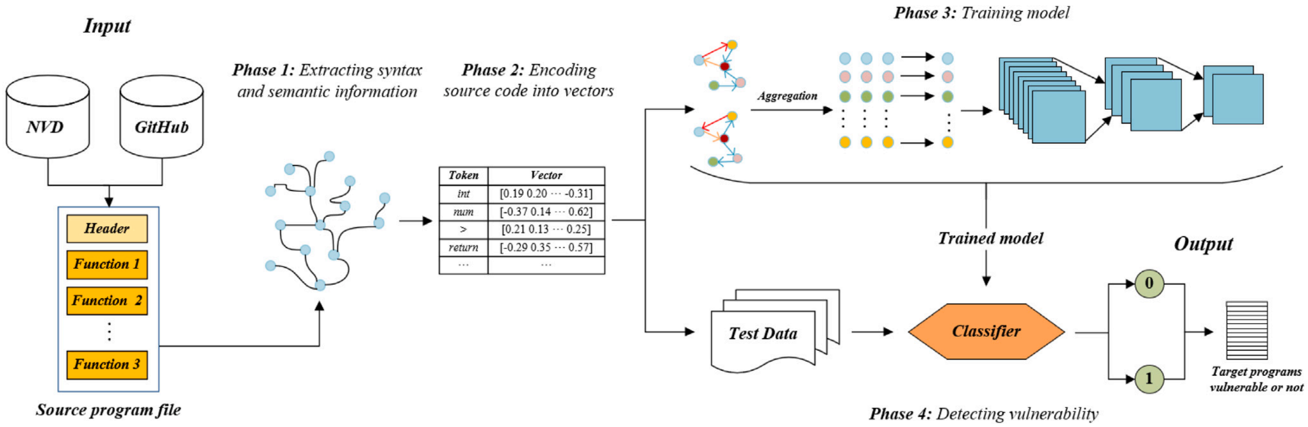
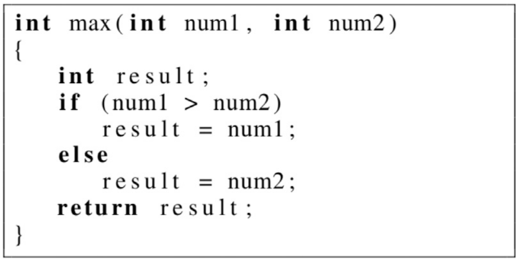


Fig. 3. The process of *BGNN4VD*.



```
int max(int num1, int num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Fig. 4. Code sample.

defined by the above equations are repeated for a fixed number of time steps T. Then, the state vectors from the last time step is the final node representations and will be used as input for classification.

3. Our approach

Fig. 3 shows the process of *BGNN4VD*. It includes four phases: (1) extracting syntax and semantic information of source code, (2) encoding source code into vectors as input to graph neural networks, (3) training the neural network model, and (4) detecting vulnerabilities.

BGNN4VD takes the vulnerable files as input. In Phase 1, the vulnerable files are transformed into graphs that preserve the semantic relationships between the elements of programs (e.g., data dependency and control dependency). In Phase 2, the nodes in graphs are encoded into vectors. In Phase 3, BGNN model is trained and the feature vectors of nodes which reflect vulnerability patterns are obtained. Finally, we use the classifier in Phase 4 to detect whether the input file is vulnerable or not at the *function level*.

3.1. Extracting syntax and semantic information

The syntax information can be represented by AST and the semantic information can be represented by CFG and DFG. We combine AST, CFG and DFG in our approach and propose Code Composite Graph (CCG). CCG consists of a set of nodes of AST and different types of edges. Statements and predicate nodes are connected by CFG and DFG. Thus, the types of edges reflect the syntax and semantic relationships between different nodes. The definition of CCG is as follows:

Definition 4 (*Code Composite Graph (CCG)*)**.** For a program $P = \{f_1, f_2, \ldots, f_\eta\}$, the code composite graph of function f_i is denoted by $G = (V, E)$, where $V = V_A^i$ represents that nodes V of the code composite graph are composed of leaf nodes V_A^i in AST, and $E = E_A^i \cup E_C^i \cup E_D^i$ represents E is a set of mixed edges with AST, CFG and DFG.

Take the function *max* in Fig. 4 for example, its AST, CFG and DFG are shown in Fig. 5. The CCG is shown in Fig. 6. Note that the order in which statements are executed cannot be determined from three graphs, but the dependencies between statements and predicates are clearly visible. The blue arrows in CCG represent child-parent relations in AST. It reflects the syntax structure of the given code. The red and orange arrows in CCG represent control dependence in CFG and data dependence in DFG, respectively. They reflect the semantic information of the given code.

3.2. Encoding source code into vectors

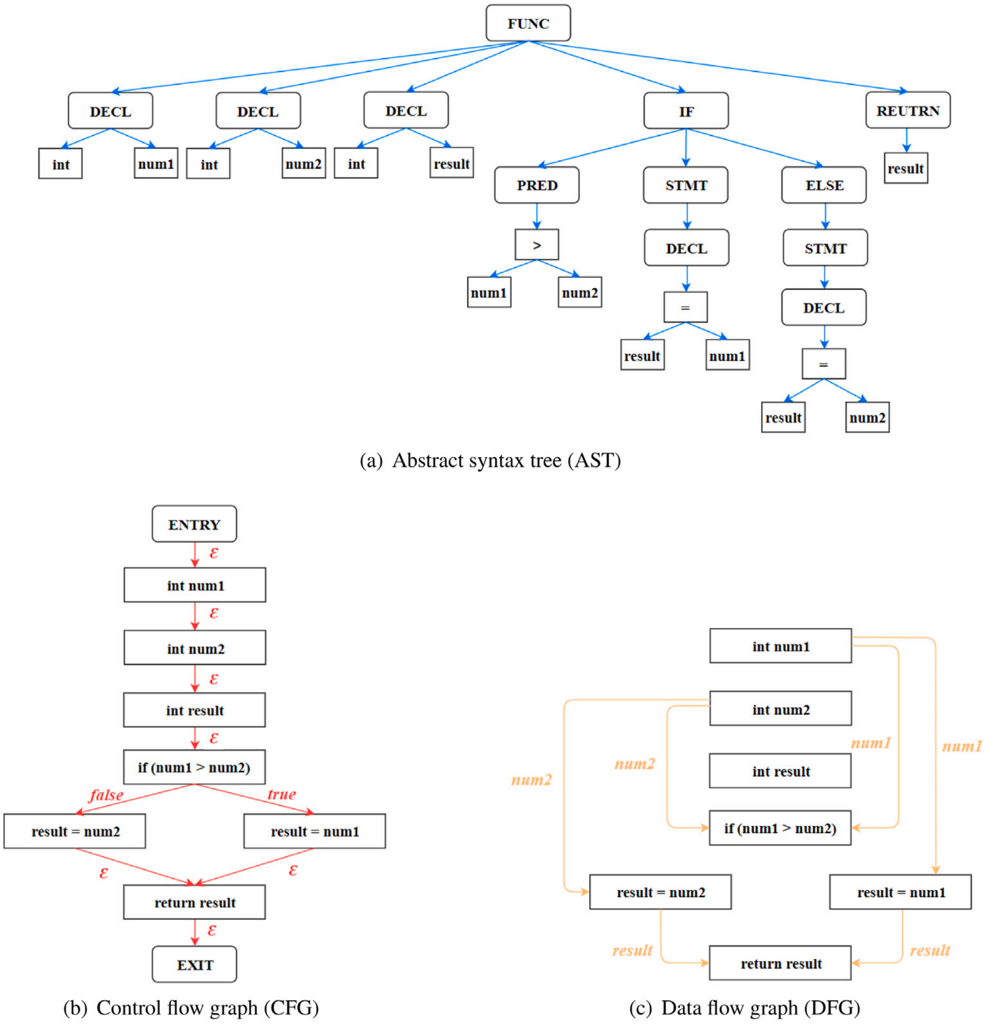
Since neural networks that used for training the detection model take vectors as input, we need to represent programs as vectors for vulnerability detection. Transforming a program into vectors directly may lose its semantic information. In this paper, we firstly represent the source code as CCG, which can preserve the semantic structure of code. Then, we transform the nodes of CCG into vectors which can be used by deep learning. We use the word embedding model Word2vec[4] to transform the textual representation of each nodes into vectors. In order to further reflect the abstract semantic relationships between words, we consider the type of each node and add it to the transformed vector to compute the initial node state. Concretely, we first extract the code representation of a node and its type through traversing the AST of a function f_i. For example, in `result` = `num1`, `result` and `num1` are code representations and both of their types are `Identifier`. Then, we encode all of the node tokens to achieve code representations by word2vec and encode node types to achieve type representations by label encoding. Finally, for each node in the graph, we concatenate the node representation with the label representation to obtain the initial representation.

3.3. Training neural network model

Having obtaining the initial node representations that can be used as input to the deep learning model, we use BGNN to train our detection model for learning aggregation information because it is suitable for non-sequential outputs and semantic learning.

We firstly initialize each state vector $\boldsymbol{h}^{(v)}$ of node v from their node initial representation $\boldsymbol{x}^{(v)}$. Then, we copy $\boldsymbol{x}^{(v)}$ into the first dimension and pad with 0 to make the size of state vectors larger than that of the node initial representation. To propagate information throughout the graph, each type of message $\boldsymbol{m}_k^{(v)}$ is computed from its current state vector $\boldsymbol{h}^{(v)}$ through a linear layer. All the state vectors are updated at the

[4] http://radimrehurek.com/gensim/models/word2vec.html.



(a) Abstract syntax tree (AST)

(b) Control flow graph (CFG)

(c) Data flow graph (DFG)

Fig. 5. Graph representation of the code.

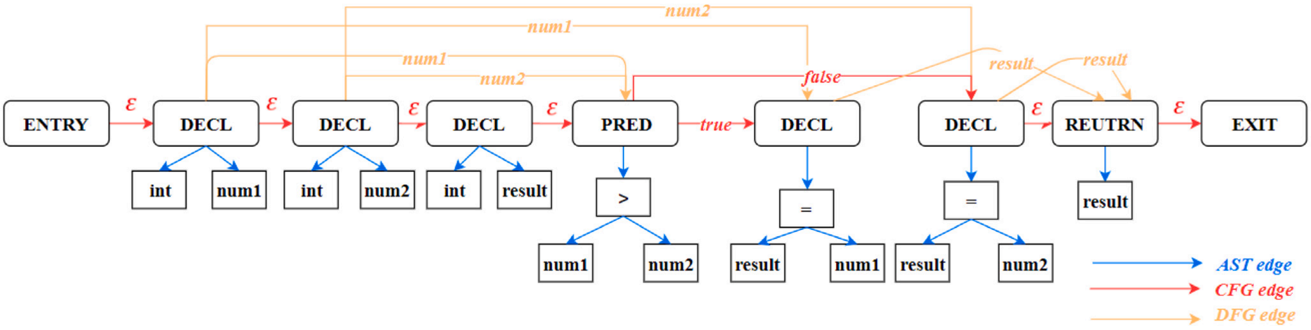


Fig. 6. Code composite graph (CCG).

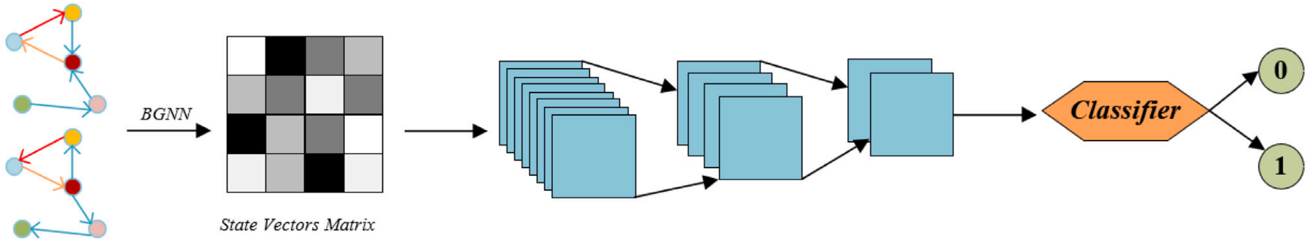


Fig. 7. Convolutional feature extraction for classification.

same time by aggregating all incoming messages through elementwise summation.

Since source code is more logical and structural than natural languages, vulnerable code snippets are usually relevant to their context. We believe that the neural network chosen for learning should be able to deal with the sequences forward and backward to obtain a deeper representation and achieve better robustness. In this paper, we construct a bidirectional graph neural-network (BGNN) by adding backward edges. For a pair of nodes $[u, v]$ with an edge k, we consider both of its forward edge $[u, v]$ and backward edge $[v, u]$. The original adjacency matrix constructed by forward edge k can be transposed to obtain corresponding adjacency symmetric matrix, which represents the backward edge k'. Therefore, the node information including code representation, node labels and their link information related to other neighbors used to capture the interdependencies between source code can be propagated forward and backward.

Given the aggregated message $\widetilde{\boldsymbol{m}}^{(v)}$ and the current state vector $\boldsymbol{h}^{(v)}$ of node v, the state of the next step $\boldsymbol{h}'^{(v)}$ is computed as $\boldsymbol{h}'^{(v)} = GRU(\widetilde{\boldsymbol{m}}^{(v)}, \boldsymbol{h}^{(v)})$. This computation iterate by T times. Then, we use the state vectors computed by the last time step as the final node representations.

3.4. Vulnerability detection

Node-level classification tasks [37] identify nodes which are not labeled by constructing an end-to-end multi-classification framework with a part of nodes with label. Different from *node-level* classification tasks, our vulnerability detection approach is a *graph-level* classification task that aims to predict the label of the CCG. If the function f_i is vulnerable, its CCG will be labeled as “1”. Otherwise, its CCG will be labeled as “0”. The predictive label $\widetilde{\boldsymbol{y}}_i$ of a function f_i is computed as follows:

$$\widetilde{\boldsymbol{y}}_i = Sigmoid(\sum_{v \in \mathcal{V}} i([\boldsymbol{H}^{(v)}, \boldsymbol{x}^{(v)}])), \tag{1}$$

where i is a neural network that uses the concatenation of $\boldsymbol{H}^{(v)}$ and $\boldsymbol{x}^{(v)}$, $\boldsymbol{H}^{(v)}$ represents the final node representation, and $\boldsymbol{x}^{(v)}$ is the initial node representation. Considering that the vulnerable code just accounts for the small proportion of the whole program, it is difficult to learn the semantic features of vulnerable code from the whole graph. Therefore, we use a CNN (including a convolutional layer, a Maxpooling layer and a fully-connected layer) to obtain the semantic features which are more related to vulnerabilities. The predictive label $\widetilde{\boldsymbol{y}}_i$ of a function f_i can be computed as follows:

$$\widetilde{\boldsymbol{y}}_i = Sigmoid(MLP(\sigma[\boldsymbol{H}^{(v)}, \boldsymbol{x}^{(v)}])), \tag{2}$$

where $\sigma[\boldsymbol{H}^{(v)}, \boldsymbol{x}^{(v)}]$ represents a soft attention mechanism that decides which nodes are relevant to the current graph-level task. $\sigma(\cdot)$ is short for $\sigma[\boldsymbol{H}^{(v)}, \boldsymbol{x}^{(v)}]$ and can be generalized by the following equation:

$$\sigma(\cdot) = MAXPOOL(Relu(CONV(\cdot))), \tag{3}$$

where $CONV(\cdot)$ represents 1-D convolution operation, $Relu(\cdot)$ represents the activation function aiming to optimize the neural network, and $MAXPOOL(\cdot)$ is used to compress the dimension of input features.

Fig. 7 shows our convolutional feature extraction process for classification. First, the subgraphs of CCG are input into BGNN. After the training phase in BGNN, the final state vectors matrix is obtained. Next, our convolutional feature extraction process consists of three steps: (1) The final state vectors matrix is put into our convolutional layer. Through 1-D convolution, the features of vulnerable code and non-vulnerable code are extracted. In order to increase the difference between the features of vulnerable code and non-vulnerable code, *Relu*, an activation function, is used to optimize the neural network. (2) The input features are compressed by a *Maxpooling layer*, so that the dimension of the feature vectors are reduced and the main features are extracted. (3) A fully-connected layer is used to connect all the features and send the final features to the classifier for vulnerability classification. Finally, we get a well-trained classifier which can predict the target functions whether vulnerable or not.

4. Experimental study

4.1. Research questions

To evaluate the performance of *BGNN4VD*, we design the following research questions:

RQ1: How effective is *BGNN4VD* when compared with state-of-the-art vulnerability detection approaches?

This question is designed to test the ability of *BGNN4VD* to detect vulnerabilities on a same dataset when compared with the state-of-the-art approaches. To answer this question, we compared *BGNN4VD* with the following state-of-the-art approaches: *Flawfinder,*[5] *RATS,*[6] *SySeVR* [28], and *VUDDY* [5]. Both *Flawfinder* and *RATS* are rule-based vulnerability detection tools. They match the vulnerability patterns in the target program by a built-in C/C++ common vulnerability database. *SySeVR* is a deep learning-based vulnerability detection approach that focuses on vulnerabilities that are related to library/API function calls. *VUDDY* is a similarity-based vulnerability detection approach which focuses on code clone. We used the default values for these models.

In this RQ, for two deep learning-based approaches *SySeVR* and *BGNN4VD*, we carried out experiments repeatedly to suppress the effects of randomness. For the others, we took the whole dataset as input to evaluate their effectiveness.

RQ2: Is the combination of both syntax and semantic edges beneficial to vulnerability detection?

Syntax and semantic information of source code are also considered in previous studies. Different from them, our study introduces the backward syntax and semantic edges for better learning effect of deep learning model. Thus, to explore whether the introduction of backward edges can improve the effect of vulnerability detection, we design this question.

To perform analysis, we added each combination of different edges into the model one by one in the setting with same parameters. The combinations of edges were divided into only syntax edge (i.e., AST), only semantic edges (i.e., CFG and DFG), three forward edges (i.e., AST, CFG and DFG), and six forward and backward edges (i.e., AST, CFG, DFG and their corresponding backward edges). We also ran our model with different configurations repeatedly.

RQ3: Can *BGNN4VD* achieve a substantially higher precision and accuracy than only using BGNN without a convolutional module?

As described in Section 3.4, a convolutional module is used to obtain features related to vulnerabilities. The precision and accuracy of vulnerability detection are affected by the extracted features. Therefore, this question is designed to evaluate the contribution of the convolutional module.

In this RQ, we compared *BGNN4VD* with BGNN that without the convolutional module under the same parameter configuration and carried out experiments repeatedly.

RQ4: How effective is *BGNN4VD* in detecting the latest vulnerabilities reported by CVE recently?

Different from the above questions based on the dataset we collected, the goal of this question is to test whether our approach can detect the latest vulnerabilities to show its applicability.

In this RQ, we applied *BGNN4VD* to detect vulnerabilities in several latest versions of four software projects (i.e., Linux Kernel, FFmpeg, Wireshark and Libav). We selected the vulnerabilities reported by CVE of each project since 2020, and manually labeled the vulnerable functions through their reports and *diff* files. For the latest vulnerabilities reported by CVE, we needed to add a step to filter those reports whose states of CVE entries were `RESERVED`, `DISPUTED` and `REJECT` before

[5] http://www.dwheeler.com/flawfinder.

[6] https://code.google.com/archive/p/rough-auditing-tool-for-security/.

Table 1
Details of vulnerability dataset.

Project	NVD(#)	VFs(#)	Non-VFs(#)	IR
Linux Kernel	1152	2484	60,013	24.16
FFmpeg	307	497	7897	15.89
Wireshark	541	709	20,447	28.84
Libav	149	177	3701	20.91
Total	2149	3867	92,058	23.81

Table 2
Evaluation metrics.

Metric	Formula	Description
Accuracy	$A = \frac{TP+TN}{TP+FP+TN+FN}$	The correctness of all detected.
Precision	$P = \frac{TP}{TP+FP}$	The correctness of detected vulnerable.
Recall	$R = \frac{TP}{TP+FN}$	The proportion of true-positive samples in the total samples that are vulnerable.
F1-measure	$F1 = \frac{2 \cdot P \cdot R}{P+R}$	The overall effectiveness considering both precision and false-negative rate.

adopting the same pre-processing approach as our dataset. Finally, we only retained vulnerabilities confirmed with fixing patches. This test set was composed of 113 vulnerable functions which came from 69 vulnerable files.

4.2. Dataset

We chose projects that satisfied the following criteria to construct our dataset. First, they are C/C++ open source projects since *Joern*[7] we used for extracting multiple graphs is only suitable for C/C++ projects. Second, they contain sufficient vulnerabilities with patches so that we can extract vulnerable code at the *function level* through their *diff* files.

According to above two criteria, we chose four popular open source projects (i.e., Linux Kernel, FFmpeg, Wireshark, and Libav). We collected vulnerability data of each project from two sources: NVD[8] and public Git repositories on GitHub.[9] NVD is a vulnerability database built upon and fully synchronized with the CVE list. In addition to a large amount of vulnerability data, it also provides enhanced information (e.g., vulnerability type, references to solutions) additionally for each record. GitHub provides a larger quantity and wider variety of code, which can help us supplement the vulnerability dataset. In total, we collected 2149 vulnerabilities from NVD and GitHub.

After obtaining the vulnerability dataset, we first cleaned dataset by removing all annotations because they were meaningless in our graph construction. Then, we manually extracted and labeled functions in vulnerable files of each vulnerability by analyzing *diff* files in patches. Concretely, it involved the following two steps: **Step 1:** We first split the vulnerable files as a set of functions and removed header files as well as externally defined global variables. **Step 2:** If there were one or more additions or deletions in a function's *diff* file, the label of this function was "1", and otherwise "0".

Labeling process was completed by the first and fourth authors independently. Every time 10% vulnerability data were labeled, the second author checked the consistency of results between them by Cohen's kappa coefficient. When the labeling results were inconsistent, we discussed these inconsistent cases to unify the labeling results. We continued to label the remaining data only when labeling results of both authors were identical (i.e., the Cohen's kappa coefficient was 1). As a result, 3867 functions were labeled as "1" and 92,058 functions were labeled as "0".

Table 1 presents the details of our dataset. Column 2 lists the number of NVD entries in each project. Column 3 lists the number of vulnerable functions (VFs) extracted in each project. Column 4 lists the number of non-vulnerable functions (Non-VFs) in each project. Column 5 lists the imbalanced rate (IR) between vulnerable functions and non-vulnerable functions.

Our data is available at https://github.com/SicongCao/BGNN4VD.

4.3. Experiment setup

We implemented *BGNN4VD* in Python with Tensorflow [38]. Since the dataset we collected is imbalanced (IR = 23.81 in Table 1), we randomly extracted a subset (including nearly 4000 functions) from Non-VFs to balance the dataset used for training. Moreover, in order to ensure the validity of the experimental results, we shuffled and divided the training/validation/test set into 8:1:1 before each experiment. After multiple times experiments (20 times in our implementation), the final evaluation result we adopted was the average of each measurement metric. Then, to check if the performance difference between a baseline model and *BGNN4VD* is statistically significant, we applied the Wilcoxon signed-rank test[10] (a statistical test method corresponding to paired sample T-test which can compare whether the difference between the average of two groups is significant or not in the case of small samples) at a 95% significance level on their performance of 20 times experiments. The hypothesis in each RQ is that there is no significant difference between *BGNN4VD* and the baseline models (or *BGNN4VD* with different configurations) in detection results, except RQ4. Our experiments were performed on a computer with a NVIDIA GeForce RTX 2060 GPU.

The trained hyper-parameters are: the number of time steps is 8; the size of the hidden layer is 200; batch size is 128; optimizer is ADAMAX [39]; learning rate is 0.00015; dropout is 0.2; momentum is 0.85, and patience is 50. We use n convolutional filters with shape $m \times D$, so each filter spans the full space of the state vectors. The filter size m determines the number of sequential tokens that are considered together and we find that a fairly large filter size of $m = 3$ worked best. Then, we use *Relu* as activation function and input into *Maxpooling* layer. Finally, we combine all the local features into global features through a fully-connected layer.

4.4. Evaluation metrics

Our approach is evaluated with the widely-used metrics which is summarized in Table 2. Table 2 shows the metrics, their computation formulas and the corresponding description. Among these metrics, TP indicates the number of vulnerable samples that are detected as vulnerable; FP indicates the number of samples that are not vulnerable but are detected as vulnerable; TN indicates the number of samples that are not vulnerable and are detected as not vulnerable; and FN indicates the number of vulnerable samples that are detected as not vulnerable.

Based on the four possible outputs of binary classification (i.e., TP, FP, TN, FN), we compute the following performance measures: Accuracy (A), Precision (P), Recall (R), and F1-measure (F1). Accuracy is a measure of true-positive and true-negative samples coverage, which shows the ability of predicting examples correctly. The Accuracy is the ratio of true-positive and true-negative samples to the total samples detected. Precision indicates how many of the samples predicted to be positive are truly positive samples. It is defined by the ratio of true-positive samples divided by the total samples that are detected as true. Recall is a measure of true-positive sample coverage. It shows the ability to predict positive examples. It is the ratio of true-positive

[7] https://joern.readthedocs.io/en/latest/import.html.

[8] https://nvd.nist.gov/.

[9] https://github.com/.

[10] Note, since we only conducted our experiments 20 times and the experimental data did not meet the normal distribution, the Wilcoxon signed-rank test may be more suitable than others.

samples to the total samples that are classified correctly. F1-measure (F1) indicates the overall effectiveness that considers both precision and recall. It is the harmonic mean of Recall and Precision.

5. Experimental results

5.1. Experiments for answering RQ1

Table 3 shows the experimental results of *BGNN4VD* versus 4 baselines on vulnerability detection capability. We can find that *BGNN4VD* outperforms over the baseline methods in terms of Accuracy (74.7%), Recall (76.3%), Precision (77.3%), and F1-measure (76.8%). Table 3 also presents the p-values (tested by the Wilcoxon signed-rank test) when comparing *BGNN4VD* with the baselines in terms of F1-measure. We can observe that *BGNN4VD* shows significant improvements (p-value < 0.05) over others.

In order to measure the improvements of our approach in detection performance compared to baselines, we calculate the percentage of improvement by the following formula:

$$Improvement = \frac{Score_{BGNN4VD} - Score_{baselines}}{Score_{baselines}} \tag{4}$$

Compared with *SySeVR*, *BGNN4VD* achieves 4.9% higher on F1-measure, 11.0% higher on accuracy, 1.5% higher on recall and 8.4% higher on precision. It is attributed to graph representations on the learning of syntax and semantics, making it easier to express the features of the vulnerabilities. In addition, several RNNs (e.g., LSTM, GRU) are used in *SySeVR* to train the detection model. Due to the poor performance of RNNs in handling the non-sequential features of graph-based data, some vital features may be ignored or masked. By contrast, the use of Bidirectional Graph Neural Network (BGNN) makes *BGNN4VD* more efficient in learning the features of graph-based data and accommodating context information through its bidirectional edges. But we also find that the results of *SySeVR* in our experiment are lower than what are shown in the original paper [28]. There are mainly two reasons for this. On the one hand, in the previous experiment of *SySeVR*, they focus on vulnerabilities related to library/API function call, array usage, pointer usage, and arithmetic expression. Constraints on the types of vulnerabilities make their result outstanding but limited. In our experiment, we take all types of vulnerabilities into consideration. Though our results seem to be lower, the performance in detecting other vulnerabilities not included in the four types discussed above is better. On the other hand, the vulnerability dataset they used are mixed and only a small part of vulnerabilities are real-world vulnerabilities. The dataset they used are reported by NVD and SARD[11] (Software Assurance Reference Dataset). Different from real-world vulnerabilities in NVD, vulnerabilities reported by SARD are test cases which are mostly synthetic code (written to test or generated) and test cases (written by students). Due to the high degree of similarity between different cases, a large proportion of vulnerabilities in SARD used for training make the classifier have a bias in detecting real-world vulnerabilities in NVD [40].

In addition, we find that *Flawfinder* and *RATS* can achieve about 50% on accuracy, but are terrible on other measures. This is due to the limitations of static analysis tools, which leads to a high false-positives rate (i.e., its recall is low, and less than 20%). Similarly, *VUDDY* is based on code similarity and it can only detect specific types of vulnerabilities. Its recall and precision are also low (15.7% and 43.3%, respectively). Compared with these approaches, *BGNN4VD* extracts the syntax and semantic information from multiple code graphs to effectively distinguish the subtle difference between vulnerability code and non-vulnerable code.

Moreover, we find that the precision scores achieved seemingly to be low in all methods. There may be two reasons for this. First, our approach detects vulnerabilities at the *function level*, but a vulnerable function we detected contains only one or several vulnerable statements. Too many statements are included while some of them are not relevant to th vulnerability. Second, due to the small amount of data available for training and testing on the CVE dataset we collected, our model may over fit easily when there are too many parameters needed to be adjusted. It leads to the fact that the classifier we trained still has certain bias. We have tried to improve the generalization ability of the model through dropout and other methods in our experiment.

Table 3
Vulnerability detection capability of the state-of-the-art methods and *BGNN4VD*. A: Accuracy; R: Recall; P: Precision.

Method	F1(%)	A(%)	R(%)	P(%)	p-value
Flawfinder	28.6	51.9	19.9	50.5	0.027
RATS	19.3	48.8	14.8	27.9	0.016
SySeVR	73.2	67.3	75.2	71.3	0.039
VUDDY	23.1	72.9	15.7	43.3	0.021
BGNN4VD	**76.8**	**74.7**	**76.3**	**77.3**	–

Table 4
The effect of different types of edges in vulnerability detection. A: Accuracy; R: Recall; P: Precision.

Types of edges	F1(%)	A(%)	R(%)	P (%)	p-value
Only syntax edge	71.7	70.2	73.2	70.3	0.020
Only semantic edges	70.4	70.9	72.3	68.6	0.014
Three forward edges	75.4	72.9	74.4	76.4	0.041
Forward and backward edges	**76.8**	**74.7**	**76.3**	**77.3**	–

Based on the above analysis, we get the following finding.

> **Finding 1:** *BGNN4VD is more effective than the state-of-the-art approaches.*

5.2. Experiments for answering RQ2

The experimental results of different combinations are shown in Table 4. Overall, our approach outperforms combinations of other types of edges in terms of Accuracy, Recall, Precision, and F1-measure. It also shows significant improvements (p-value < 0.05).

When considering either syntax or semantic edges, the difference of accuracy and F1-measure between two types of edges is not obvious. Moreover, we find that only considering syntax edge (i.e., AST) has 1.9% improvement in F1-measure than only considering semantic edges (i.e., CFG and DFG). The reason may be that in the construction process of graphs, syntax edges link all token nodes in the program while semantic edges only link statements and predicate nodes, which makes the learning effect of syntax edge slightly greater than semantic edges. When compared with different types of edges in the training phase, using all six forward and backward edges can improve recall and precision by 4.1% and 8.0% on average, respectively. The reason is that, combining forward and backward edges can accommodate more information of the statements before and after the vulnerable statement.

Based on the above analysis, we get the following finding.

> **Finding 2:** *Using all six forward and backward edges is more effective than other combinations of edges.*

[11] https://samate.nist.gov/SRD/index.php.

Table 5
Comparison between *BGNN4VD* and BGNN. A: Accuracy; R: Recall; P: Precision.

Neural network	F1(%)	A(%)	R(%)	P(%)	p-value
BGNN	74.2	71.9	75.4	73.0	0.031
BGNN4VD	**76.8**	**74.7**	**76.3**	**77.3**	–

Table 6
A part of vulnerabilities detected by *BGNN4VD* on the real-world dataset.

Target project	CVE ID	Vulnerable file	Release date
Linux Kernel	CVE-2020-12771	drivers/md/bcache/btree.c	05/09/2020
	CVE-2020-12770	drivers/scsi/sg.c	05/09/2020
	CVE-2020-12464	drivers/usb/core/message.c	04/29/2020
	CVE-2020-11494	drivers/net/can/slcan.c	04/02/2020
FFmpeg	CVE-2020-12284	libavcodec/cbs_jpeg.c	04/28/2020
Wireshark	CVE-2020-13164	epan/dissectors/packet-nfs.c	05/19/2020
	CVE-2020-11647	epan/dissectors/packet-bacapp.c	04/10/2020

5.3. Experiments for answering RQ3

The comparison results of *BGNN4VD* and BGNN are reported in Table 5. We can see that, comparing with using BGNN only, *BGNN4VD* shows significant improvements (p-value < 0.05) on vulnerability detection capability. Concretely, *BGNN4VD* achieves 3.9% higher on accuracy, 1.2% higher on recall, 5.9% higher on precision, and 3.5% higher on F1-measure. This can be explained by the fact that BGNN aggregates information for each node based on the entire graph. However, for vulnerable functions, the proportion of vulnerable statements in the function is often much smaller than that of non-vulnerable statements. This hinders effective classification over entire graphs [41]. To alleviate this problem, *BGNN4VD* refines the learned features through a CNN module (including a convolutional layer, a Maxpooling layer and a fully-connected layer), and classifies them through MLP.

Based on the above analysis, we get the following finding.

> **Finding 3:** *BGNN4VD achieves a higher vulnerability detection precision and capability than only using BGNN for detecting vulnerabilities.*

5.4. Experiments for answering RQ4

During the experiment, we detected 51 of 113 vulnerable functions, with a precision of 45.1%. This shows the feasibility of *BGNN4VD* in practical application.

For example, in CVE-2020-12284,[12] which is shown in Table 6, this vulnerability occurs due to a missing length check. Through *BGNN4VD*, it can be easily detected because this type of vulnerabilities is closely related to control flow and data flow. Therefore, the semantic feature of this vulnerability can be modeled by combining CFG and DFG. Furthermore, we manually examined the vulnerable code and find that *BGNN4VD* show a poor performance when faced with vulnerabilities which are caused by lacking control constraints. For example, in CVE-2020-11668,[13] the vulnerable file *xirlink_cit.c* misses descriptor sanity checks and may result in NULL-pointers and memory corruption. This vulnerability is caused by the lack of exception handling for unexpected descriptors. This kind of vulnerabilities rely on run-time properties, making it hard to be detected by graph modeling.

Based on the above analysis, we get the following finding.

> **Finding 4:** *BGNN4VD still performs well in detecting the latest reported vulnerabilities of real-world software projects.*

6. Threats to validity

In this section, we discuss the threats to our study, including external threats and internal threats.

External threats. One of the main external threats to our study is the reliability of the data sources. For example, it is important to evaluate the quality of dataset because these data may have some wrong items and are unbalanced in different projects. In addition, since the vulnerability dataset we constructed only contains C/C++ projects, the detection results on our dataset may be specific to the vulnerabilities we included and not applicable to projects written in other programming languages. However, our approach is general because techniques we used are not tied to programming languages. In the practical application, developers can apply our approach to detect potential vulnerabilities for evaluating the security of their products and the detection model of our approach can be trained based on their own vulnerability dataset. Another problem is about the accuracy of the labels. Incorrect labels will directly affect the classification results of the deep learning model. In this paper, we labeled and checked vulnerability data from NVD and GitHub manually. This reduces the probability of mislabeling. Furthermore, since extracting and computing AST, CFG and DFG is time-consuming, we filtered out those vulnerabilities with node size larger than 400 (nearly 13% in our dataset) to improve the efficiency of *BGNN4VD*. In order to process the larger graphs effectively, we plan to use program slicing techniques to reduce the number of irrelevant code.

Internal threats. There are also some internal threats in our implementation. Many uncertain issues in the data processing phase and model training phase, such as sample imbalance and hyper-parameter adjustment may threaten the validity of our approach. With the size of the dataset increasing, more and more factors need to be controlled in the experiment. It is easy to cause accidental experimental results. To alleviate this potential threat, we randomly shuffled the dataset in a same proportion and used validation set to test the generalization performance of the model. We also plan to use Best Linear Unbiased Estimation (BLUE) to evaluate the selection of parameters of our model in the near future. In addition, our experiment only considers AST in the learning of syntax information of the program. In certain aspects, it may mask some other specific syntax information such as library/API function calls and so on.

7. Related work

In this section, We review the most closely related work in four aspects, i.e., bug detection, metric-based vulnerability detection, pattern-based vulnerability detection, and bytecode-level vulnerability detection.

Bug detection. Many techniques have been developed for bug detection. In existing bug detection approaches [20,22–26,42], several programming rules are predefined to statically detect common programming flaws or defects. Nam et al. [20] proposed novel approaches, *CLA* and *CLAMI*, that show the potential for defect prediction on unlabeled dataset in an automated manner through labeling an unlabeled dataset by using the magnitude of metric values. Pradel et al. [42] presented *DeepBugs*, a learning approach to name-based bug detection, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them. To address the challenge that effectively learning a bug detector requires examples of both correct and incorrect code, they created likely incorrect code examples from an existing corpus of code through

[12] https://nvd.nist.gov/vuln/detail/CVE-2020-12284.

[13] https://nvd.nist.gov/vuln/detail/CVE-2020-11668.

simple code transformations. However, bug detection approaches cannot be used to detect vulnerabilities directly because not all bugs belong to vulnerabilities.

Metric-based vulnerability detection. In these approaches, vulnerability detection is based on some metrics (or rules) defined by human experts. These metrics mainly contain seven categories: cohesion metrics, complexity metrics, coupling metrics, documentation metrics, inheritance metrics, code duplication metrics, and size metrics. Younis et al. [43] selected eight code measures (including line of code, degree of nesting, information flow, calling function and so on) to describe vulnerability characteristics from Linux and Apache HTTP server. They tested the predictive ability of the selected measures using four classifiers. Du et al. [4] proposed a generic, lightweight and extensible framework which combined code complexity metrics with vulnerability metrics to rank the functions and identifies the top ones as potentially vulnerable. Different from their methods, our work uses deep learning-based approaches instead of hand-craft metrics, which does not rely on human experts to define features and can achieve lower *false-negative rates*.

Pattern-based vulnerability detection. Li et al. [27,28] proposed two vulnerability detection systems based on deep learning (i.e., *VulDeePecker* and *SySeVR*). *VulDeePecker* used *code gadget* to represent programs and then transformed them into vectors to automatically learn vulnerability patterns. *SySeVR* was based on syntax and semantics and can fully consider data dependence and control dependence. They encoded the obtained slice data into vectors to input BGRU for learning the characteristics of vulnerabilities. For the above two models, they need to truncate the vectors transformed when vectors are longer than a threshold, which will lose part of the information. Our model retains the syntax and semantic information by using a larger state vector through padding of node features, which makes our model more expressive. Closest to our work is the work of Zhou et al. [32] who proposed a graph neural network-based model for graph-level classification through learning on a rich set of code semantic representations. It included a novel *Conv* module to efficiently extract useful features in the learned rich node representations for graph-level classification. Our work is based on BGNN by introducing forward edges and their backwards edges, which can speed up information propagating and improve the accuracy of vulnerability detection.

Binary-level vulnerability detection. Several methods have proposed to detect vulnerabilities in bytecode [13–15,17,19]. Bytecode is a kind of code between the source code and the machine code, which can represent the semantic information of the code at a lower-level and executed directly in the virtual machine such as Java, Python, and PHP. It can reduce hardware and operating system dependence by allowing the same code to run cross-platform and has shown performance advantage on vulnerability detection. Guo et al. [13] presented *VulHunter*, an automated vulnerability detection system based on deep learning and bytecode. With graph-based static analysis methods, *VulHunter* could find the code related to the vulnerability and then transformed it into bytecode slices. *VulHunter* achieved good performance for SQL injection, XSS and mixed types vulnerabilities detection respectively. Xu et al. [17] proposed the Binary X-Ray ($BinXray$), a patch based vulnerability matching approach, is proposed to identify the specific 1-day vulnerabilities in target programs accurately and effectively. They designed a basic block mapping algorithm to extract the signature of a patch and applied the semantics of patches to reduce irrelevant basic block traces to speed up the signature searching. These approaches have different features and face different challenges. Therefore, we did not compare *BGNN4VD* with them.

8. Conclusion

We present *BGNN4VD*, a bidirectional graph neural network-based vulnerability detection approach. We add the backward edges during the model training phase to learn rich syntax and semantics information of programs and use a convolutional layer to extract outputs for classification. The evaluation results show that *BGNN4VD* achieves a higher precision and accuracy than the state-of-the-art methods. In addition, we apply *BGNN4VD* on the latest vulnerabilities reported by CVE, and *BGNN4VD* can achieve a precision at 45.1% , which demonstrates the feasibility of *BGNN4VD* in practical application.

In future work, we will improve our study in two aspects. First, we only focus on three types of edges and their backward edges. Future research should be conducted by considering more types of edges to increase the performance of vulnerability detection. Second, we evaluate *BGNN4VD* on C/C++ projects. In the future, we will try to develop a general graph mining tool for different programming languages to extract different types of code graphs.

CRediT authorship contribution statement

Sicong Cao: Methodology, Software, Validation, Writing - original draft, Writing - review & editing. **Xiaobing Sun:** Conceptualization, Methodology, Supervision, Project administration. **Lili Bo:** Conceptualization, Supervision. **Ying Wei:** Software, Validation. **Bin Li:** Supervision, Writing - review & editing.

Declaration of competing interest

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

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61872312, No. 61972335, No. 62002309); the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project, China (No. YZU201803); the Six Talent Peaks Project in Jiangsu Province, China (No. RJFW-053), the Jiangsu "333" Project, China, the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University, China (No. KFKT2020B15, No. KFKT2020B16), and Yang-zhou University Top-level Talents Support Program (2019), China.

References

[1] X. Sun, X. Peng, K. Zhang, Y. Liu, Y. Cai, How security bugs are fixed and what can be improved: an empirical study with mozilla, Sci. China Inf. Sci. 62 (1) (2019) 19102:1–19102:3.

[2] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, Q. He, Smart contract vulnerability detection using graph neural network, in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 3283–3290.

[3] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, M. Nagappan, Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project, Empir. Softw. Eng. 22 (3) (2017) 1305–1347.

[4] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, Y. Jiang, Leopard: identifying vulnerable code for vulnerability assessment through program metrics, in: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, 2019, pp. 60–71.

[5] S. Kim, S. Woo, H. Lee, H. Oh, VUDDY: a scalable approach for vulnerable code clone discovery, in: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, IEEE Computer Society, 2017, pp. 595–614.

[6] Y. Shin, A. Meneely, L. Williams, J.A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (6) (2011) 772–787.

[7] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: 28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012, ACM, 2012, pp. 359–368.

[8] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, X. Shi, Analyzing bug fix for automatic bug cause classification, J. Syst. Softw. 163 (2020) 110538.

[9] L. Jiang, H. Liu, H. Jiang, L. Zhang, H. Mei, Heuristic and neural network based prediction of project-specific api member access, IEEE Trans. Softw. Eng. (2020) 1.

[10] W. Drozd, M.D. Wagner, Fuzzergym: A competitive framework for fuzzing and learning, 2018, CoRR abs/1807.07490.

[11] E.H. Boudjema, S. Verlan, L. Mokdad, C. Faure, VYPER: Vulnerability detection in binary code, Secur. Priv. 3 (2) (2020).

[12] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, C. Zhao, Eliminating path redundancy via postconditioned symbolic execution, IEEE Trans. Softw. Eng. 44 (1) (2018) 25–43.

[13] N. Guo, X. Li, H. Yin, Y. Gao, Vulhunter: An automated vulnerability detection system based on deep learning and bytecode, in: J. Zhou, X. Luo, Q. Shen, Z. Xu (Eds.), Information and Communications Security - 21st International Conference, ICICS 2019, Beijing, China, December 15-17, 2019, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 11999, Springer, 2019, pp. 199–218.

[14] K. Xu, Y. Li, R.H. Deng, K. Chen, Deeprefiner: Multi-layer android malware detection system applying deep neural networks, in: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018, IEEE, 2018, pp. 473–487.

[15] Z. Yuan, Y. Lu, Z. Wang, Y. Xue, Droid-sec: deep learning in android malware detection, in: ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014, ACM, 2014, pp. 371–372.

[16] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, L. Zhang, Deep learning based program generation from requirements text: Are we there yet?, IEEE Trans. Softw. Eng. (2020) 1.

[17] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, T. Liu, Patch based vulnerability matching for binary programs, in: S. Khurshid, C.S. Pasareanu (Eds.), ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, ACM, 2020, pp. 376–387.

[18] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, L. Zhang, Deep learning based code smell detection, IEEE Trans. Softw. Eng. (2019) 1.

[19] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song, SPAIN: security patch analysis for binaries towards understanding the pain and pills, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE / ACM, 2017, pp. 462–472.

[20] J. Nam, S. Kim, CLAMI: Defect prediction on unlabeled datasets (T), in: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society, 2015, pp. 452–463.

[21] T. Zhou, X. Sun, X. Xia, B. Li, X. Chen, Improving defect prediction with deep forest, Inf. Softw. Technol. 114 (2019) 204–216.

[22] T. Gyimóthy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Softw. Eng. 31 (10) (2005) 897–910.

[23] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.

[24] D. Radjenovic, M. Hericko, R. Torkar, A. Zivkovic, Software fault prediction metrics: A systematic literature review, Inf. Softw. Technol. 55 (8) (2013) 1397–1418.

[25] F. Zhang, Q. Zheng, Y. Zou, A.E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 309–320.

[26] Q. Huang, X. Xia, D. Lo, Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction, Empir. Softw. Eng. 24 (5) (2019) 2823–2862.

[27] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, 2018.

[28] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, J. Wang, Sysevr: A framework for using deep learning to detect software vulnerabilities, 2018, CoRR abs/1807.06756.

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

[30] R.L. Russell, L.Y. Kim, L.H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P.M. Ellingwood, M.W. McConley, Automated vulnerability detection in source code using deep representation learning, in: 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018, IEEE, 2018, pp. 757–762.

[31] H.K. Dam, T. Tran, T. Pham, S.W. Ng, J. Grundy, A. Ghose, Automatic feature learning for vulnerability prediction, 2017, CoRR abs/1708.02368.

[32] Y. Zhou, S. Liu, J.K. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada, 2019, pp. 10197–10207.

[33] Y. Li, D. Tarlow, M. Brockschmidt, R.S. Zemel, Gated graph sequence neural networks, in: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.

[34] I.D. Baxter, A. Yahin, L.M. de Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: 1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998, IEEE Computer Society, 1998, pp. 368–377.

[35] S. Sparks, S. Embleton, R. Cunningham, C.C. Zou, Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting, in: 23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA, IEEE Computer Society, 2007, pp. 477–486.

[36] H. Gascon, F. Yamaguchi, D. Arp, K. Rieck, Structural detection of android malware using embedded call graphs, in: AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-Located with CCS 2013, Berlin, Germany, November 4, 2013, ACM, 2013, pp. 45–54.

[37] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net, 2017.

[38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P.A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, Tensorflow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, USENIX Association, 2016, pp. 265–283.

[39] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[40] W. Zheng, J. Gao, X. Wu, F. Liu, Y. Xun, G. Liu, X. Chen, The impact factors on the performance of machine learning-based vulnerability detection: A comparative study, J. Syst. Softw. 168 (2020) 110659.

[41] M. Zhang, Z. Cui, M. Neumann, Y. Chen, An end-to-end deep learning architecture for graph classification, in: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th Innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, AAAI Press, 2018, pp. 4438–4445.

[42] M. Pradel, K. Sen, Deepbugs: a learning approach to name-based bug detection, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 147:1–147:25.

[43] A.A. Younis, Y.K. Malaiya, C. Anderson, I. Ray, To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit, in: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, la, USA, March 9-11, 2016, ACM, 2016, pp. 97–104.