The Journal of Systems & Software 204 (2023) 111772

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

BiTCN_DRSN: An effective software vulnerability detection model based on an improved temporal convolutional network☆

Jinfu Chen [a,b], Wei Lin [a,b], Saihua Cai [a,b,*], Yemin Yin [a,b], Haibo Chen [a,b], Dave Towey [c]

[a] School of Computer Science and Communication Engineering, Jiangsu University, 301 Xuefu Road, Zhenjiang, 212013, Jiangsu, China
[b] Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace, Jiangsu University, 301 Xuefu Road, Zhenjiang, 212013, Jiangsu, China
[c] School of Computer Science, University of Nottingham Ningbo China, 199 East Taikang Road, Ningbo, 315100, Zhejiang, China

ARTICLE INFO

Article history:
Received 8 January 2023
Received in revised form 20 May 2023
Accepted 31 May 2023
Available online 7 June 2023

Keywords:
Software security
Vulnerability detection
Deep learning
Deep residual shrinkage network

ABSTRACT

The detection of software vulnerabilities is a challenging task in the field of security. With the increasing scale of software and the rapid development of artificial intelligence technology, deep learning has been extensively applied to automatic vulnerability detection. Temporal Convolutional Networks (TCNs) have been shown to perform well in tasks that can be processed in parallel; they can adaptively learn complex structures (including in-time series data); and they have exhibited stable gradients — they are relatively easier to train, and can quickly converge to an optimal solution. However, TCNs cannot simultaneously capture the bidirectional semantics of the source code, since they do not have a bidirectional network structure. Furthermore, because of the weak noise resistance of residual TCN connections, TCNs are also susceptible to learning features that are not related to vulnerabilities when learning the source code features. To overcome the limitations of the traditional TCN, we propose a bidirectional TCN model based on the Deep Residual Shrinkage Network (DRSN), namely BiTCN_DRSN. BiTCN_DRSN combines TCN and DRSN to enhance the noise immunity and make the network model more attentive to the features associated with vulnerabilities. In addition, addressing the limitation that the TCN is a unidirectional network structure, the forward and backward sequences are utilized for bidirectional source-code feature learning. The experimental results show that the proposed BiTCN_DRSN model can effectively improve the accuracy of source-code vulnerability detection, compared with some existing neural-network models. Compared with the traditional TCN, our model increases the accuracy by 4.22%, 2.42% and 2.66% on the BE-ALL, RM-ALL and HY-ALL datasets, respectively. The proposed BiTCN_DRSN model also exhibits improved detection stability.

1. Introduction

With the rapid development of the Internet, computers have penetrated into all walks of life, including medical, automotive, and military. The availability and reliability of Internet-based services are increasingly demanding (Zheng et al., 2017, 2022; Chen et al., 2020). The huge leap forward of software technology and the growing demands of users have led to increasing software complexity, which has increased the possibility of software vulnerabilities. Attackers can exploit software vulnerabilities to attack software systems, thereby violating the privacy of users and threatening the security of the system. Accordingly, security issues caused by software vulnerabilities receive significant attention. Unfortunately, vulnerabilities remain pervasive, as indicated by their continuing rise in number (as reported by Skybox Security skyboxsecurity, 2021): There were 9444 new vulnerabilities reported in the first half of 2021 alone. The detection of software vulnerabilities is a challenging issue in the field of security. Vulnerability detection for source code has, therefore, become increasingly important. The model proposed in this paper can be used to detect vulnerabilities in open-source code, and thus help safeguard software security.

The development of deep-learning techniques in recent years has led to the application of deep models to vulnerability detection (Zhou et al., 2019; Li et al., 2018; Liu et al., 2019). Compared with traditional machine-learning methods, deep-learning models are capable of automatically extracting advanced features from structured data, thereby reducing the feature-extraction effort, and freeing experts from the tedious and subjective tasks of manually defining vulnerability features (Cao et al., 2022). In

☆ Editor: Aldeida Aleti.

* Corresponding author at: School of Computer Science and Communication Engineering, Jiangsu University, 301 Xuefu Road, Zhenjiang, 212013, Jiangsu, China.
E-mail addresses: jinfuchen@ujs.edu.cn (J. Chen), 2211908019@stmail.ujs.edu.cn (W. Lin), caisaih@ujs.edu.cn (S. Cai), 2212108027@stmail.ujs.edu.cn (Y. Yin), hbchen@stmail.ujs.edu.cn (H. Chen), Dave.Towey@nottingham.edu.cn (D. Towey).

https://doi.org/10.1016/j.jss.2023.111772
0164-1212/

addition, the features automatically extracted by deep-learning methods have better generalization capability than those extracted manually (Lin et al., 2019). Source code can be considered a kind of text data with temporal features. Variants of recurrent neural networks (RNNs) have been used to capture the contextual semantics of source code, to detect vulnerabilities: These have included: Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997); Gated Recurrent Unit (GRU) (Cho et al., 2014); Bidirectional LSTM (BiLSTM); Bidirectional GRU (BiGRU); and BiLSTM with attention mechanisms. The RNN self-looping mechanism facilitates the modeling of temporal data, but it is also the loop structure that leads to the problems of gradient vanishing or gradient explosion in RNNs (Hochreiter, 1998; Pascanu et al., 2013). Although these RNN variants can, to some degree, solve the vanishing/exploding gradient issues through the gating mechanism, they cannot parallel-process the data in the way that convolutional neural networks (CNNs) can. Furthermore, each LSTM unit needs to compute three gates, which can be computationally expensive.

Considering the above-mentioned limitations, the temporal convolutional network (TCN) has often been used to process temporal data in parallel to detect vulnerabilities in source code (Zhang, 2019; Chen et al., 2021). TCNs can better capture longer dependencies in the source code than traditional CNNs. In contrast to the RNN variants, the weight of each time step in a TCN is updated simultaneously (Huang and Hain, 2021), providing high parallelism. Unlike the self-looping structure of RNNs, TCNs use a back-propagation path that is different from the standard processing direction. This allows TCNs to avoid the RNN issues of gradient explosion and gradient disappearance. In recent years, TCNs have been used in many areas, including anomaly detection in Internet of Things communication (Cheng et al., 2021), speech processing (Huang and Hain, 2021), and traffic prediction (Kuang et al., 2020). TCN's application in these studies demonstrates its viability for learning time-series data features.

However, TCN also has certain limitations in the area of vulnerability detection, including:

1. The detection of vulnerabilities relies on some vulnerability-related code information, but most source-code statements are not correlated with vulnerabilities. Due to the interference of these irrelevant statements, the source code features learned by traditional TCNs are often not sufficient to correctly detect the vulnerabilities.
2. The traditional TCN is a neural network with a unidirectional structure (because of causal convolutions). However, source code (regarded as text) requires a bidirectional neural-network structure to capture the contextual semantics.
3. As the number of TCN layers increases, the receptive field of the TCN model grows: The number of nodes in the previous layer that can impact or influence increases, which can affect the stability of the model. The Residual Shrinkage Network (RSN) commonly used in TCNs can improve the stability of the model (Zhao et al., 2020). However, noisy data in the source code can reduce the RSN's ability to select effective features, thereby affecting the TCN vulnerability-detection.

Fig. 1 shows a simple buffer overflow vulnerability. The vulnerability relates to when the length of the first input parameter exceeds the size of the buffer (8): The copying of the second parameter exceeds the buffer, and overwrites the adjoining memory. Hackers can exploit this kind of flaw to damage software systems. As the figure shows, other code statements between the vulnerability-related statements are not related to the flaw, and are thus considered noise. Due to the limitations of TCNs, buffer vulnerabilities like this (with bi-directional semantics) cannot be identified.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char* argv[]) {
5      char buffer[8];
6      int password = 0;
7
8      printf("Please enter the password: ");
9      scanf("%d", &password);
10
11     if (password == 0x12345678) {
12         printf("Access granted.\n");
13         strcpy(buffer, argv[1]);
14     } else {
15         printf("Access denied.\n");
16     }
17
18     return 0;
19 }
```

Fig. 1. Motivating example (buffer overflow vulnerability).

Research goal: We propose a new TCN-based vulnerability-detection model that captures the contextual information of the code and reduces the impact of vulnerability-irrelevant information on the vulnerability detection.

Principal idea: In order to capture the contextual information of the source code, we change our traditional TCN model to a bi-directional network model. To reduce the impact of vulnerability-irrelevant information on the vulnerability detection, we combine a DRSN with a TCN. Combining these two improvements to the TCN, we obtain our vulnerability detection model: BiTCN_DRSN. This paper was motivated by the following research questions (RQs):

1. **RQ1**: Does the bidirectional TCN network model have better detection capability and stability?
2. **RQ2**: Do deep residual networks improve on TCN network models in the field of vulnerability detection?
3. **RQ3**: Does the combination of these two improvements to the TCN network architecture deliver an enhancement?
4. **RQ4**: Does the proposed BiTCN_DRSN model outperform the state-of-the-art models in vulnerability-detection capability and stability?

The RQ1 answer should clarify the extent of improvement provided by the bidirectional temporal convolutional model over the underlying network model. The RQ2 answer should show the detection-efficiency enhancement resulting from combining the bidirectional time convolution model with the deep residual network. The answer to RQ3 should indicate whether or not the BiTCN_DRSN vulnerability detection outperforms that of each component network (TCN, TCN DRSN, and BiTCN). Finally, the RQ4 answer should show whether or not our proposed BiTCN_DRSN[1] vulnerability-detection model is superior to existing models.

The major contributions of this paper are as follows:

1. We integrate DRSN with TCN to reduce the effect of vulnerability-irrelevant source-code statements and redundant information on software-vulnerability detection,

[1] https://github.com/mochu8042/BiTCN_DRSN

thereby further improving the vulnerability-detection stability.
2. We propose the BiTCN_DRSN model, based on DRSN and TCN, which can capture the source-code information, both forwards and backwards, thereby capturing the semantic dependencies in both directions.
3. We conduct extensive experiments on three publicly available datasets to evaluate the BiTCN_DRSN model. The experimental results indicate that, compared with existing neural networks used in source-code vulnerability detection, the proposed BiTCN_DRSN model achieves higher detection accuracy and precision, lower false-positive rates, faster neural-network convergence, and better stability.

The remaining parts of this paper are organized as follows. Section 2 introduces the related work and background. Section 3 describes the software-vulnerability-detection model based on the bidirectional TCN and the DRSN: BiTCN_DRSN. Section 4 presents the experimental results and analysis. Section 5 concludes the paper, and discusses future work.

2. Related work and background

The use of deep learning methods has made automated source-code vulnerability-detection easier. This section discusses a variety of deep-learning-based vulnerability-detection methods, and introduces the TCN and DRSN.

2.1. Related work

Software vulnerabilities are often related to insecure code (Wu et al., 2022), and vulnerabilities in open-source code can proliferate easily. Earlier detection of vulnerabilities can reduce losses, making vulnerability detection for source code a crucial activity. Current approaches to source-code-based static vulnerability detection can be divided into three categories: rule-based (Reynolds et al., 2017; Fang et al., 2017); code-similarity-based (Kim et al., 2017; Zhu et al., 2019; Shi et al., 2019); and machine-learning-based (Grieco et al., 2016; Younis et al., 2016). Rule-based methods require experts to generate rules that should detect vulnerabilities. Examples of tools in this category are Flawfinder, Cppcheck, and Splintt (Liu et al., 2012). Challenges for this category of tools include that they are generally more dependent on vulnerability-related databases and pre-defined vulnerability rules: This can mean that they are only suitable for simple vulnerabilities, and that they may have high false-positive and false-negative rates when processing more complex vulnerabilities. Code-similarity-based methods are only able to detect vulnerabilities related to code cloning, and can have high false-negative rates for vulnerabilities that are not caused by code cloning (Novak et al., 2019). Traditional machine-learning-based vulnerability detection typically requires extraction of features from the source code, and using them in a machine-learning model for learning and classification. These methods do not only rely on the domain knowledge, but can use a coarse granularity to represent the program, which may lead the identified vulnerabilities being located in a large segment of code (Wang et al., 2021).

Because RNNs are designed to handle sequential data, and source code text is a kind of sequential data, many studies have used RRN variants to learn the semantics of software vulnerabilities. LSTM and GRU address the vanishing-gradient problem with a memory-gating mechanism, and their bidirectional versions (BiLSTM and BiGRU) can capture long-term and bidirectional dependencies of the source code: BiLSTM and BiGRU have, therefore, become popular in software-vulnerability detection. Li et al. (2018) proposed VulDeePecker, a vulnerability-detection system based on BiLSTM and code gadgets: Code statements related to each argument (or variable) of library/API function calls are used to compose one or more code gadgets, which are then labeled as a vulnerability or not.

The dataset and code-slicing method used in the current study were based on those use in the VulDeePecker study (Li et al., 2018). This code-slicing method was selected because:

1. The generated code slices usually consist of a small number of code statements, which provides the desired granularity for vulnerability detection.
2. Data dependencies are used to combine semantically-related code into code gadgets, reducing the amount of vulnerability-irrelevant information generated.
3. This method maps user-defined variables and functions to symbolic names in a one-to-one fashion, eliminating noise caused by different naming styles.

AIdetectorX (Chen et al., 2021) uses a self-attention mechanism in the last layer to improve on the TCN model: It only assigns the weights to the features learned in the last layer. In contrast, our proposed method adds a deep residual shrinkage network (DRSN) to each residual block to filter features learned in that block. DRSN can identify unimportant features through the self-attention mechanism, and then reduce or remove their influence, thereby focusing on the useful features. Furthermore, the AIdetectorX TCN is unidirectional, but our proposed method includes an improved TCN with a bidirectional structure that facilitates integration of both prior and later code information. These two improvements can improve the vulnerability-detection ability.

Li et al. (2022a) introduced VulDeeLocator, a vulnerability-detection model for the C programming language. VulDeeLocator connects multiple project files through a define-use relationship, and replaces the source-code representation with an intermediate code representation. VulDeeLocator also extends the bidirectional Recurrent Neural Network (BiRNN) model with an attention mechanism and detection granularity refinement, calling the extension BiRNN-vdl. VulDeeLocator can not only detect vulnerabilities, but can also reduce the vulnerability scope to just three lines. In contrast to the VulDeeLocator neural-network model, the underlying TCN model in this paper combines aspects of the RNN and CNN architectures, which ensures that the TCN has a longer effective-history size than an RNN.

Zhou et al. (2019) proposed Devign, a graph neural network model based on a composite-code representation. Devign uses an abstract syntax tree (AST), encoding different levels of data dependencies and control dependencies as a joint graph whose different edge types represent different dependency features. This composite code representation combines various semantic, syntactic and structural information to capture the vulnerability types and patterns. This makes it possible for Graph Neural Networks (GNNs) to better learn node representations, and to pass information about neighboring nodes in the graph. Finally, a convolutional module extracts important nodes for prediction.

Chakraborty et al. (2021) proposed the ReVeal code-vulnerability-detection method. ReVeal uses a code property graph (CPG) and a gated graph neural network (GGNN) to obtain the graph-structure features for feature extraction. They introduced the SMOTE (Synthetic Minority Over-sampling Technique) algorithm to solve the sample imbalance problem, and used MLP for vulnerability detection. Their method can achieve good detection results on real datasets of C code, at a function-detection granularity.

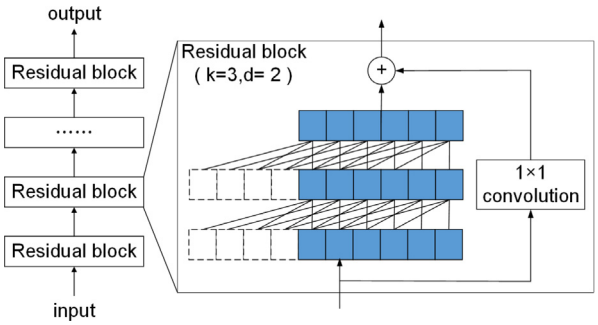


Fig. 2. The structure of TCN.

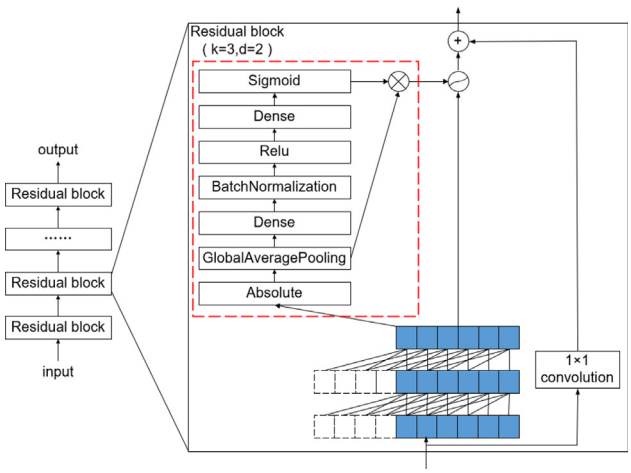


Fig. 3. The structure of TCN_DRSN.

2.2. Background

2.2.1. TCN

Bai et al. (2018) proposed a temporal convolutional network (TCN) model for sequence modeling that outperformed LSTM and GRU when learning temporal data. The TCN structure is shown in Fig. 2. The TCN is stacked with multiple residual blocks, each of which consists of two dilated causal convolutional layers and a skip connection. In Fig. 2, the 1×1 convolution is used to adjust the dimensionality of the tensor, k denotes the size of the 1-D convolution kernel, d is the dilated rate, and the dashed boxes represent the padding of each layer. The right side of Fig. 2 shows the structure of the residual block for $k = 3$ and $d = 2$. Because the view range of causal convolution on time-series data is bounded by the size of the convolution kernel, it is typically necessary to overlay multiple layers in order to capture long-term dependencies. The TCN dilated convolution method alleviates this problem through interval sampling, where $d = 2$ means that only every second point is sampled. Being further up the residual block indicates a larger dilated rate, which makes the effective window size grow exponentially with the number of layers. The dilated causal convolutional method efficiently widens the receiving field of convolutional layers while avoiding the degradation of training results due to stacking multiple layers of convolution. The residual connection incorporates the input of the residual block and the output of the dilated causal convolutions, which gives the network the ability to pass messages across layers. The $\oplus$ symbol in Fig. 2 denotes an additive operation. The residual connection can address the gradient-vanishing issue in the back-propagation process and, to a certain extent, solve the network-degradation problem (He et al., 2016). Compared with the RNN variants, TCN has very good parallel computation, low memory consumption, stable gradient, and a flexible perceptual field.

2.2.2. DRSN

Zhao et al. (2020) introduced "soft thresholding" as a "shrinkage layer" into the residual module of their proposed DRSN, and then innovated the adaptive setting method of thresholds. Because of the soft thresholding, the DRSN can effectively reduce the messages that are not relevant to the current task when learning the features. The key concept of soft thresholding is the dropping of features whose absolute value is less than a certain threshold, and the shrinking of features whose absolute value is greater than this threshold. If x represents the input feature before soft thresholding, τ is the threshold value (a positive parameter), and y denotes the output feature after soft thresholding, then the soft thresholding is calculated as in Eq. (1):

$$y = \begin{cases} x - \tau & x > \tau \\ 0 & -\tau \le x \le \tau \\ x + \tau & x < -\tau \end{cases} \tag{1}$$

It can be observed from Eq. (1) that when $x = \tau$, the derivative of y to x is 0; otherwise, it is equal to 1. This prevents the problems of both vanishing and exploding gradients.

DRSN uses deep learning to automatically determine the threshold, which avoids the challenges of manual determination.

3. Proposed model

When addressing the temporal nature of the source code as text data, TCNs have the advantages of being able to process the data in parallel, and having perceptual field flexibility, and gradient stability. However, the unidirectional TCN structure still has some limitations: TCNs cannot adequately capture bidirectional features of the source code; and traditional TCNs are not able to eliminate the feature information that is not associated with the vulnerability-detection task. To address these two weaknesses, we made some improvements to the traditional TCN.

3.1. TCN_DRSN

During the process of vulnerability detection, the examined samples may contain some noise, which could be understood as features of the source code that are not relevant to the vulnerability-detection task. To address the influence of noise in the source code, we propose an improved TCN model, called TCN_DRSN, that changes the residual connection in TCN to a DRSN, to decrease the effect of noise in the source code. Fig. 3 shows the improved neural-network model. Unlike the original TCN, TCN_DRSN has specially-designed subnetworks for threshold estimation, as shown in the red dashed box in Fig. 3. Also unlike the original TCN, TCN_DRSN does not directly add the output of the dilated causal convolutional layer to the output of the 1×1 convolutional layer: Instead, features output from the dilated causal convolutional layer are first soft thresholded and then added to the features output from the convolutional layer. When learning the source-code features, the improved model can reduce the influence of irrelevant features and identify key features.

A channel is equivalent to the data in a certain feature distribution after the previous input has been convolved. To ensure that each sample's channel has its own thresholds, the thresholds are learned by a threshold estimation building unit (Zhao et al., 2020). Each threshold-estimation building unit first takes the absolute value of each element of the output matrix of the inflated causal convolution layer. And then, the one-dimension global average pooling is applied to the matrix after taking the

absolute values, to obtain the mean value of the features after removing the absolute value in each channel. Its output is a one-dimensional vector. If the output of the second dilated causal convolution layer is a matrix $X \in R^{T\times K}$ (where T is the number of tokens and K is the number of convolution kernels), then the one-dimensional vector $X_{mean} \in R^K$ after global average pooling can be calculated as in Eq. (2). This is used to average each channel as a basis for later thresholding.

$$X_{mean} = \frac{1}{T}\sum_{i=1}^{T}|X_i| \tag{2}$$

where X_i denotes the ith row of features in X, and $|\cdot|$ denotes the absolute value. This one-dimensional vector is then propagated sequentially to a fully connected layer, then the batch normalization layer, the Relu activation function, and, finally, another fully connected layer. Finally, the threshold parameter for each channel is scaled in the range (0,1) using a sigmoid activation function. The scaling parameter can be described as in Eq. (3):

$$scales = \frac{1}{1+e^{-Z}} \tag{3}$$

where $Z \in R^K$ is the output of the second fully connected layer, and $scales \in R^K$ is the scaling parameter. In other words, $scales_i$ is the ith element in scales, and corresponds to the scaling parameter of the ith channel. In conclusion, the threshold $\tau \in R^k$ for the channels can be calculated using Eq. (4).

$$\tau = scales * X_{mean} \tag{4}$$

where $*$ is the element-wise product of inputs. Similarly, τ_i denotes the ith element of vector τ, and is the threshold for the ith channel.

It is clear that the learning of thresholds is similar to the attention mechanism. It is possible to identify the degree of unimportance of each channel feature. Manually setting an appropriate threshold for noise reduction in vulnerability detection is almost impossible: The thresholds in the DRSN residual shrinkage building unit are set through deep learning. This ensures that each threshold is positive, is not too large, and can be controlled within a reasonable range by the neural network. The threshold estimation subnetwork enables each sample's channel to have its own unique shrinkage threshold. This means that the neural-network model can address situations where the noise content of each sample is different.

Algorithm 1 Soft Thresholding

Input: the output of the two-layer dilated causal Convolutional $X \in R^{T\times K}$

Output: Feature $y \in R^{T\times K}$ after soft thresholding

1: $y \leftarrow \emptyset$
2: $X_{abs} \leftarrow$ absolute(X)
3: $X_{mean} \leftarrow$ global average pooling X_{abs}
4: $scales \leftarrow$ calculating scaling coefficient by network
5: $\tau \leftarrow$ the element-wise product of X_{mean} and $scales$
6: **for** each channel $X^i_{abs} \in X_{abs}$ **do**
7: $\quad y_i \leftarrow X^i_{abs} - \tau_i$
8: **end for**
9: **for** each element $y_{ij} \in y$ **do**
10: $\quad y_{ij} \leftarrow sign(x_{ij}) \times max(0, y_{ij})$
11: **end for**
12: **return** y

Algorithm 1 presents the soft thresholding algorithm. Lines 2–5 contain the process of learning the threshold for each channel through the residual shrinkage building unit. Lines 6–11 contain the soft thresholding process. $X_{abs} \in R^{T\times K}$ is the matrix of absolute values of feature map X, and X^i_{abs} denotes the absolute values of the features of the ith channel in X. X^i_{abs} needs to be subtracted from the corresponding threshold generated in lines 2–5 to obtain the features after shrinking in the zero direction. If the value of each element less its own threshold is less than zero, then that element's value is taken as zero: The near-zero features are set to zero due to soft thresholding to retain the interesting features. At the end, each element value in the matrix is multiplied by the positive and negative sign of the corresponding element in the original matrix, to obtain the features after soft thresholding.

To some extent, the learning of thresholds and soft thresholding can be regarded as a special kind of attention mechanism. The intention is to identify features that are not important to the vulnerability-detection task, and set them to zero or reduce their impact.

3.2. BiTCN_DRSN

The traditional TCN is a unidirectional structure, whose prediction is based on historical information. Because the TCN causal convolution has strict temporal constraints, its neurons at moment t can only see the information before that moment, not anything later. However, numerous studies have suggested that a bidirectional neural network structure is more conducive to the learning of features in text data (Li et al., 2018). Source code is also text data, and its semantic information may be associated with its context: The semantic features of the code involved in the vulnerability are not only relevant to the code that appears before the vulnerability-related code, but also to the code that follows it. This is the reason for which BiLSTM and BiGRU have become so popular for vulnerability detection (Li et al., 2018). Because the unidirectional TCN has some limitations for learning overall semantic features from source code, we developed the BiTCN_DRSN model based on TCN_DRSN: BiTCN_DRSN has the ability to simultaneously capture both historical and future information from the source code.

Given the input token vector sequence $\overrightarrow{X} = \{x_1, x_2, \ldots, x_T\}$, $\overrightarrow{X} \in R^{T\times E}$, where T is the number of tokens, and E is the vector length of each token embedding. The first step is to obtain the reversed vector sequence $\overleftarrow{X} = \{x_T, x_{T-1}, \ldots, x_1\}$, $\overleftarrow{X} \in R^{T\times E}$: $\overleftarrow{X}$ has the opposite order to the token vector sequence. The two vector sequences are then fed into the two TCN_DRSN models for feature learning. The forward feature vector $\overrightarrow{y}$ and the backward feature vector $\overleftarrow{y}$ are the features learned by the respective TCN_DRSN and global maximum pooling. Both $\overrightarrow{y}$ and $\overleftarrow{y}$ are of length K, the number of convolutional kernels in TCN_DRSN. The forward and backward features are varied nonlinearly, separately, as Eqs. (5) and (6) (Qiu, 2020).

$$\overrightarrow{h} = \sigma(W_1\overrightarrow{y} + b_1) \tag{5}$$

$$\overleftarrow{h} = \sigma(W_2\overleftarrow{y} + b_2) \tag{6}$$

where $\overrightarrow{h}$ and $\overleftarrow{h}$ are the forward and backward features after nonlinear variations, respectively; $W_1, W_2 \in R^{K\times K}$ are the weight matrices; $b_1, b_2 \in R^K$ are the biases; and $\sigma(\cdot)$ is the activation function. The activation function used in this model is Relu (Daubechies et al., 2022), as shown in Eq. (7):

$$\sigma(x) = \begin{cases} x & x > 0 \\ 0 & x \le 0 \end{cases} \tag{7}$$

Finally, the two learned features are fused using Eq. (8):

$$y = Concatenate(\overrightarrow{h}, \overleftarrow{h}) \tag{8}$$

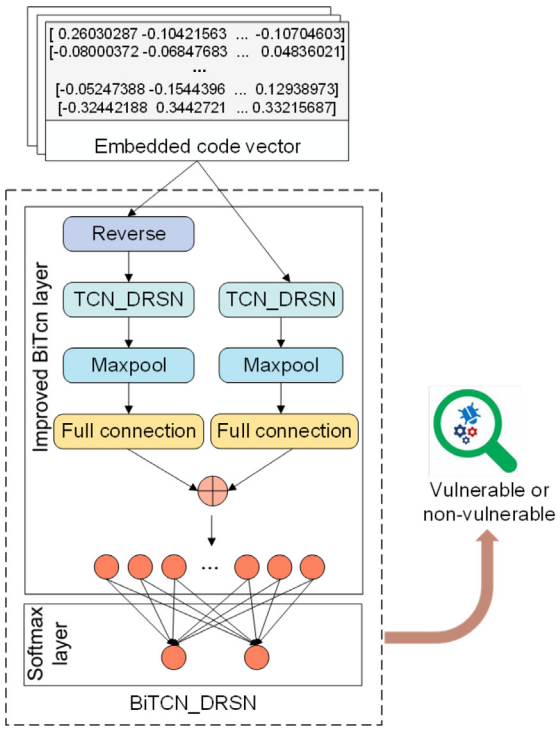


Fig. 4. The structure of BiTCN_DRSN.

where *Concatenate*(·) is the concatenation operation, that is splicing two features learned by TCN_DRSN. It is the combination of the $\overrightarrow{h}$ and $\overleftarrow{h}$ vectors into [$\overrightarrow{h}$, $\overleftarrow{h}$].

The BiTCN_DRSN model enables the final softmax layer input, which is used to identify whether or not a code slice contains vulnerabilities. This is done through consideration of the contextual information, which can enhance the source-code vulnerability identification.

3.3. Vulnerability detection model

The code slices in our study came from the VulDeepecker code gadgets (Li et al., 2018). They were generated by forward and backward analysis of the source code statements, and are related to the parameters passed when calling the library/API functions. The extracted source code statements are spliced in the order of statement execution, with the vulnerability of the code slices marked. The user-defined variable and function names in the token sequence are replaced with standard symbols.

Fig. 4 presents the overview structure of our proposed BiTCN_DRSN vulnerability-detection model. In the first step, the vector sequences of both the code slices (obtained with word2vec) and the reversed code slices are input into two TCN_DRSNs. Their outputs are then globally max pooled, separately. The pooled features are then nonlinearly varied using fully connected layers and the Relu activation function, with the outputs of the two fully connected layers being concatenated together. (The ⊕ symbol in Fig. 4 refers to the concatenation operation.) Finally, the concatenated result is fed into a softmax layer to identify whether or not the code slice has a vulnerability.

4. Experiment

This section reports on extensive experiments conducted on three datasets to answer the four research questions.

Table 1
The information of datasets.

Dataset	Vulnerable code slices	Non-vulnerable code slices
BE-ALL	10,440	29,313
RM-ALL	7,285	14,600
HY-ALL	17,725	43,913

4.1. Dataset description

We conducted the experiments on the VulDeepecker datasets,[2] which focus on the CWE-119 and CWE-399 vulnerabilities: CWE-119 is a buffer-error vulnerability and CWE-399 is a resource-management-error vulnerability. Table 1 shows the datasets' vulnerability distributions: The BE-ALL dataset contains code slices with CWE-119 vulnerabilities; RM-ALL has code slices with CWE-399, and HY-ALL has code slices with both CWE-119 and CWE-399. In the experiments, 80% of the code slices were selected as the training set, and the remaining 20% as the test set.

The main steps in generating the code slices relate to library/API function calls, and were divided into forward and backward calls: The forward function calls directly accepted one or more external inputs (e.g. from the command line, or from a file); and the backward function calls did not accept any external input. Forward and backward program slices (semantically interrelated lines of code) were generated corresponding to the parameters of the library/API function call extracted from the training source code. One or more slices were generated for each parameter. The forward slices correspond to statements affected by the relevant parameters, while the backward slices correspond to statements that could affect the relevant parameters. Finally, these program statements were arranged in order of execution, according to the relevant parameters, to obtain the slices.

4.2. Evaluation metrics

We evaluated the performance of our improved network model for vulnerability detection in terms of accuracy (A), false positive rate (FPR), false negative rate (FNR), precision (P) and F1-measure (F1) (Li et al., 2022a). Details of these five widely-used metrics are presented in Table 2, where *TP* is the number of vulnerable samples correctly detected; *FP* is the number of non-vulnerable samples incorrectly identified as vulnerable; *TN* is the number of non-vulnerable samples correctly identified as being without vulnerabilities; and *FN* is the number of vulnerable samples incorrectly identified as being without vulnerabilities. The higher the indicators of A, P and F1, the better. The lower the FNR and FPR indicators, the better.

4.3. Statistical analysis

The *p*-value and effect size were calculated as part of the statistical analyses. The Wilcoxon rank-sum test (Arcuri and Briand, 2014) was used to evaluate whether or not the difference between the proposed and the compared methods was significant, at the aggregated level, as measured by accuracy, precision, etc: A result (*p*-value) of less than 0.05 indicated a significant difference. The non-parametric Vargha and Delaney effect size (Arcuri and Briand, 2014) was adopted to indicate the probability that one method outperformed the other.

4.4. The set of parameters and running environments

The experiments were conducted on an NVIDIA GeForce RTX 3090 GPU with an Intel(R) Core(TM) i5-8400 CPU. The software

[2] https://github.com/CGCL-codes/VulDeePecker

Table 2
Evaluation metric.

Metric	Formula	Meaning
Accuracy	$A = \frac{TP+TN}{TP+FP+TN+FN}$	The proportion of correctly predicted samples.
FPR	$FPR = \frac{FP}{FP+TN}$	The number of samples incorrectly predicted as vulnerable, as a proportion of the actual non-vulnerable samples.
FNR	$FNR = \frac{FN}{TP+FN}$	The number of samples that are incorrectly predicted as not vulnerable, as a proportion of the actual vulnerable samples.
Precision	$P = \frac{TP}{TP+FP}$	The number of samples correctly detected as vulnerable, as a proportion of the total number of samples predicted as vulnerable.
F1-measure	$F1 = \frac{2*P(1-FNR)}{P+(1-FNR)}$	A metric combining both Precision and FNR.

Table 3
Tuned parameters for BiTCN_DRSN model.

Parameter	Description
Input dimension	Batch size × the maximum number of tokens in the code slices × the embedded length of a token (32 × 150 × 100)
Output dimension	Batch size × 2 (32 × 2)
TCN_DRSN units	The number of residual blocks was 5, the dilation was 1, 2, 4, 8, and 16.
Batch size	The number of samples that train through the network (32)
Epoch	The number of times for a complete training using all training data sets (10)
Loss function	The function to evaluate the extent to which the predicted value is different from the true value (binary_crossentropy)
Optimizer	Optimization algorithms for neural networks (Nadam)
Activation function	Functions of nonlinear transformations (relu)
Learning rate	The step or magnitude size of the model parameter update (0.002)
Dropout rate	The probability of randomly turning off a portion of neurons while training a neural network (0.05)

was written in Python 3.7. The neural network was constructed with the Keras framework in Tensorflow-gpu (version 2.6.0). The main BiTCN_DRSN parameters are listed in Table 3. The number of tokens in the vector representation of code slices was set to 150, and the vector length of each token embedding was 100. The number of residual blocks for TCN DRSN was set to 5, the batch size was set to 32, the number of epochs was set to 10, and optimizer was Nadam. All the experiments were repeated 20 times, with the average being recorded as the final result.

4.5. Experimental results

4.5.1. Answer to RQ1

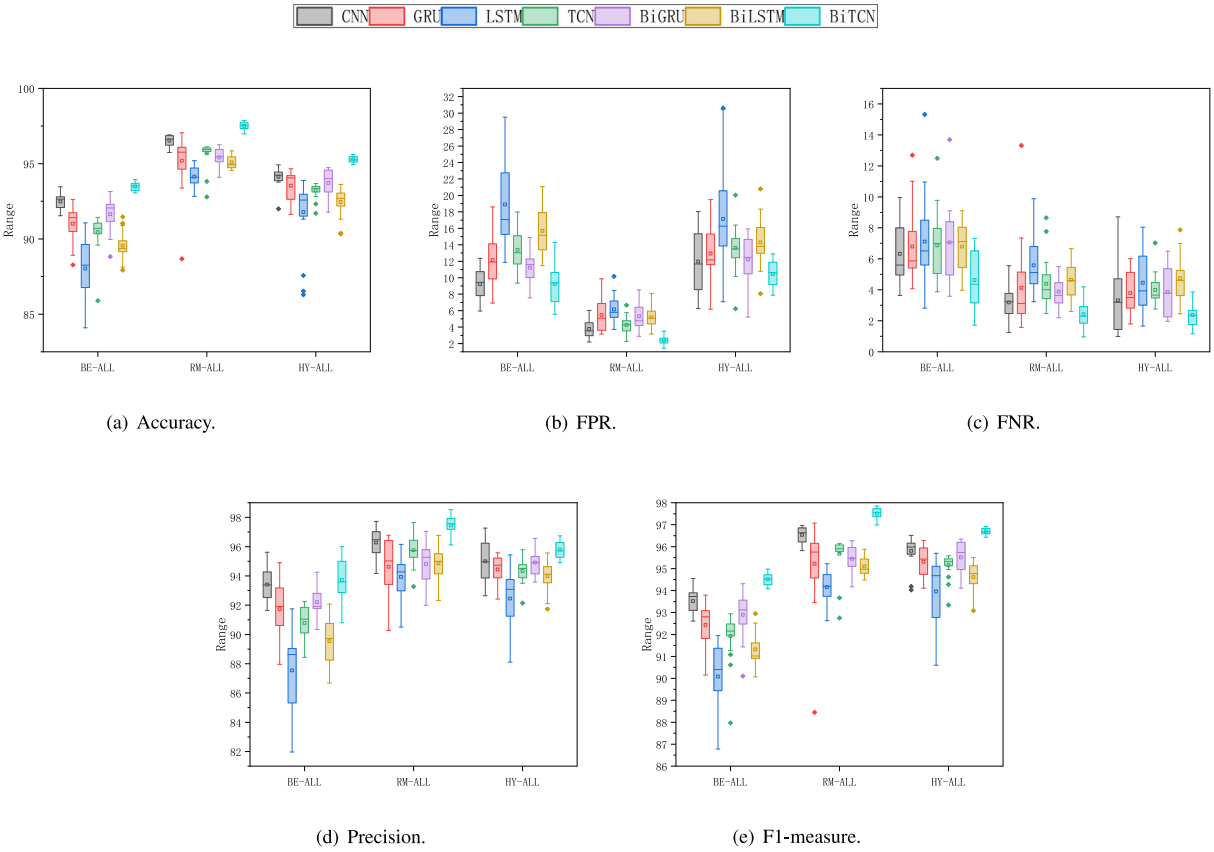
To answer RQ1 — about whether or not the bidirectional TCN network model has better detection capability and stability — we compared BiTCN with six neural network models: CNN, GRU, LSTM, TCN, BiGRU and BiLSTM. In the experiments, the concatenated CNN had four sequentially-connected CNNs. Each CNN had 16 filters, the sizes of which were 3, 4, 5, and 6. The experimental vulnerability-detection results (in terms of A, FPR, FNR, P, and F1) are shown in Table 4, and the stability of each model is shown in Fig. 5. The statistical analysis of the seven models compared with BiTCN_DRSN, on the three different datasets, is shown in Table 5.

Table 4 shows that, for the BE-ALL dataset, BiTCN has the best performance according to all five metrics. The accuracy, precision and F1-measure results for BiTCN are also the best for the RM-ALL and HY-ALL datasets, but the FNR and FPR results (for these two datasets) are less good, compared with the other deep learning models. The experimental results indicate that, compared with the unidirectional models, the bidirectional models could learn both the forward and reverse semantic features of the source code, which enhanced the vulnerability-detection. Compared with the CNN and RNN variants (GRU and LSTM), the TCN model flexibly expands its receptive field by increasing the dilation rate. It can also simultaneously update the weights at each time step for more efficient processing of the source-code features, which also improves the detection efficiency. In conclusion, the bidirectional TCN model can improve the vulnerability-detection capability.

Fig. 5 shows that BiTCN mostly outperforms the other six models, in terms of the accuracy, precision and F1-measure. Most of the BiTCN FPR and FNR results are also better than for the other six models. It can also be seen that there are no outliers in the BiTCN experimental data. The BiTCN interquartile ranges (the box length) seem smallest, indicating that the BiTCN detection efficiency is the most stable. In a comparison of the

Table 4
Vulnerability detection results for the seven different neural-network models.

Methods	Dataset	Accuracy (%)	FPR (%)	FNR (%)	Precision (%)	F1-measure (%)
CNN	BE-ALL	92.48	**9.22**	6.32	93.40	93.52
	RM-ALL	96.52	3.77	3.19	96.28	96.53
	HY-ALL	94.13	11.96	3.32	95.00	95.80
GRU	BE-ALL	91.01	12.15	6.79	91.73	92.43
	RM-ALL	95.20	5.47	4.13	94.62	95.22
	HY-ALL	93.53	12.93	3.79	94.44	95.31
LSTM	BE-ALL	88.02	18.89	7.11	87.54	90.08
	RM-ALL	94.14	6.14	5.58	93.93	94.16
	HY-ALL	91.78	17.13	4.45	92.45	93.96
TCN	BE-ALL	90.45	13.39	6.87	90.78	91.92
	RM-ALL	95.68	4.26	4.39	95.76	95.67
	HY-ALL	93.21	13.60	4.00	94.34	95.16
BiGRU	BE-ALL	91.31	11.03	7.04	92.22	92.55
	RM-ALL	95.41	5.30	3.88	94.80	95.45
	HY-ALL	93.71	12.23	3.86	94.9	95.51
BiLSTM	BE-ALL	89.56	15.68	6.79	89.54	91.31
	RM-ALL	95.15	5.13	4.56	94.92	95.17
	HY-ALL	92.45	14.29	4.74	93.98	94.61
BiTCN	BE-ALL	**93.48**	9.24	**4.63**	**93.72**	**94.51**
	RM-ALL	**97.50**	**2.58**	**2.42**	**97.44**	**97.50**
	HY-ALL	**95.27**	**10.45**	**2.37**	**95.79**	**96.70**



(a) Accuracy. (b) FPR. (c) FNR.

(d) Precision. (e) F1-measure.

Fig. 5. The stability of the seven different neural-network models.

corresponding unidirectional and bidirectional models (GRU and BiGRU; LSTM and BiLSTM; and TCN and BiTCN), the bidirectional models present a more stable effect. This could be because the bidirectional structure of the model can learn both forward and reverse semantic features of the source code, thereby improving the detection accuracy to some extent.

Table 5 shows the statistical comparison of BiTCN_DRSN against the seven other models, for the three datasets. Our proposed BiTCN_DRSN method is significantly different to the compared methods, for all five metrics, with the effect sizes showing it to be significantly better than the other methods.

Answer to RQ1:

The experiments show that the bidirectional neural network structure is able to learn forward and reverse semantic features from the source code, which can improve the vulnerability-detection performance. The BiTCN model has the best performance, which confirms that the bidirectional structure can improve the TCN detection ability.

Table 5
Statistical analysis of the seven different neural-network models compared with BiTCN_DRSN.

Methods	Dataset	Accuracy p-value/effect_size	FPR p-value/effect_size	FNR p-value/effect_size	Precision p-value/effect_size	F1-measure p-value/effect_size
CNN	BE-ALL	0.00/1.00	0.05/0.30	0.00/0.15	0.01/0.77	0.00/0.99
	RM-ALL	0.00/1.00	0.00/0.02	0.00/0.01	0.00/0.99	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.03	0.42/0.43	0.01/0.74	0.00/0.84
GRU	BE-ALL	0.00/1.00	0.00/0.12	0.00/0.09	0.00/0.92	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.00	0.00/0.07	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.08	0.00/0.20	0.00/0.94	0.00/0.91
LSTM	BE-ALL	0.00/1.00	0.00/0.00	0.00/0.17	0.00/1.00	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.00	0.00/0.00	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.05	0.00/0.15	0.00/0.98	0.00/0.97
TCN	BE-ALL	0.00/1.00	0.00/0.01	0.00/0.15	0.00/1.00	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.01	0.00/0.01	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.04	0.00/0.12	0.00/0.94	0.00/0.90
BiGRU	BE-ALL	0.00/1.00	0.00/0.10	0.00/0.20	0.00/0.95	0.00/0.99
	RM-ALL	0.00/1.00	0.00/0.00	0.00/0.01	0.00/1.00	0.01/1.00
	HY-ALL	0.00/1.00	0.00/0.11	0.00/0.23	0.00/0.87	0.00/0.89
BiLSTM	BE-ALL	0.00/1.00	0.00/0.00	0.00/0.11	0.00/1.00	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.00	0.00/0.00	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.03	0.00/0.08	0.00/0.95	0.00/0.98
BiTCN	BE-ALL	0.00/1.00	0.31/0.41	0.17/0.37	0.31/0.60	0.00/0.93
	RM-ALL	0.00/1.00	0.00/0.20	0.00/0.22	0.00/0.81	0.00/1.00
	HY-ALL	0.00/0.95	0.00/0.20	0.58/0.45	0.02/0.71	0.00/0.85

Table 6
Effectiveness of TCN_Attention vs. TCN vs. BiTCN_Attention vs. TCN_DRSN vs. BiTCN_DRSN.

Methods	Dataset	Accuracy	FPR	FNR	Precision	F1-measure
TCN	BE-ALL	90.45	13.39	6.87	90.78	91.92
	RM-ALL	95.68	4.26	4.39	95.76	95.67
	HY-ALL	93.21	13.60	4.00	94.34	95.16
TCN_Attention	BE-ALL	92.79	11.17	4.45	92.51	93.99
	RM-ALL	97.26	2.64	2.62	97.37	97.37
	HY-ALL	94.25	11.43	3.41	95.44	95.98
BiTCN_Attention	BE-ALL	92.04	10.97	5.86	92.52	93.30
	RM-ALL	93.30	3.29	3.02	96.72	96.85
	HY-ALL	94.50	11.41	3.08	95.40	96.15
TCN_DRSN	BE-ALL	93.18	10.41	**4.33**	92.99	94.29
	RM-ALL	97.54	2.42	2.50	97.59	97.54
	HY-ALL	95.28	10.60	2.30	95.74	96.70
BiTCN_DRSN	BE-ALL	**94.67**	**7.39**	4.52	**94.75**	**95.37**
	RM-ALL	**98.10**	**1.82**	**1.72**	**98.11**	**98.05**
	HY-ALL	**95.87**	**8.05**	**2.47**	**96.15**	**96.83**

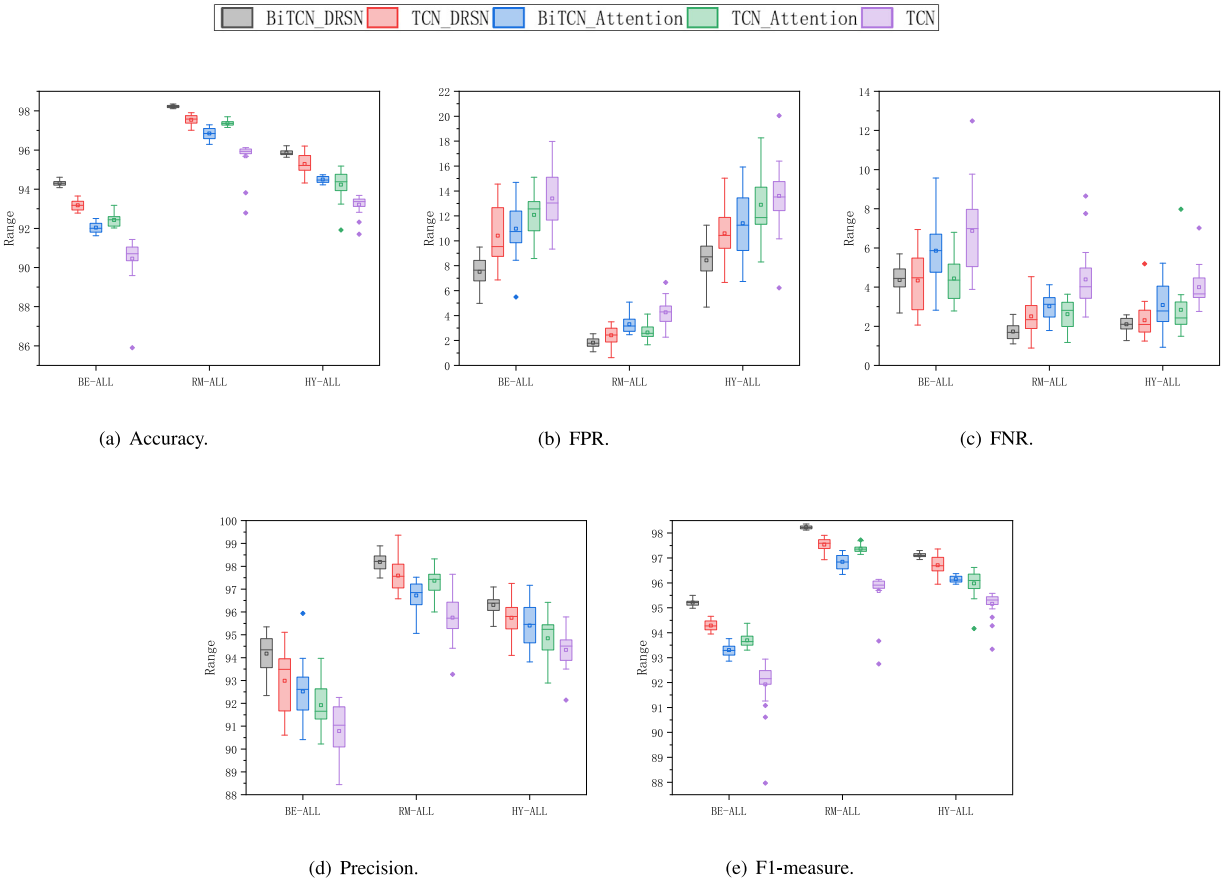
4.5.2. Answer to RQ2

To answer RQ2 — about whether or not deep residual networks improve on TCN network models' vulnerability detection — in this section, we compare the proposed BiTCN_DRSN model with TCN, TCN_Attention (also called AIdetectorX (Chen et al., 2021)), BiTCN_Attention (a combination of BiTCN and the self-attention mechanism), and TCN_DRSN (a combination of TCN and a Deep Residual Network). The comparison again involved the three datasets, and aimed to verify that deep residual networks could improve on the TCN vulnerability-detection ability. The experimental setup and parameters of the model remained the same as in Table 3. Again, the experiments were repeated 20 times, with the average being shown in Table 6.

Table 6 shows that TCN_DRSN outperforms the original TCN, on all three datasets, according to all five metrics. Compared with the original TCN model on the BE-ALL dataset, TCN_DRSN shows improvements of 2.73%, 2.21%, 2.37%, 2.98%, and 2.54%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. The experimental results show that the vulnerability-detection capability of the TCN_DRSN model is enhanced relative to the traditional TCN model, providing further evidence to support the effectiveness of the combination of DRSN and TCN for vulnerability detection. The use of DRSN can effectively reduce the impact of source-code features that are not related to vulnerabilities, which results in the neural network focusing more on the vulnerability-related features. The advanced features learned by the neural network can have better discrimination power.

TCN_DRSN outperforms TCN_Attention, on all three datasets, according to all five metrics. Compared with TCN_Attention on the BE-ALL dataset, TCN_DRSN shows improvements of 0.39%, 0.48%, 0.3%, 0.76%, and 0.12%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. On the RM-ALL dataset, TCN_DRSN shows improvements of 0.28%, 0.22%, 0.17%, 0.22%, and 0.12%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. Finally, on the HY-ALL dataset, TCN_DRSN shows improvements of 1.03%, 0.3%, 0.72%, 0.83%, and 1.11%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. TCN_DRSN has a certain improvement effect on all three datasets, which confirms that the combination of DRSN and TCN is more effective than a combination with the attention mechanism. Similarly, in the BiTCN model, the combination of DRSN is also more effective than the combination with the self-attention mechanism. Therefore, we can conclude that deep residual networks can further improve the vulnerability-detection capability of TCN models.

Fig. 6(a) shows that the detection accuracy of the proposed BiTCN_DRSN model is the highest. It also shows that the BiTCN_DRSN interquartile range (the box length) is the smallest, and that BiTCN_DRSN has no outliers. Figs. 6(b)–(e) show that the BiTCN_DRSN interquartile ranges for the FPR, FNR, precision and F1-measure metrics are also the smallest. This indicates that BiTCN_DRSN is the most stable of the five compared models. The BiTCN_DRSN interquartile ranges are smaller than BiTCN_Attention's, and BiTCN_DRSN has no outliers while



(a) Accuracy. (b) FPR. (c) FNR.

(d) Precision. (e) F1-measure.

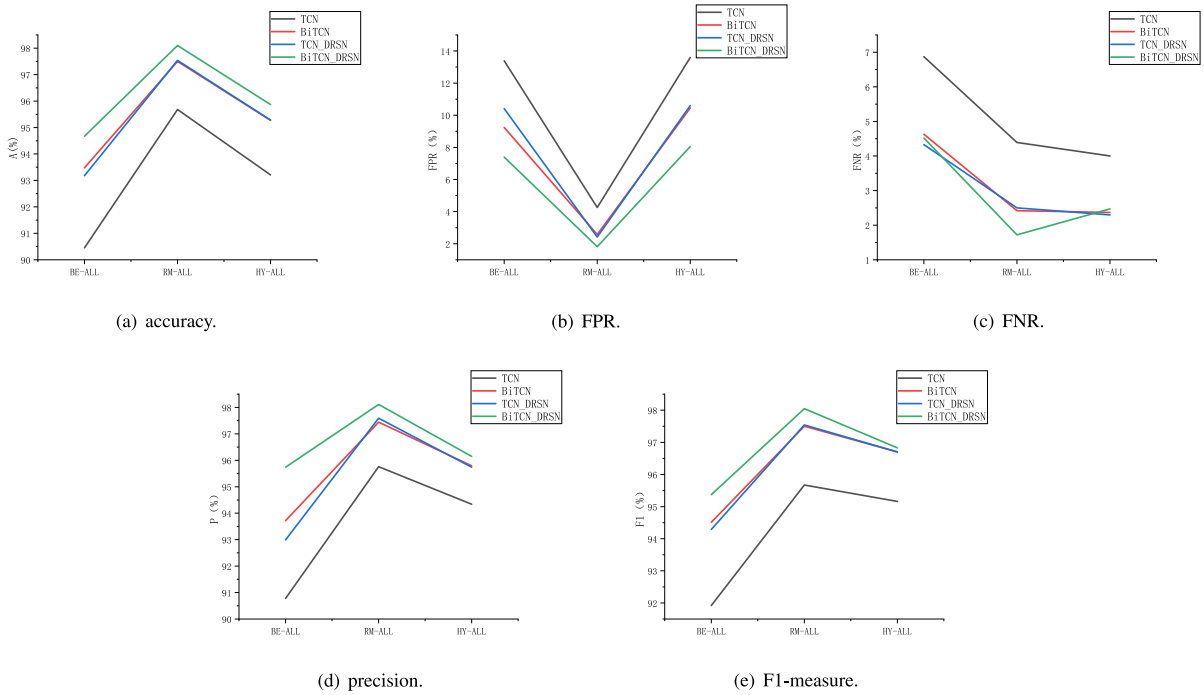
Fig. 6. The stability of TCN, TCN_Attention, BiTCN_Attention, TCN_DRSN and BiTCN_DRSN.

Table 7
Statistical analysis of TCN, TCN_Attention, BiTCN_Attention, TCN_DRSN and BiTCN_DRSN.

Methods	Dataset	Accuracy p-value/effect_size	FPR p-value/effect_size	FNR p-value/effect_size	Precision p-value/effect_size	F1-measure p-value/effect_size
TCN	BE-ALL	0.00/1.00	0.00/0.01	0.00/0.15	0.00/1.00	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.01	0.00/0.01	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.04	0.00/0.12	0.00/0.94	0.00/0.90
TCN_Attention	BE-ALL	0.00/1.00	0.00/0.06	0.13/0.36	0.00/0.95	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.12	0.00/0.19	0.00/0.89	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.06	0.15/0.37	0.00/0.91	0.00/0.87
BiTCN_Attention	BE-ALL	0.00/1.00	0.00/0.15	0.00/0.17	0.00/0.88	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.01	0.00/0.05	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.18	0.01/0.26	0.00/0.76	0.00/0.85
TCN_DRSN	BE-ALL	0.00/1.00	0.02/0.28	0.31/0.41	0.01/0.74	0.00/0.96
	RM-ALL	0.00/1.00	0.00/0.23	0.00/0.24	0.00/0.78	0.00/1.00
	HY-ALL	0.00/0.82	0.00/0.19	0.67/0.54	0.02/0.71	0.02/0.71

BiTCN_Attention has a few: This indicates that the DRSN model can improve the stability of the detection. The TCN_DRSN interquartile ranges are very large, even though it has no outliers for any of the five metrics; BiTCN_DRSN also has no outliers, but has a small interquartile range: This also confirms that the use of the bidirectional structure can improve the stability. The experimental results on stability confirm that the use of the bidirectional structure and DRSN can effectively improve the stability of the vulnerability-detection model.

Table 7 presents the statistical analysis of the four models (TCN, TCN_Attention, BiTCN_Attention and TCN_DRSN) compared with BiTCN_DRSN. For accuracy, BiTCN_DRSN shows significant differences compared with all four models on all three datasets (p-values < 0.05). The effect size indicates that BiTCN_DRSN has the best accuracy. In term of FPR, BiTCN_DRSN is also significantly different compared with the other four models (p-values < 0.05). The effect sizes are also all less than 0.5, which indicates that BiTCN_DRSN typically has better FPR performance than the other models. The BiTCN_DRSN FNR is different to the compared models in most cases, but the difference is not significant in some cases: TCN_Attention on BE-ALL (p-values $= 0.13$) and HY-ALL (p-values $= 0.15$); and TCN_DRSN on BE-ALL (p-values $= 0.31$) and HY-ALL (p-values $= 0.67$). However, the effect size results indicate that BiTCN_DRSN has better FNR than the others (except for TCN_DRSN on HY-ALL (effect size $= 0.54$)). The BiTCN_DRSN precision shows significant differences compared with the other models, with the effect size results confirming BiTCN_DRSN's better precision. Likewise, we can draw the same conclusion for the F1-measure results according to Table 7 (all the p-values are less than 0.05, and all the effect size values are larger than 0.5).



(a) accuracy. (b) FPR. (c) FNR.

(d) precision. (e) F1-measure.

Fig. 7. Results of ablation experiments.

Answer to RQ2:

The experimental results show that the combination of the deep residual network and TCN is superior to the combination of TCN and the self-attention mechanism. They also show that the performance of BiTCN_DRSN model is even better.

4.5.3. Answer to RQ3

To answer RQ3 — about whether or not the combination of the two improvements to the TCN network architecture can deliver an enhancement — we conducted a series of ablation experiments: Fig. 7 presents the results.

For accuracy, precision and the F1-measure, the BiTCN and TCN_DRSN lines are above the TCN line, meaning better performance. This shows that the two improvement methods for TCN can improve the accuracy, precision and F1-measure of the traditional TCNs model for vulnerability detection. The BiTCN_DRSN line is above the TCN_DRSN and BiTCN lines (for accuracy, precision and F1-measure), which shows that the combination of these two methods can further improve the performance of traditional TCNs, without conflict. For FNR and FPR, BiTCN_DRSN is at the bottom (indicating best performance), and the BiTCN and TCN_DRSN lines are in the middle, which also reflects the effectiveness of the two improvements for TCN, and that the improvements do not conflict.

Answer to RQ3:

The experiments show that the combination of the two improved methods does not conflict, and can further improve the TCN vulnerability-detection ability.

4.5.4. Answer to RQ4

To answer RQ4 — about whether or not the proposed BiTCN_DRSN model outperforms other models in vulnerability-detection capability and stability — we compared BiTCN_DRSN with five state-of-the-art vulnerability detectors: SySeVR (Li et al., 2022b); VulDeeLocator (Li et al., 2022a); ReVeal (Chakraborty et al., 2021); VulDeePecker (Li et al., 2018); and AIdetectorX (Chen et al., 2021). The parameter settings for these tools were consistent with those reported in the original papers. The SySeVR's Library/API Function Call sliced dataset (FC)[3] was used with the SySeVR method. Because previous experiments indicated that the number of tokens had some effect on the experimental results, therefore, our token-embedding settings were consistent with SyseVR in this comparison experiment. For the VulDeeLocator method, the slicing criterion of the used dataset was similar to that of SySeVR, and the dataset was downloaded from Github.[4] The BE-ALL, RM-ALL and HY-ALL datasets were used for ReVeal, VulDeePecker and AIdetectorX. The experimental results are shown in Tables 8–10, and the stability of these four compared models is shown in Fig. 8.

Table 8 presents the experimental results comparing BiTCN_DRSN with SySeVR. BiTCN_DRSN shows improvement over SySeVR of 0.65%, 5.53%, 0.68%, 5.32%, and 0.52%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. The *p*-value results show that BiTCN_DRSN is significantly different from SySeVR (except for FNR = 0.60), and is better than SySeVR (where the effect size of accuracy, precision, and F1-measure is greater than 0.5). Compared with SySeVR, BiTCN_DRSN has a bidirectional network structure, which can better capture the semantic information of the source code, and thus improve the vulnerability-detection capability.

Table 9 shows that, compared with VulDeeLocator, BiTCN_DRSN has better accuracy, precision, F1-measure, FPR, and FNR. BiTCN_DRSN shows improvement over VulDeeLocator of 5.76%, 0.5%, 20.18%, 0.05%, and 27.45%, in terms of accuracy, precision, F1-measure, FPR, and FNR, respectively. The *p*-value results show that BiTCN_DRSN is significantly different from VulDeeLocator (except for FPR = 0.65), and the effect size results confirm

[3] https://github.com/SySeVR/SySeVR

[4] https://github.com/VulDeeLocator/VulDeeLocator/tree/master/data

Table 8
Comparative experimental results of the effectiveness of BiTCN_DRSN and SySeVR.

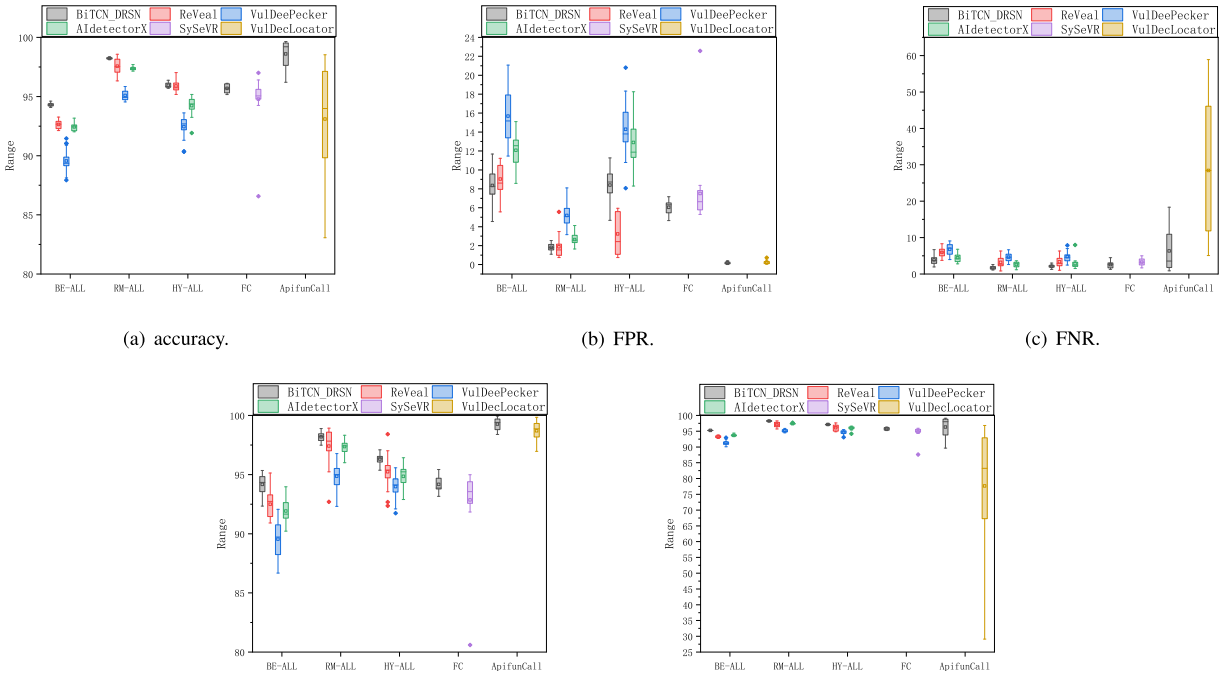
Methods	Accuracy	FPR	FNR	Precision	F1-measure
BiTCN_DRSN	95.78	5.59	2.87	94.68	95.88
SySeVR	95.13	10.91	3.39	89.15	95.21
Statistical Analysis (p-value/effect size)	0.01/0.84	0.02/0.19	0.60/0.43	0.00/1.00	0.01/0.85

Table 9
Comparative experimental results of the effectiveness of BiTCN_DRSN and VulDeeLocator.

Methods	Accuracy	FPR	FNR	Precision	F1-measure
BiTCN_DRSN	98.77	0.17	6.31	99.27	96.32
VulDeeLocator	92.84	0.22	33.76	98.77	76.14
Statistical Analysis (p-value/effect size)	0.00/0.92	0.65/0.45	0.00/0.10	0.04/0.70	0.00/0.91

Table 10
Comparative experimental results of the effectiveness of ReVeal, VulDeePecker, AIdetectorX and BiTCN_DRSN.

Methods	Dataset	Accuracy	FPR	FNR	Precision	F1-measure
BiTCN_DRSN	BE-ALL	94.67	7.39	4.52	94.75	95.37
	RM-ALL	98.10	1.82	1.72	98.11	98.05
	HY-ALL	95.87	8.05	2.47	96.15	96.83
ReVeal	BE-ALL	92.62	9.03	5.81	92.36	93.25
	RM-ALL	97.46	2.78	3.20	97.15	96.95
	HY-ALL	95.79	9.64	4.60	95.46	96.03
BiLSTM	BE-ALL	89.56	15.68	6.79	89.54	91.31
	RM-ALL	95.15	5.13	4.56	94.92	95.17
	HY-ALL	92.45	14.29	4.74	93.98	94.61
AIdetectorX	BE-ALL	92.79	11.17	4.45	92.51	93.99
	RM-ALL	97.26	2.64	2.62	97.37	97.37
	HY-ALL	94.25	11.43	3.41	95.44	95.98



(a) accuracy. (b) FPR. (c) FNR.

(d) precision. (e) F1-measure.

Fig. 8. Stability of SySeVR, VulDeeLocator, VulDeePecker, AIdetectorX and BiTCN_DRSN.

that BiTCN_DRSN is better: The effect size for accuracy, precision and F1-measure are greater than 0.5; and less than 0.5 for FPR and FNR. Compared with VulDeeLocator, BiTCN_DRSN combines a deep residual shrinkage network to improve the learning efficiency on both noisy and complex data, which results in BiTCN_DRSN outperforming VulDeeLocator in terms of detection results.

It can be seen from Table 10 that, compared with ReVeal, VulDeePecker and AIdetectorX, on all three datasets, BiTCN_DRSN outperforms in terms of accuracy, precision, F1-measure, FPR, and FNR. According to the statistical results shown in Table 11, BiTCN_DRSN is significantly different, for all five metrics, across all three datasets, compared with the other three vulnerability-detection models. The effect size results also show

Table 11
Statistical analysis of ReVeal, VulDeePecker, AIdetectorX and BiTCN_DRSN.

Methods	Dataset	Accuracy p-value/effect_size	FPR p-value/effect_size	FNR p-value/effect_size	Precision p-value/effect_size	F1-measure p-value/effect_size
ReVeal	BE-ALL	0.00/ 1.00	0.24/ 0.39	0.00/ 0.12	0.00/ 0.88	0.00/ 1.00
	RM-ALL	0.00/ 0.79	0.32/ 0.59	0.00/ 0.19	0.23/ 0.61	0.00/ 0.94
	HY-ALL	0.02/ 0.28	0.00/ 0.95	0.02/ 0.29	0.84/ 0.48	0.00/ 0.80
VulDeePecker	BE-ALL	0.00/1.00	0.00/0.00	0.00/0.11	0.00/1.00	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.00	0.00/0.00	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.03	0.00/0.08	0.00/0.95	0.00/0.98
AIdetectorX	BE-ALL	0.00/1.00	0.00/0.15	0.00/0.17	0.00/0.88	0.00/1.00
	RM-ALL	0.00/1.00	0.00/0.01	0.00/0.05	0.00/1.00	0.00/1.00
	HY-ALL	0.00/1.00	0.00/0.18	0.01/0.26	0.00/0.76	0.00/0.85

that the proposed BiTCN_DRSN model has a certain improvement in the detection ability.

Fig. 8 shows that BiTCN_DRSN performs best in terms of stability, across all datasets: It has the smallest interquartile range (box length) and no outlier points. Compared with the other five models, BiTCN_DRSN uses the DRSN model to reduce the impact of vulnerability-irrelevant source-code statements and other redundant information on software-vulnerability detection, which can improve the detection stability of the model.

Answer to RQ4:

The experimental results show that the proposed BiTCN_DRSN model has better detection capability and stability compared to the other vulnerability detectors.

4.6. Threats to validity

The experimental results confirm the effectiveness and feasibility of the proposed BiTCN_DRSN model. This section examines some potential threats to the validity of our study, both in terms of internal and external validity.

4.6.1. Threats to internal validity

The performance of the BiTCN_DRSN model depends on the number of tokens to be processed. In our experiments on the VulDeePecker datasets, we found that when the number of tokens was 50 or 100, the BiTCN_DRSN performance was poorer than when the number of tokens was set to 150. When the number of tokens was set to 200, however, there was no significant performance enhancement. The reason for this is that most of the generated code slices in the VulDeePecker datasets contain less than 150 tokens: Accordingly, the number of tokens in each code slice was set to 150 in our proposed model. Nevertheless, identification of the most appropriate cut-off length still requires further study.

A second potential threat relates to the model's hyperparameter settings (such as the number of residual blocks in the BiTCN_DRSN model, the number and size of convolution kernels, etc.), which can influence the experimental results. Our hyperparameters' settings were determined through experimentation. However, our model needs more fine-tuning of the hyperparameters, which we will address in our future work.

Finally, because Li et al. (2018) have not made their source code publicly available, the model compared in the experiment is based on the code reproduced by others.[5] In our experiments, using our reproduction models for situations where the original code was not available affected the comparison tests. We chose the ReVeal model as an example deep-learning-based vulnerability-detection method. However, because ReVeal is not an open source project, we recreated it ourselves, and processed the dataset into the form required for our comparison experiments.

4.6.2. Threats to external validity

We evaluated our model on the datasets of VulDeepecker, SySeVR and VulDeeLocator. However, these datasets only represent a portion of all possible vulnerabilities. Furthermore, because our model was designed for C/C++ source code, its performance with other programming languages is unknown. Finally, due to the significant differences in code complexity, the detection results in real-world scenarios may be less satisfactory. When applied to more complex vulnerabilities or other languages (e.g., Java), the results may differ. Nevertheless, our approach is generic and can be extended to other vulnerabilities and languages.

5. Conclusion and future work

5.1. Conclusion

With the growth of software and the rising popularity of open-source software, the need for detecting vulnerabilities in source code has become more crucial than ever. Our model is designed to detect vulnerabilities in open-source code, which can effectively enhance software security. Addressing inadequacies of TCNs for code-vulnerability detection, this paper has proposed the BiTCN_DRSN model to improve vulnerability-detection effectiveness. For source-code statements that are not relevant to the vulnerability (i.e., the noise interference), BiTCN_DRSN leverages a DRSN to reduce the focus on features in the code slices that are not associated with vulnerability information. The DRSN achieves this through soft thresholding: It can focus on the features related to the vulnerability information and retain them. Its soft thresholding is more flexible as its shrinkage threshold is arrived at by a neural network. BiTCN_DRSN targets the limitations of TCN unidirectionality in text-information capture with nonlinear fusion of features learned from both forward and backward sequences: This enables exploitation of the information from both directions for vulnerability detection. Our proposed BiTCN_DRSN model not only focuses more on the source-code features related to vulnerabilities, but also addresses the lack of learning ability of the TCN unidirectional structure for source-code bidirectional structures.

We evaluated our model on VulDeepecker, SySeVR and VulDeeLocator datasets. As shown in Table 6, compared with the traditional TCN, BiTCN_DRSN improved the precision by 4.22%, 2.42% and 2.66% on the BE-ALL, RM-ALL and HY-ALL datasets, respectively. Our experiments have shown that both of our proposed improvements for TCNs are effective and conflict-free. In addition, the experimental results show that our proposed BiTCN_DRSN vulnerability-detection model outperforms existing models such as SySeVR, VulDeeLocator, ReVeal, VulDeePecker, and AIdetectorX.

[5] https://github.com/johnb110/VDPython

Our model is currently only used for C/C++ source code, and the datasets (obviously) do not include all possible vulnerability types. This means that the results achieved in our experiments may not be the same for more complex vulnerabilities or with other programming languages (e.g. Java). This is something that we plan to address in our future work.

5.2. Future work

This paper has explored and investigated the feature-engineering and neural-network model aspects of vulnerability-detection tasks. We have proposed the BiTCN_DRSN model to improve on TCN code-vulnerability detection. Our future work and research will include the following:

1. The datasets used in our experiments do not represent all types of software applications or vulnerabilities. Therefore, more code-vulnerability types will be collected to enrich the existing datasets in the future.
2. Our proposed method currently only works with C/C++ source code: Its vulnerability-detection performance with other programming languages is unknown. We look forward to verifying whether or not the method can effectively detect vulnerabilities in source code written in other programming languages, especially in Java or python, two programming languages commonly used in enterprise projects.
3. Our proposed method can detect whether or not the code fragments produced by the slicing technique are vulnerable, but it does not determine the *type* of the vulnerability. This is a common challenge in source-code-based vulnerability detection. Determining the vulnerability type is a worthwhile research direction: Knowing the vulnerability type may help the code auditor or developer to understand the underlying cause, and thereby identify the statements that may trigger the vulnerability in the already-narrowed code scope.
4. The detection granularity of existing deep-learning-based vulnerability-detection methods is either at the file or function level. Both of those levels may contain a large number of code statements. Even though the slicing level in this study can reduce the number of irrelevant statements, the resulting code fragments may still involve a large amount of code. This is especially the case if it is a sensitive variable that runs throughout the program. Therefore code-vulnerability identification at a finer granularity level is an attractive direction for further research.

CRediT authorship contribution statement

Jinfu Chen: Investigation, Methodology, Writing – original draft. **Wei Lin:** Methodology, Writing – original draft. **Saihua Cai:** Software, Validation, Supervision, Writing – review & editing. **Yemin Yin:** Data curation, Visualization. **Haibo Chen:** Investigation, Data curation. **Dave Towey:** Validation, 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.

Data availability

Data will be made available on request.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China (NSFC) (Grant nos. 62172194, 62202206 and U1836116), the National Key R and D Program of China (Grant no. 2020YFB1005500), the Natural Science Foundation of Jiangsu Province, China (Grant no. BK20220515), the Leading-edge Technology Program of Jiangsu Natural Science Foundation, China (Grant no. BK20202001), the China Postdoctoral Science Foundation, China (Grant no. 2021M691310), and Qinglan Project of Jiangsu Province, China.

References

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.

Bai, S., Kolter, J.Z., Koltun, V., 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. CoRR abs/1803.01271.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, pp. 1456–1468, arXiv preprint arXiv:2203.02660.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng. 48 (9), 3280–3296.

Chen, J., Kudjo, P.K., Mensah, S., Brown, S.A., Akorfu, G., 2020. An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection. Journal of Systems and Software 167, 110616.

Chen, J., Liu, B., Cai, S., Wang, W., Wang, S., 2021. AIdetectorx: A vulnerability detector based on TCN and self-attention mechanism. In: Dependable Software Engineering. Theories, Tools, and Applications: 7th International Symposium, SETTA 2021, Beijing, China, November 25–27, 2021, Proceedings 7. Springer, pp. 161–177.

Cheng, Y., Xu, Y., Zhong, H., Liu, Y., 2021. Leveraging semisupervised hierarchical stacking temporal convolutional network for anomaly detection in IoT communication. IEEE Internet Things J. 8 (1), 144–155.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the Empiricial Methods in Natural Language Processing. EMNLP, ACL, Association for Computational Linguistics, Doha, Qatar, pp. 1724–1734.

Daubechies, I., DeVore, R., Foucart, S., Hanin, B., Petrova, G., 2022. Nonlinear approximation and (deep) ReLU networks. Constr. Approx. 55 (1), 127–172.

Fang, Z., Liu, Q., Zhang, Y., Wang, K., Wang, Z., Wu, Q., 2017. A static technique for detecting input validation vulnerabilities in android apps. Sci. China Inf. Sci. 60 (5), 1–16.

Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L., 2016. Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. ACM, New York, New York, USA, pp. 85–96.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Identity mappings in deep residual networks. In: European Conference on Computer Vision, Vol. 9908. Springer, Amsterdam, The Netherlands, pp. 630–645.

Hochreiter, S., 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int. J. Uncertain. Fuzziness Knowl. Based Syst. 6 (2), 107–116.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Huang, Q., Hain, T., 2021. Improving audio anomalies recognition using temporal convolutional attention networks. In: IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP, IEEE, Toronto, Ontario, Canada, pp. 6473–6477.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In: 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice. SER&IP, IEEE, San Jose, California, USA, pp. 595–614.

Kuang, L., Hua, C., Wu, J., Yin, Y., Gao, H., 2020. Traffic volume prediction based on multi-sources GPS trajectory data by temporal convolutional network. Mobile Netw. Appl. 25 (4), 1405–1417.

Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2022a. VulDeeLocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans. Dependable Secure Comput. 19 (4), 2821–2837.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2022b. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure 19 (4), 2244–2258.
Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium. ISOC, San Diego, California, USA, pp. 1–15.
Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans. Dependable Secure Comput. 18 (5), 2469–2485.
Liu, J., Chen, L., Dong, L., Wang, J., 2012. UCBench: A user-centric benchmark suite for c code static analyzers. In: 2012 IEEE International Conference on Information Science and Technology. IEEE, pp. 230–237.
Liu, S., Lin, G., Han, Q.-L., Wen, S., Zhang, J., Xiang, Y., 2019. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans. Fuzzy Syst. 28 (7), 1329–1343.
Novak, M., Joy, M., Kermek, D., 2019. Source-code similarity detection and detection tools used in academia: a systematic review. ACM Trans. Comput. Educ. (TOCE) 19 (3), 1–37.
Pascanu, R., Mikolov, T., Bengio, Y., 2013. On the difficulty of training recurrent neural networks. In: Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013. In: JMLR Workshop and Conference Proceedings, vol. 28, JMLR.org, pp. 1310–1318.
Qiu, X., 2020. Neural Networks and Deep Learning. China Machine Press, Beijing, China, ISBN 978-7-111-64968-7.
Reynolds, Z.P., Jayanth, A.B., Koc, U., Porter, A.A., Raje, R.R., Hill, J.H., 2017. Identifying and documenting false positive patterns generated by static code analysis tools. In: 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice. SER&IP, IEEE, Buenos Aires, Argentina, pp. 55–61.
Shi, H., Wang, R., Fu, Y., Jiang, Y., Dong, J., Tang, K., Sun, J., 2019. Vulnerable code clone detection for operating system through correlation-induced learning. IEEE Trans. Ind. Inform. 15 (12), 6551–6559.
skyboxsecurity, 2021. Vulnerability and threat trends mid-year report 2021. https://www.skyboxsecurity.com/trends-report/.
Wang, J., Huang, M., Nie, Y., Li, J., 2021. Static analysis of source code vulnerability using machine learning techniques: A survey. In: 2021 4th International Conference on Artificial Intelligence and Big Data. ICAIBD, IEEE, pp. 76–86.
Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H., 2022. VulCNN: An image-inspired scalable vulnerability detection system. In: 44th IEEE/ACM 44th International Conference on Software Engineering,ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, pp. 2365–2376.
Younis, A., Malaiya, Y., Anderson, C., Ray, I., 2016. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. ACM, New York, New York, USA, pp. 97–104.
Zhang, H., 2019. Vulnerability Detection Based on Deep Learning. Huazhong University of Science and Technology, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China (in Chinese).
Zhao, M., Zhong, S., Fu, X., Tang, B., Pecht, M., 2020. Deep residual shrinkage networks for fault diagnosis. IEEE Trans. Ind. Inform. 16 (7), 4681–4690.
Zheng, Z., Li, C., Liu, Y., Xi, Z., 2022. A phase-type expansion approach for the performability of composite web services. IEEE Trans. Reliab. 71 (2), 579–589.
Zheng, Z., Trivedi, K.S., Wang, N., Qiu, K., 2017. Markov regenerative models of webservers for their user-perceived availability and bottlenecks. IEEE Trans. Dependable Secure Comput. 17 (1), 92–105.
Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems. MIT Press, Vancouver, British Columbia, Canada, pp. 8–14.
Zhu, C., Tang, Y., Wang, Q., Li, M., 2019. Enhancing code similarity analysis for effective vulnerability detection. In: Proceedings of the 2nd International Conference on Computer Science and Software Engineering. ACM, New York, New York, USA, pp. 153–158.

Jinfu Chen received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology, Wuhan, China, in 2009. He is currently a full professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software security, and trusted software. He has published more than 80 papers in some famous journals or conferences, including in IEEE Transactions on Reliability, Information Sciences, Journal of Systems and Software, Information and Software Technology, Software: Practice and Experience, IET Software, The Computer Journal, ISSTA, ASE, ISSRE, and QRS. He is a member of the IEEE and the ACM, and a member of the China Computer Federation.

Wei Lin received the B.E. and Master's degrees, in Computer Science and Technology, from Jiangsu University, Zhenjiang, China, in 2019 and 2022, respectively. Her research interests include vulnerability detection.

Saihua Cai received his Ph.D. degree in Agricultural Engineering from China Agricultural University, Beijing, China, in 2020. He is currently a lecturer in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include outlier detection, software testing, and network traffic detection. He has published more than 50 papers in journals or conferences, including in Information Sciences, The Computer Journal, Knowledge-Based Systems, ISSRE, and QRS. He is a member of the IEEE and the China Computer Federation.

Yemin Yin received the B.E. degree in 2020 in Information Security from Jiangsu University in Zhenjiang, China, where he is currently working towards the Master's degree in Computer Science and Technology. His research interests include vulnerability detection.

Haibo Chen received the B.E. degree in 2018 from Changzhou Institute of Technology, Changzhou, China; and Master's degree in 2021 from Jiangsu University, Zhenjiang, China: Both in Computer Science and Technology. He is currently working towards the Ph.D. in Computer Science and Technology, at Jiangsu University, Zhenjiang, China. His current research interests include fuzzing and adaptive random testing.

Dave Towey received the B.A. and M.A. degrees in computer science, linguistics, and languages, from the University of Dublin, Trinity College, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from The University of Hong Kong, China. He is a full professor at University of Nottingham Ningbo China (UNNC), in Zhejiang, China, where he serves as the associate dean of education and student experience. He also serves as the deputy head of the School of Computer Science, and the deputy director of the International Doctoral Innovation Center. He is a member of the UNNC Artificial Intelligence and Optimization research group. His current research interests include software testing (especially adaptive random testing, for which he was amongst the earliest researchers who established the field, and metamorphic testing), computer security, and technology-enhanced education. He co-founded the ICSE International Workshop on Metamorphic Testing in 2016. He is a fellow of the HEA, and a senior member of both the ACM and IEEE.