Hindawi
Security and Communication Networks
Volume 2021, Article ID 5566423, 12 pages
https://doi.org/10.1155/2021/5566423

*Research Article*

# An Automatic Source Code Vulnerability Detection Approach Based on KELM

**Gaigai Tang,[1,2] Lin Yang,[2] Shuangyin Ren,[2] Lianxiao Meng,[1,2] Feng Yang,[2] and Huiqiang Wang[1]**

[1]*School of Computer Science and Technology, Harbin Engineering University, Harbin, China*
[2]*National Key Laboratory of Science and Technology on Information System Security, Institute of System Engineering, Chinese Academy of Military Science, Beijing, China*

Correspondence should be addressed to Huiqiang Wang; wanghuiqiang@hrbeu.edu.cn

Received 12 January 2021; Revised 16 March 2021; Accepted 19 May 2021; Published 16 June 2021

Academic Editor: Xiaokang Zhou



Traditional vulnerability detection mostly ran on rules or source code similarity with manually defined vulnerability features. In fact, these vulnerability rules or features are difficult to be defined accurately, which usually cost much expert labor and perform weakly in practical applications. To mitigate this issue, researchers introduced neural networks to automatically extract features to improve the intelligence of vulnerability detection. Bidirectional Long Short-term Memory (Bi-LSTM) network has proved a success for software vulnerability detection. However, due to complex context information processing and iterative training mechanism, training cost is heavy for Bi-LSTM. To effectively improve the training efficiency, we proposed to use Extreme Learning Machine (ELM). The training process of ELM is noniterative, so the network training can converge quickly. As ELM usually shows weak precision performance because of its simple network structure, we introduce the kernel method. In the preprocessing of this framework, we introduce doc2vec for vector representation and multilevel symbolization for program symbolization. Experimental results show that doc2vec vector representation brings faster training and better generalizing performance than word2vec. ELM converges much quickly than Bi-LSTM, and the kernel method can effectively improve the precision of ELM while ensuring training efficiency.

## 1. Introduction

As software becomes more and more complicated, software vulnerabilities caused by design flaws and implementation errors become an inevitable problem in engineering [1]. According to statistics released by the Common Vulnerabilities and Exposures (CVE) [2] and National Vulnerability Database (NVD) [3], the number of software vulnerabilities has increased from 1600 to nearly 100000 since 1999 [4]. Software systems containing these vulnerabilities will face serious security risks.

On the one hand, existing vulnerability detection techniques are mostly driven by rules [5–10] and code similarity metrics [11, 12]. Vulnerability detection rules are usually defined by experienced experts. The performance of these methods is limited by the experience of experts. Generally, the features of software vulnerabilities are very difficult to be described accurately, which leads to the corresponding detection rules which are also difficult to be defined accurately and completely.

These problems inspired researchers to propose automatic vulnerability detection (source code level). Neural networks show great potential [13–17]. Neural networks can automatically extract complex features from input data, avoiding the problems of high cost, instability, and incompleteness of manually constructing features and empirically defining rules. VulDeePecker [16] utilized Bi-LSTM [18] for software vulnerability detection. Zhen Li et al. [17] discussed the performance of different neural networks on vulnerability detection separately, namely, MLP, CNN, LSTM, and Bi-LSTM. All of the above neural networks train the detection model with an iterative training mechanism, which usually costs a lot of time. To solve this problem, we introduce ELM [19], which trains the detection model with a

noniterative training mechanism. In order to improve precision performance, we then introduce the kernel method.

On the other hand, there are two most classical data preprocessing methods in neural network-based automatic vulnerability detection, namely, vector representation and program symbolization. The most common vector representation method is word2vec [20], which can vectorize the software codes into form of vector (variable length) as the input of neural network. However, word2vec usually requires additional work to further preprocess the output vector (e.g., padding zeros). The final vectors usually with large dimension can heavily affect the training efficiency of detection model. Moreover, word2vec may also lose important semantic information of the source codes, which can affect the precision of detection model. As for program symbolization, the normal way is to symbolize the variables and user-defined functions in the source code at the same time [16,17], which can be seen as a single symbolization level of 2. This idea ignores to consider the influence of multiple symbolization levels on performance of vulnerability detection model.

To alleviate the above problems, we propose a multilevel symbolization method for symbolic representation and introduce doc2vec [21] for vector representation. In detail, we first obtain symbolic representations of the source codes related to vulnerabilities through three symbolizations. Using three levels of symbolization can significantly reduce the noise introduced by irrelevant information of vulnerable codes. Then, we use doc2vec to automatically transform symbolic representation of source codes to corresponding vector representation. Compared to word2vec used in [16], we found that doc2vec is more suitable for modeling vector representation because it can not only transform source codes with arbitrary length into a fixed-length feature representation but also grasp the semantic information of source codes better. These advantages are helpful to improve the precision and training efficiency of vulnerability detection model.

The rest of this paper is organized as follows. Section 2 discusses the work related to automatic detection of software vulnerability. Section 3 describes the details of the proposed automatic software vulnerability detection method. Section 4 gives the details of experimental environment and parameter configuration, experimental results, and corresponding analysis. The conclusions and future works are presented in Section 5.

## 2. Related Work

*2.1. Vulnerability Detection Techniques.* Existing classical vulnerability detection techniques range from making use of manually defined features [5–10] to code similarity metrics [11, 12]. However, there are several primary flaws among them. First, the effort for defining vulnerability features is error-prone and manual labor consuming. Second, the features can hardly be integral and usually contain only partial information about the vulnerabilities, which may lead to high false-positive and false-negative rates [16]. Moreover, the application of the code similarity method is limited to vulnerabilities caused by code clones.

Vulnerability detection with traditional machine learning techniques such as Decision Tree [22] and Support Vector Machine (SVM) [23] mainly extracts vulnerability features from preclassified vulnerabilities. However, vulnerability detection patterns based on this type of feature are usually available for specific vulnerabilities. In the paper by Boris Chernis [24], both simple text features (e.g., character count, character diversity, and maximum nesting depth) and complex text features (e.g., character n-grams, word n-grams, and suffix trees) are extracted from the source codes and analyzed by using the naive Bayes classifier. Experimental results show that simple features performed unexpectedly better by comparing with the complex features.

Neural networks can learn complex vulnerability features automatically. Zhen Li [16] presented a vulnerability detection system VulDeePecker based on deep learning, which initiates the study of using deep learning for vulnerability detection. VulDeePecker collects the samples by first extracting code gadgets from the buggy programs and then transforming them into the vector representations using word2vec. The detection model is designed based on Bi-LSTM. Siqi Ma [13] proposed a tool called VuRLE for automatic detection and repair of vulnerabilities. VuRLE uses the context patterns to detect vulnerabilities and customizes the corresponding edit patterns to repair them. Jacob A. Harer [14] implemented various machine learning models for detecting bugs that can lead to security vulnerabilities in C/C++ code. Specifically, they used features derived from the build process and the source code. Rebecca L. Russell [15] developed a vulnerability detection tool based on deep feature representation learning that can directly interpret the parsed source codes. The source codes are firstly transformed into tokens and then embedded as vectors for both CNNs and Recurrent Neural Networks (RNNs). Zhen Li [25] proposed a systematic framework by using deep learning to detect vulnerabilities that combined syntax-based, semantics-based, and vector representations (SySeVR). SySeVR can accommodate syntax and semantic information pertinent to vulnerabilities. The source codes are successively represented by syntax-based, semantics-based, and vector representations. Zhen Li [17] performed a quantitative evaluation of the impacts of different factors (e.g., data dependency and control dependency) on the effectiveness of neural network-based vulnerability detection techniques. Zhen Li [26] presented VulDeeLocator, a deep learning-based fine-grained vulnerability detector. It leverages intermediate code to capture semantic information that cannot be conveyed by source code-based representations and presents a new idea of granularity refinement. Xin Li [27] proposed an automated and intelligent vulnerability detection method in source code based on the minimum intermediate representation learning. The sample in the form of source code is first transformed into a minimum intermediate representation; then, it is transformed into a real value vector through pretraining on an extended corpus. The vector is fed to

three concatenated convolutional neural networks to obtain high-level features of vulnerability.

*2.2. Preprocessing Method.* The commonly used preprocessing methods for automatic source code vulnerability detection are program symbolization and vector representation. Zhen Li [16, 25] first maps variable names to symbolic names (e.g., "$V1$" and "$V2$") in a one-to-one fashion, then maps function names to symbolic names (e.g., "$F1$" and "$F2$") in a one-to-one fashion, and finally uses word2vec to perform vector representation. Gustavo Grieco [28] uses word2vec to preprocess the dynamic features of source codes since it was successfully used in a variety of text mining applications. Savchenko [29] proposed a system for vulnerability detection based on deep learning approach, which performs the following steps: source code preprocessing, AST creation, code gadget extraction, and code gadget vectorization using word2vec.

*2.3. Kernel Method.* The kernel method is often used to solve the linear indivisibility problems. Qin-Qin Tao [30] proposed a locality-sensitive support vector machine using kernel combination (LS-KC-SVM) algorithm, which solved the large appearance variations due to some real-world factors on face detection. Liang [31] proposed an SVM-based method combining with the deep quasilinear kernel (DQLK) learning for large-scale image classification. It could train SVM on a large-scale dataset with less memory space and less training time. Zhang [32] developed a least-squares (LS) SVM-based identification scheme, where the system parameters were estimated in a reproducing kernel Hilbert space. It can effectively solve the issue that LS results in low accuracy in ill-conditioned scenarios. Lu Li [33] proposed the AdaBoost-WCKELM made of ELM, AdaBoost, and composite kernel method, which derived a good improvement in HSI classification accuracy.

## 3. The Methodology

Figure 1 is an overview of the proposed automatic source code vulnerability detection system using enhanced ELM on the source code level. Starting with the dataset in form of code gadget, it then obtains symbolic representation of each code gadget using multilevel symbolization. Next, it transforms the symbolic representations into vector representations with a low-dimension using doc2vec. Finally, it applies enhanced ELM neural networks to train the detection model. As for testing, code gadget is firstly preprocessed successively through multilevel symbolization and doc2vec, and then the vector representations of them are input to detection model to get the detection results. In the subsequent sections, we give the details of the main components of this system.

*3.1. Symbolic Representation.* A code gadget is composed of several program statements (i.e., lines of code), which are semantically related to each other in terms of data dependency or control dependency [16]. It can be further transformed into a form of symbolic representation using symbolization. The symbolic representation is then collected as a corpus for training the vector representation tool, such as doc2vec.

The benefit of symbolic representation is that it can result in higher training effectiveness by further reducing the length of code gadget. In symbolization, vulnerability features of each code gadget such as local variables, user-defined functions, and data types are transformed into short and fixed-length symbolic presentations, where the same features are mapped to the same symbolic presentation. In this work, we deploy three symbolization types that are shown as follows:

(i) Function calls symbolization ($F$): User-defined function names are symbolically represented as F$N$. This symbolization type is assigned the priority because vulnerability is mostly caused by improper utilization of library/API function calls. Symbolization on user-defined functions can improve the Signal-Noise Ratio (SNR) of library/API function in vulnerability information.

(ii) Variable symbolization ($V$): Variable names including parameters and local variables are symbolically represented as V$N$. In practice, the variables account for a large proportion of the codes.

(iii) Data type symbolization ($T$): Data types of variable and user-defined function are symbolically represented as T$N$. It has the least priority since many data types are not related to vulnerability information.

The symbol $N$ mentioned above in symbolization is a number which represents the index of the first occurrence of the feature while noting that multiple functions may be mapped to the same symbolic name when they appear in different code gadgets. Moreover, all the symbolization types will reserve keywords of C/C++ language.

We build a multilevel symbolization mechanism according to the priority of symbolization shown in Table 1. Level 2 includes two symbolization groups, namely, $F + V$ and $F + T$. This is because symbolizations V and T may have different effects on SNR of vulnerability information in different datasets.

We take Sample 0 as an example to show how the symbolization works, where the symbolization group $F + V$ is chosen from level 2. From Figure 2, we can observe that there are 2 user-defined functions, 5 variables, and 2 data types in Sample 0.

(i) In level 1, the two user-defined functions are symbolically represented as $F1$ and $F2$.

(ii) In level 2, the five variable names are symbolically represented as $Vi, i \in [1, 5]$.

(iii) In level 3, the two data types are symbolically represented as $T1$ and $T2$.

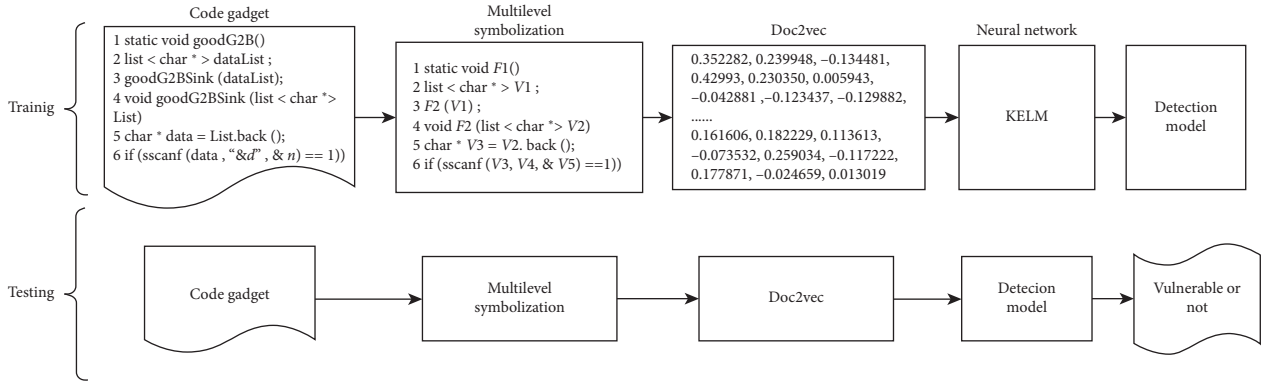


Figure 1: Overview of the proposed automatic source code vulnerability detection system using KELM.

Table 1: Multilevel symbolization.

| Symbolization level | Symbolization group |
|---|---|
| Level 1 | $F$ |
| Level 2 | $F+V$; $F+T$ |
| Level 3 | $F+V+T$ |

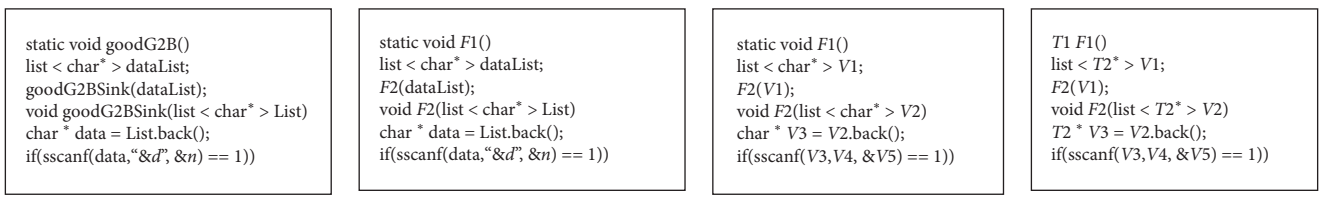


Figure 2: An example of multilevel symbolization of source code. (a) Sample 0. (b) Level 1 $F$. (c) Level 2 $F+V$. (d) Level 3 $F+V+T$.

As a result, through three levels of symbolization, Sample 0 is gradually simplified to a generalized symbolic representation, which can effectively characterize different manifestations of the same vulnerability.

*3.2. Vector Representation.* Since the neural network can only accept vector as input, the symbolic representation of source code needs to be further converted to the vector representation. Currently, the most popular vectorization methods are word2vec [34] and doc2vec [35].

Compared with the one-hot representation, a high-dimensional and sparse representation method, word2vec, outputs a low-dimensional and dense vector representation, which is conducive to improving training efficiency and precision of the model, making it widely used for vulnerability detection recently [14, 16, 17]. However, there is a drawback of word2vec; that is, it ignores the influence of word order that relates to information of a sentence or a document.

doc2vec was proposed in [35], and the authors proposed the unsupervised algorithm called Paragraph Vector that can learn fixed-length feature representation from texts with arbitrary length, ranging from a sentence to a document. Moreover, the Paragraph Vector can memorize the topic of the paragraph, which makes it be able to better extract global features than word2vec.

Given the fact that word2vec converts word to vector representation in a one-to-one fashion, thus, the length of the converted vector varies with the length of the input text. To satisfy the neural network requirement of input with a fixed length, the vector generated by word2vec needs to be further processed to obtain the corresponding fixed-length form. Different from word2vec, doc2vec can directly output fixed-length vectors from input texts with arbitrary length. Furthermore, doc2vec can also grasp more semantic information from the context of input text than word2vec. In summary, doc2vec shows great potential in source code vector representation.

*3.3. Neural Network Model.* ELM is a special type of feed-forward neural network with the noniterative training mechanism, which was proposed by Huang et al. in the 1990s [19]. Unlike traditional neural networks, which use gradient descent techniques to iteratively fine-tune all the parameters of the model, ELM randomly assigns values to some parameters according to certain rules and keeps these parameters frozen throughout the training process, while other parameters are calculated by the least square method. In other words, the training mechanism of ELM is noniterative, which can bring it much faster training speed than conventional neural networks on some tasks with relatively large data scale. Here, we take ELM with a single hidden layer network

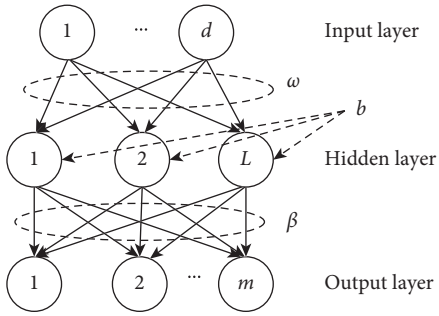


Figure 3: A typical ELM with a single hidden layer network structure.

structure as an example to introduce its training mechanism. The network structure of ELM is shown in Figure 3.

*3.3.1. ELM.* In Figure 3, $d$, $L$, and $m$ refer to the number of the input layer neurons, the hidden layer neurons, and the output layer neurons, respectively. $\omega$ is the input weights connecting the input layer to the hidden layer, $\mathbf{b}$ is the thresholds of the hidden layer neurons, and $\beta$ is the output weights connecting the hidden layer to the output layer. $\omega$ and $\mathbf{b}$ are generated randomly from the range (−1, 1) and (0, 1) under a uniform distribution. They are kept frozen throughout the training process of the model.

Given a training data set $D = \{(\mathbf{x}_i, \mathbf{t}_i) | \mathbf{x}_i \in R^d, \mathbf{t}_i \in R^m\}, i = 1, 2, \ldots, N$, the ELM model can be represented as

$$\begin{aligned} &\mathbf{H}\boldsymbol{\beta} = \mathbf{T}, \\ &\mathbf{H} = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} h(\mathbf{x}_1) & \ldots & h_L(\mathbf{x}_1) \\ \vdots & \vdots & \vdots \\ h_1(\mathbf{x}_N) & \ldots & h_L(\mathbf{x}_N) \end{bmatrix}, \\ &\mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_N^T \end{bmatrix} = \begin{bmatrix} t_{11} & \ldots & t_{1m} \\ \vdots & \vdots & \vdots \\ t_{N1} & \ldots & t_{Nm} \end{bmatrix}, \end{aligned} \tag{1}$$

where $\mathbf{T}$ is the expected output matrix and $\mathbf{H}$ is the hidden layer output matrix. $\mathbf{h}(\mathbf{x}_i) = \sum_{j=1}^{L} g(\omega_j \cdot \mathbf{x}_i + b_j), i = 1, 2, \ldots, N$, which is the output vector of the hidden layer with respect to the input $\mathbf{x}_i$. $g(\cdot)$ is the activation function of the ELM. And $\omega_j \cdot \mathbf{x}_i$ denotes the inner product of the input weights and the features of the $i$th training sample. The output weights $\beta$ can be obtained by

$$\boldsymbol{\beta} = \mathbf{H}^{+}\mathbf{T} = \begin{cases} \mathbf{H}^T\left(\lambda\mathbf{I} + \mathbf{H}\mathbf{H}^T\right)^{-1}\mathbf{T}, & \text{when } N \le L, \\ \left(\lambda\mathbf{I} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{T}, & \text{when } N \ge L, \end{cases} \tag{2}$$

where $\mathbf{H}^{+}$ refers to the Moore − Penrose generalized inverse of $\mathbf{H}$, $L$ refers to neuron number of hidden layers, $\mathbf{I}$ refers to an $N$ identity matrix, and $\lambda$ refers to a regularization factor with a value between [0,1].

The ELM output function is

$$f(\mathbf{x}) = \begin{cases} \mathbf{h}(\mathbf{x})\mathbf{H}^T\left(\lambda\mathbf{I} + \mathbf{H}\mathbf{H}^T\right)^{-1}\mathbf{T}, & \text{when } N \le L, \\ \left(\lambda\mathbf{I} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{h}(\mathbf{x})\mathbf{T}, & \text{when } N \ge L. \end{cases} \tag{3}$$

The optimization objective of the ELM model can be expressed as

$$\min\left(\sum_{i=1}^{N} \left\| f(\mathbf{x}_i) - \mathbf{t}_i \right\|^2\right), \tag{4}$$

where $f(\mathbf{x}_i)$ and $\mathbf{t}_i$ refer to the predictive label and the real label of the $i$th sample, respectively.

*3.3.2. KELM.* Kernel method is an effective way to solve the nonlinear problems by mapping the data to high-dimensional space so that the nonlinear problem can be transformed into a linear problem. With the combination of kernel method, there are two benefits compared with conventional ELM. For one thing, it solves the problem that the number of hidden layer nodes in conventional ELM depends on manual setting, which shows better stability [36]. For another thing, the kernel function maps the data to the high-dimensional space, and the distribution of the data in the transformed space is very smooth. In fact, the smooth new data make the classification problem easier, so the model can show better effectiveness. Radial Basis Function (RBF) is the preferred kernel function in our experiments because it has only one hyperparameter which simplifies the model configuration and training cost. RBF kernel function can be expressed as

$$K(\mathbf{x}, \mathbf{y}) = e^{-\gamma\|\mathbf{x}-\mathbf{y}\|^2}, \tag{5}$$

where $\mathbf{x}$ and $\mathbf{y}$ represent the samples, $\gamma$ represents the unique hyperparameter of Gaussian kernel function, and $\|\mathbf{x} - \mathbf{y}\|$ denotes the norm of vectors.

The kernel matrix for ELM can be defined as [37]

$$\begin{aligned} \boldsymbol{\Omega}_{\text{ELM}} &= \mathbf{H}^T\mathbf{H}, \\ \boldsymbol{\Omega}_{\text{ELM}_{ij}} &= \mathbf{h}(\mathbf{x}_i) \cdot \mathbf{h}(\mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j). \end{aligned} \tag{6}$$

And we can revise equation (2) when $N \ge L$ as

$$\boldsymbol{\beta} = (\lambda\mathbf{I} + \boldsymbol{\Omega})^{-1}\mathbf{H}^T\mathbf{T}, \tag{7}$$

and then, the ELM output function (3) can be as follows:

$$\begin{aligned} f(\mathbf{x}) &= \left(\lambda\mathbf{I} + \mathbf{H}^T\mathbf{H}\right)^{-1}\mathbf{H}^T\mathbf{h}(\mathbf{x})\mathbf{T} \\ &= (\lambda\mathbf{I} + \boldsymbol{\Omega})^{-1}\begin{bmatrix} K(\mathbf{x}, \mathbf{x}_1) \\ \vdots \\ K(\mathbf{x}, \mathbf{x}_N) \end{bmatrix}^T\mathbf{T}. \end{aligned} \tag{8}$$

From equation (8), we can find that ELM combined with the kernel method can avoid the problem that the number of hidden layer nodes in conventional ELM depends on manual setting.

## 4. Experiment and Evaluation

The goal of our work is to construct an automatic software vulnerability detection model with both superior precision and efficiency. To be specific, we investigate the following questions in experiments:

(i) Question1 ($Q1$): How differently do neural network models perform on vulnerability detection?

(ii) Question2 ($Q2$): How differently do vector representation methods affect the performances of neural networks? Specifically, does doc2vec outperform word2vec on vulnerability detection?

(iii) Question3 ($Q3$): What are the effects of different symbolization on the performances of neural networks?

### 4.1. Experiment Setting and Implementation

*4.1.1. Dataset.* In our experiments, we include the following three datasets from [16]. Each sample is a piece of source code with known vulnerabilities. Table 2 shows the number of samples (i.e., source code files) in each dataset.

(i) BE-ALL includes samples with buffer error vulnerabilities (CWE-119) and ALL library/API function calls.

(ii) RM-ALL includes samples with resource management error vulnerabilities (CWE-399) and ALL library/API function calls.

(iii) HY-ALL includes samples with hybrid buffer error vulnerabilities (CWE-119), resource management error vulnerabilities (CWE-399), and ALL library/API function calls.

Each dataset is partitioned into two parts with a proportion of 80% and 20%, where the larger part is for training and the other part is for testing. Each sample in the dataset is in the form of code gadget with a ground truth label.

*4.1.2. Evaluation Metrics.* In our experiment, we used the indexes mentioned in [38] to evaluate the effectiveness of vulnerability detection model, that is, False Positive Rate (FPR), True Positive Rate (TPR), Precision (P), and F1-measure (F1). The value range of these four indicators is [0, 1]. For FPR, the closer their values are to 0, the better the performance of the model is; for other indicators, the closer their values are to 1, the better the performance of the model is.

The quality of vector representation can be evaluated by Cosine Similarity (cosine) between vectors in the vector space, which can be calculated by the following formula. The range of cosine value is [−1, 1]. The closer the value is to 1 or −1, the more similar the two vectors are.

$$\text{cosine}(A, B) = \frac{A \cdot B}{\|A\|_2 \|B\|_2}, \tag{9}$$

Table 2: Number of samples in each dataset.

| Dataset | Code gadgets | Vulnerable code gadgets |
|---|---|---|
| BE-ALL | 39753 | 10440 |
| RM-ALL | 21885 | 7285 |
| HY-ALL | 61638 | 17725 |

where A and B refer to vectors. Given the fact that Cosine Similarity only considers the angle between vectors, so that it can avoid too large output deviation due to different dimension of input vectors. This is the main reason why we choose Cosine Similarity as the evaluation metric of vector representation.

*4.1.3. Parameters Setting for Neural Networks.* In our experiments, we used two types of neural networks for the vulnerability detection model, namely, Bi-LSTM and ELM. For both, there is only one hidden layer in the network structure. We build the following five configurations. We do not list the configuration of AdaBoost KELM because it is predictable that the calculation of KELM with weight and iteration mechanism is very complex and the efficiency will be greatly reduced.

(i) word2vec with Bi-LSTM ($w + B$), which was used by VulDeePecker

(ii) doc2vec with Bi-LSTM ($d + B$)

(iii) doc2vec with ELM ($d + E$)

(iv) doc2vec with AdaBoost ELM ($d +$ Ada-E)

(v) doc2vec with KELM ($d +$ KE)

We have implemented the CPU versions of Bi-LSTM and ELM, and all the models were trained in the PC environment with CPU. For Bi-LSTM, the batch size, the dropout rate, the number of epochs, and the number of the hidden layer neurons were set to 64, 0.5, 2, and 60, respectively, and the optimizer chosen was Root Mean Square Prop (RMSProp). For ELM, the number of the hidden layer neurons was set to 5000 and the activation function used sigmoidal function. The input weights and the hidden biases of ELM were generated randomly from (−1, 1) and (0, 1), respectively, under a uniform distribution. The details of the parameters' configuration of ELM are given as follows.

To determine which activation function is the best choice for the ELM-based detection model, we implement an experiment to discuss the effectiveness of ELM with five activation functions, respectively. The number of neurons is set to 250 and the dataset is HY-ALL. From the results in Figure 4, we can find that ELM with sigmoidal function outperforms the other activation functions on precision and F1.

In terms of neuron configuration of ELM, we have done several experiments to analyze the effect of a different number of neurons on the precision of ELM as shown in Table 3. Generally speaking, when the number of neurons ranges from 250 to 12000, the precision of ELM gradually increases with the number of neurons increasing, but when the number of neurons is more than or equal to 15000, the

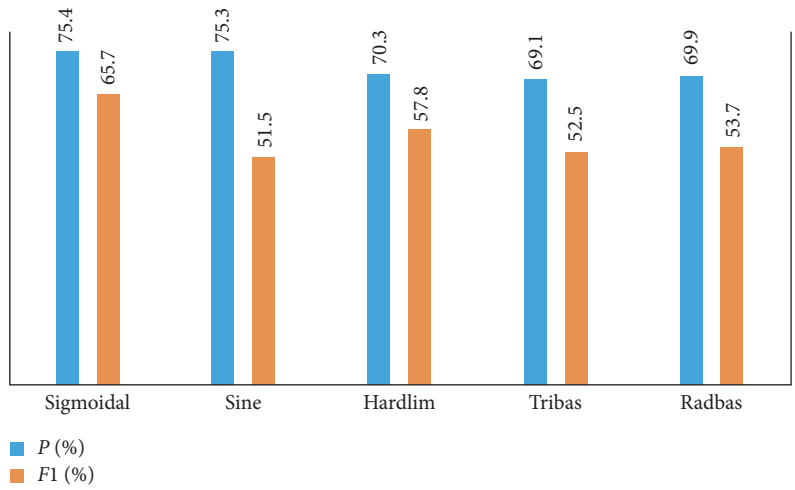


Figure 4: Effect of different activation function on the precision of ELM.

Table 3: Effect of different number of neurons on the precision of ELM.

| Neuron number | FPR (%) | TPR (%) | $P$ (%) | $F1$ (%) | Training time (s) |
|---|---|---|---|---|---|
| 250 | 7.5 | 58.2 | 75.4 | 65.7 | 0.92 |
| 500 | 6.6 | 66.1 | 80.0 | 72.4 | 2.52 |
| 1000 | 6.4 | 72.8 | 81.9 | 77.1 | 7.75 |
| 3000 | 5.4 | 80.5 | 85.5 | 82.9 | 49.11 |
| 5000 | 4.4 | 83.6 | **88.3** | **85.9** | **128.72** |
| 10000 | 4.4 | 85.9 | 88.6 | 87.2 | 496.02 |
| 12000 | 4.3 | 85.9 | 88.9 | 87.4 | 640.68 |
| 15000 | 4.4 | 86.6 | 88.6 | 87.6 | 1143.70 |
| 20000 | 4.6 | 86.7 | 88.2 | 87.4 | 2201.60 |

precision of ELM begins to decline slowly. In particular, when the number is between 250 and 5000, the precision improvement is more obvious, while the number increases from 5000 to 12000, the precision improvement is slight, nearly 0.3%, and the training time increased by 4 times. Considering the cost-effectiveness of precision improvement and time consumption, we set the number of neurons as 5000.

Kernel function plays a very important role in KELM, which largely determines its precision performance. We collect three commonly used kernel functions to make a comparison experiment. The comparison of the results after fine-tuning is shown in Figure 5. It is clear from the result that RBF shows the best overall performance than the other two kernel functions. Thus, the subsequent KELM-related experiments set the RBF as the kernel function.

### 4.2. Results and Evaluation

*4.2.1. Results for Q1.* Regarding the impacts of different neural network models on the performances of vulnerability detection, we evaluate the precision and efficiency of the above five configurations on all datasets. Table 4 shows the effect of different neural network models on vulnerability detection precision, while Table 5 gives the efficiency of different neural network models on vulnerability detection. In the experiments, all three datasets are preprocessed with the symbolization group $F + V$.

According to the results in Table 4, we analyze them from two aspects: precision comparison of conventional Bi-LSTM and ELM and enhanced effect of conventional ELM using kernel function and AdaBoost method.

Compared with ELM, Bi-LSTM is slightly inferior in RM-ALL, a small-scale dataset, but superior in BE-ALL and HY-ALL, the large-scale datasets. This may be due to the fact that the deep learning model is more suitable for large dataset scenarios. Besides, Bi-LSTM shows lower FPR than ELM on all three datasets, which can be explained by the fact that Bi-LSTM can express the long-term dependency information in the input, while ELM is based on forwarding neural network; it is slightly inferior to Bi-LSTM in the context processing.

For Ada-E, it outperforms conventional ELM on RM-ALL and HY-ALL, which shows the advantage of the ensemble learning, for example, combination enhancement. However, it shows similar P and lower TPR than ELM on RM-ALL, which may be due to the overfitting effect of ensemble learning for high-precision base classifiers. It can be seen that if the base classifier is with very high precision, the final classifier generated by AdaBoost does not always show the higher precision but may be worse if the basic classifier shows high enough precision. For KE, it shows the lowest FPR and the highest P on the three datasets compared with the other five configurations, which benefits from its effective way to solve nonlinear problems through high-dimensional mapping. Besides, it also results in the lowest TPR, but this is acceptable; it is due to the fact that the high false-positive rate is the primary problem of vulnerability detection tools in practical application.

From Table 5, we can find that the configuration $w + B$ performs the longest time for training and detection on HY-ALL, while configuration $d + B$ costs less than 1/30 of configuration $w + B$. It is because the configuration $w + B$ in [16] outputs vectors with a longer dimension of 2500, which

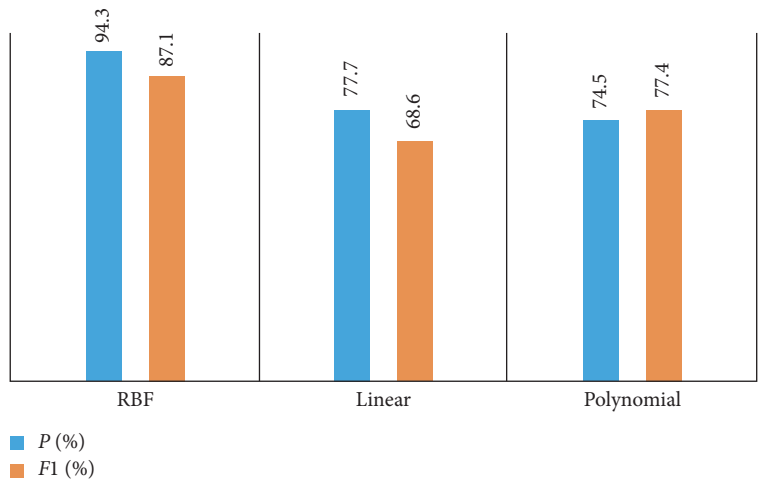


Figure 5: Effect of different kernel function on the precision of KELM.

Table 4: Effect of different neural network models on vulnerability detection precision.

| Dataset | Pattern | FPR (%) | TPR (%) | $P$ (%) | $F1$ (%) |
|---|---|---|---|---|---|
| BE-ALL | $w+B$ | 2.9 | 82.0 | 91.7 | **86.6** |
| | $d+B$ | 3.9 | 83.1 | 88.1 | 85.5 |
| | $d+E$ | 4.4 | 81.9 | 86.8 | 84.3 |
| | $d$ + Ada-E | 3.9 | 82.7 | 88.1 | 85.3 |
| | $d$ + KE | 1.8 | 78.7 | **93.8** | 85.6 |
| RM-ALL | $w+B$ | 2.8 | 95.3 | 94.6 | **95.0** |
| | $d+B$ | 3.8 | 90.3 | 91.9 | 91.1 |
| | $d+E$ | 3.9 | 92.4 | 94.9 | 92.1 |
| | $d$ + Ada-E | 2.8 | 83.8 | 93.4 | 88.3 |
| | $d$ + KE | 1.1 | 82.7 | **97.4** | 89.5 |
| HY-ALL | $w+B$ | 5.1 | 83.9 | 86.9 | 85.4 |
| | $d+B$ | 3.3 | 83.8 | 91.1 | **87.2** |
| | $d+E$ | 4.4 | 83.6 | 88.3 | 85.9 |
| | $d$ + Ada-E | 3.8 | 84.3 | 89.8 | 87.0 |
| | $d$ + KE | 1.9 | 81.0 | **94.3** | 87.1 |

Table 5: Efficiency of different neural network models on vulnerability detection.

| Pattern | Training code gadgets | Detection code gadgets | Training time (s) | Detection time (s) |
|---|---|---|---|---|
| $w+B$ | 48744 | 12894 | 36372.2 | 156.2 |
| $d+B$ | 49310 | 12328 | 1543.5 | 3.0 |
| $d+E$ | 49310 | 12328 | **128.7** | **2.7** |
| $d$ + Ada-E | 49310 | 12328 | 2914.0 | 7.8 |
| $d$ + KE | 49310 | 12328 | 335.4 | 11.0 |

results in a higher computation complexity for Bi-LSTM. Moreover, compared with configuration $d+B$, configuration $d+E$ further reduces the time cost of the training and detection to a few minutes. This can be explained by the fact that the noniterative training mechanism of ELM reduces the computation of parameters. Ada-E improves the precision of ELM by adding the iteration mechanism and introducing the weight mechanism to ELM, but these operations increase the computational complexity. Therefore, the training and detection time of ELM will be multiplied accordingly. KE shows a lower efficiency than conventional ELM because it maps the input and output to a higher dimension for calculation which will result in a larger computational complexity than the former.

Thus, we can conclude that configuration with conventional Bi-LSTM achieves a higher precision, while the configuration with conventional ELM is more effective. Using AdaBoost and kernel function can effectively further improve the precision of conventional ELM in vulnerability detection. In particular, the kernel function achieves a very good precision improvement effect while maintaining higher efficiency than conventional Bi-LSTM.

*4.2.2. Results for Q2.* To answer the second question, we evaluate the effectiveness of the two vector representation methods, namely, doc2vec and word2vec. We implement experiments with four samples shown in Figure 6. Sample 2 and Sample 4 are labeled as "vulnerable," while Sample 1 and Sample 3 are not. We collect these four samples from dataset

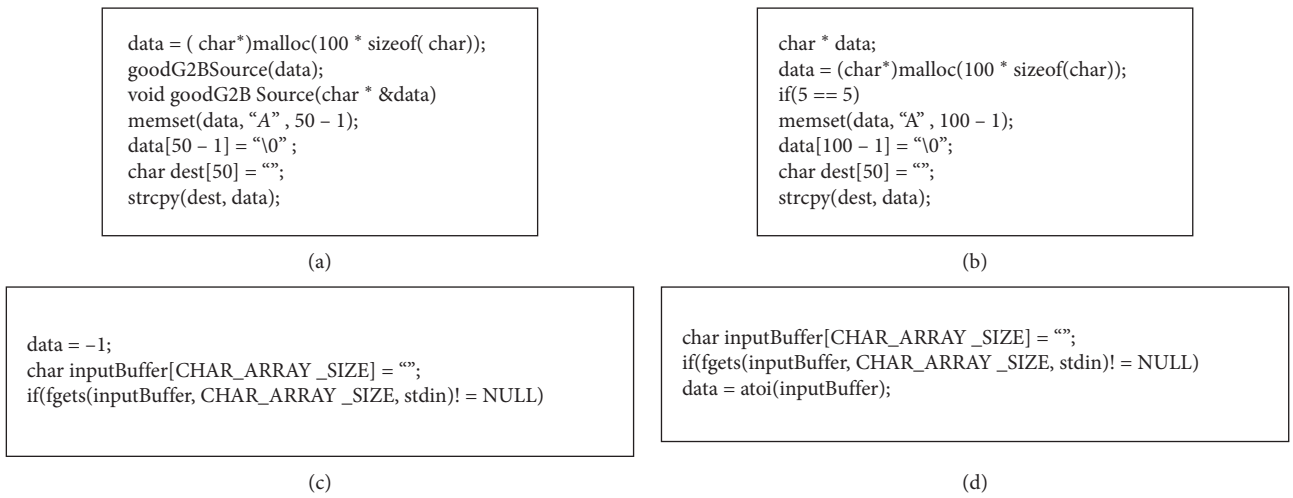


Figure 6: Four samples used to evaluate the effectiveness of the two vector representation methods. (a) Sample 1. (b) Sample 2. (c) Sample 3. (d) Sample 4.

BE-ALL and dataset HY-ALL for two experiments. And both samples are preprocessed with symbolization group of $F + V$.

We evaluate the effectiveness of vector representations by using the similarity measure cosine. The output vector dimension of word2vec is set to 2500, where the output vector dimension of one word is set to 50, and the number of words to represent a paragraph is set to 50. The output vector dimension of doc2vec is set to 250. The reason of making different output vector dimension settings of word2vec and doc2vec is due to the fact that if both dimensions are set to be the same (e.g., 250), then word2vec outputs vector dimension of one word will be 5, or the number of words to represent a paragraph will be 5, which may have a great influence on the effectiveness of vector representation. As a result, the comparison of the effectiveness of word2vec and doc2vec is carried out under the condition that they both use a proper dimension of output vector representation.

From the perspective of vulnerability detection, in terms of the fact that two similar samples are given different labels, it is better to make the similarity between the two vectors after vectorization be small as far as possible, which is conducive to the training of vulnerability detection model using neural network. From Table 6, we find that, for Sample 1 and Sample 2, word2vec outputs vectors with a higher cosine value than doc2vec, while for Sample 3 and Sample 4, it outputs a lower cosine value than doc2vec. Generally, compared with word2vec, doc2vec can output nearly similar or better vector representation with smaller dimension. It can be explained by the fact that, as noted in [16], in order to obtain a fixed length of vector representation, vectors generated by word2vec should be padded with zeros, which may cause the loss of semantic information of the samples. Moreover, it is obvious that a neural network model with low-dimension input vectors can result in good efficiency. We can also observe that, for the same sample in different dataset, doc2vec can output more similar cosine results than word2vec; the biggest output cosine deviation of doc2vec is 0.008, while word2vec results in a value of 0.032. It shows that doc2vec can perform well on large datasets. This conclusion also can be justified by the results in Table 4, where the configurations with doc2vec show better results than the ones with word2vec on HY-ALL.

*4.2.3. Results for Q3.* To answer the third question, we take the configuration $d + B$ and $d +$ KE as baselines to discuss whether symbolization can further improve the precision of the neural network model. We implement experiments with all the three datasets. And for each dataset, we apply symbolization level from 1 to 3 for preprocessing the datasets.

Table 7 summarizes results of how differently symbolization levels affect the precision of Bi-LSTM. From the perspective of different datasets, symbolization levels have a bigger impact on the precision of Bi-LSTM vulnerability detection model with smaller datasets, which shows a maximum deviation of precision at 3.1% in BE-ALL and 2.6% in RM-ALL. However, with the largest dataset HY-ALL, the maximum precision deviation is 0.9%. This may be because the scale of datasets can affect generalization performance of detection model, while the impact of symbolization is gradually reduced according to the scale becoming smaller. From the perspective of symbolization levels, configuration $d + B$ with the symbolization level 1 shows a better and more stable performance than other symbolization levels, while the symbolization level 2 results in an unstable performance, and the symbolization level 3 shows the worst performance. The main reason is that a high level of symbolization may lose some key vulnerability information in the source codes. Moreover, it should be mentioned that symbolization groups of $F + T$ outperform than symbolization groups of $F + V$ with all datasets; it may be due to the fact that there are many codes related to data type in the source codes; symbolizing them can better capture the vulnerability information.

Table 8 summarizes results of how differently symbolization levels affect the precision of KELM. From the perspective of different datasets, symbolization levels have a big

Table 6: Effectiveness of word2vec compared with doc2vec.

| Sample | Tool | Vector dimension | Cosine (BE-ALL) | Cosine (HY-ALL) |
|---|---|---|---|---|
| 1 and 2 | word2vec | 2500 | 0.832 | 0.828 |
| | doc2vec | 250 | **0.642** | **0.639** |
| 3 and 4 | word2vec | 2500 | **0.482** | **0.514** |
| | doc2vec | 250 | 0.586 | 0.578 |

Table 7: Effect of different symbolization levels on Bi-LSTM precision.

| Dataset | Symbolization group | FPR (%) | TPR (%) | $P$ (%) | $F1$ (%) |
|---|---|---|---|---|---|
| BE-ALL | F | 2.9 | 86.2 | **91.2** | **88.6** |
| | $F+V$ | 3.9 | 83.1 | 88.1 | 85.5 |
| | $F+T$ | 3.2 | 84.6 | 90.4 | 87.4 |
| | $F+V+T$ | 3.5 | 84.5 | 89.4 | 86.9 |
| RM-ALL | $F$ | 2.8 | 92.2 | 94.1 | 93.1 |
| | $F+V$ | 3.8 | 90.3 | 91.9 | 91.1 |
| | $F+T$ | 2.6 | 93.5 | **94.5** | **94.0** |
| | $F+V+T$ | 3.4 | 90.9 | 92.7 | 91.8 |
| HY-ALL | $F$ | 3.2 | 87.8 | **92.0** | **89.8** |
| | $F+V$ | 3.3 | 83.8 | 91.0 | 87.2 |
| | $F+T$ | 3.2 | 85.6 | 91.7 | 88.5 |
| | $F+V+T$ | 3.3 | 86.0 | 91.1 | 88.5 |

Table 8: Effect of different symbolization levels on KELM precision.

| Dataset | Symbolization group | FPR (%) | TPR (%) | $P$ (%) | $F1$ (%) |
|---|---|---|---|---|---|
| BE-ALL | F | 2.1 | 83.5 | 93.4 | 88.1 |
| | $F+V$ | 1.8 | 78.7 | **93.8** | 85.6 |
| | $F+T$ | 2.0 | 83.6 | 93.6 | **88.3** |
| | $F+V+T$ | 2.3 | 79.1 | 92.3 | 85.2 |
| RM-ALL | $F$ | 1.3 | 88.0 | 97.0 | **92.3** |
| | $F+V$ | 1.1 | 82.7 | **97.4** | 89.5 |
| | $F+T$ | 1.2 | 87.5 | 97.3 | 92.1 |
| | $F+V+T$ | 1.3 | 83.1 | 96.8 | 89.5 |
| HY-ALL | $F$ | 1.1 | 81.8 | 96.8 | 88.7 |
| | $F+V$ | 1.9 | 81.0 | 94.3 | 87.1 |
| | $F+T$ | 0.87 | 82.3 | **97.5** | **89.3** |
| | $F+V+T$ | 1.8 | 81.6 | 94.7 | 87.7 |

impact on the precision of the KELM-based vulnerability detection model with dataset HY-ALL, which shows a maximum deviation of precision at 3.2%. However, with smaller datasets BE-ALL and RM-ALL, the maximum deviation precision is 1.5% and 0.8%, respectively. This is because the semantic changes of samples generated by different symbolization are smaller in small datasets and larger in large datasets. Therefore, it will cause a large deviation of precision performance. From the perspective of symbolization levels, configuration $d+B$ with the symbolization group of $F+T$ shows the best and most stable performance than other symbolization levels, while the symbolization level 1 results in a better performance than symbolization level 3 with all the datasets. The former phenomenon may be due to the fact that KELM is more suitable for extracting the vulnerability information with dataset preprocessed by symbolization of $F+T$, and the latter one can be explained by the reason mentioned above.

Table 9: Efficiency of different symbolization levels on preprocessing.

| Dataset | Symbolization group | Training time (s) |
|---|---|---|
| BE-ALL | $F$ | 1721.9 |
| | $F+V$ | 1645.5 |
| | $F+T$ | 1674.1 |
| | $F+V+T$ | 1639.3 |
| RM-ALL | $F$ | 930.8 |
| | $F+V$ | 920.8 |
| | $F+T$ | 908.1 |
| | $F+V+T$ | 907.4 |
| HY-ALL | $F$ | 2762.7 |
| | $F+V$ | 2693.6 |
| | $F+T$ | 2692.2 |
| | $F+V+T$ | 2665.2 |

Furthermore, to verify the training efficiency of the proposed multilevel symbol representation, we also give the comparative analysis of time complexity as shown in Table 9. From Table 9, we can observe two phenomena as follows: one is that, for the same dataset, there is a linear downward trend of training time as the symbolization level increases from 1 to 3; the other is that the training time increases correspondingly as the size of dataset increasing. Meanwhile, compared with the symbolization level 1, symbolization level 3 improves training efficiency by about 20% on all three datasets. This can indicate that multilevel symbolization can slightly improve the efficiency of preprocessing, which is not worth mentioning when it is used to improve the precision performance of neural networks.

## 5. Conclusions

We have made the first effort to use ELM to solve the training efficiency issue of the vulnerability detection model.

Moreover, we then introduce the kernel method to improve the precision of ELM. Experimental results show that ELM with the kernel method is an effective combination of both efficiency and precision. Particularly, for the data preprocessing issue, we find that vector representation using doc2vec performs well on large datasets, and an appropriate symbolization level can effectively improve the precision of vulnerability detection. These experimental conclusions will provide researchers and engineers with guidelines when choosing neural networks and data preprocessing methods for vulnerability detection.

There are several limitations of this paper, which are expected to be researched in the future. First, from more than one kind of single-layer feedforward neural network that could be used for vulnerability detection, we only used ELM in this work. Second, not limited to the kernel method, we expect to explore other methods to improve the precision of ELM subsequently. Third, the datasets used in our experiment are provided by a single source, and more datasets from different sources can be expanded to verify the effectiveness of our proposed approach.

## Data Availability

Previously reported vulnerability data were used to support this study and are available at https://github.com/CGCL-codes/VulDeePecker. These prior studies (and datasets) are cited at relevant places within the text as references [16].

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This research was funded by the Nature Science Foundation of China (grant no. 61872104) and Fundamental Research Funds for the Central Universities in China (grant no. 3072020CF0603).

## References

[1] G. Tang, L. Meng, H. Wang et al., "A comparative study of neural network techniques for automatic software vulnerability detection," in *Proceedings of the 2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Hangzhou, China, 2020.

[2] Common vulnerabilities and exposures. https://cve.mitre.org.

[3] National vulnerability database. https://nvd.nist.gov/.

[4] T. Ji, Y. Wu, C. Wang et al., "The coming era of alphahacking?: a survey of automatic software vulnerability detection, exploitation and patching techniques," in *Proceedings of the Third IEEE International Conference on Data Science in Cyberspace*, DSC, Guangzhou, China, 2018.

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

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

[7] J. Viega and J. T. Bloch, "A static vulnerability scanner for C and C++ code," in *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000*, vol. 257, New Orleans, LA, USA, 2000.

[8] Checkmarx, https://www.checkmarx.com/.

[9] Coverity. https://scan.coverity.com/.

[10] H. P. Fortify. https://www.hpfod.com/.

[11] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: a scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2017.

[12] Z. Li, D. Zou, S. Xu et al., "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, Los Angeles, CA, USA, 2016.

[13] S. Ma, F. Thung, D. Lo et al., "Vurle: automatic vulnerability detection and repair by learning from examples," in *Proceedings of the Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security*, Oslo, Norway, 2017.

[14] J. A. Harer, L. Y. Kim, R. L. Russell et al., "Automated software vulnerability detection with machine learning," *CoRR*, 2018.

[15] R. L. Russell, L. Y. Kim, L. H. Hamilton et al., "Automated vulnerability detection in source code using deep representation learning," in *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications*, ICMLA 2018, Orlando, FL, USA, 2018.

[16] Z. LiD. Zou et al., "Vuldeepecker: a deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018*, San Diego, CA, USA, 2018.

[17] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103184–103197, 2019.

[18] K. Cho, Bart van Merrienboer, and D. Bahdanau, "On the properties of neural machine translation: encoder-decoder approaches," in *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, Doha, Qatar, 2014.

[19] G.-B. Huang, Q.-Yu Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, vol. 2, pp. 985–990, Budapest, Hungary, 2004.

[20] Word2vec. https://nvd.nist.gov/.

[21] Doc2vec. https://radimrehurek.com/gensim/models/doc2vec.html.

[22] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[24] B. Chernis, M. Rakesh, and Verma, "Machine learning methods for software vulnerability detection," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, Tempe, AZ, USA, 2018.

[25] Z. Li, D. Zou, S. Xu et al., "Sysevr: a framework for using deep learning to detect software vulnerabilities," *CoRR*, 2018.

[26] Z. Li, D. Zou, S. Xu, and Z. Chen, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *CoRR*, 2020.

[27] F. Yamaguchi, N. Golde, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014*, Berkeley, CA, USA, 2014.

[28] G. Grieco, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016*,

E. Bertino, R. S. Sandhu, and P. Alexander, Eds., ACM, New Orleans, LA, USA, 2016.

[29] A. Savchenko, O. Fokin, A. Chernousov, O. Sinelnikova, and S. Osadchyi, "Deedp: vulnerability detection and patching based on deep learning," *Theoretical and Applied Cybersecurity*, vol. 2, p. 8, 2020.

[30] Q.-Q. Tao, S. Zhan, X.-H. Li, and T. Kurihara, "Robust face detection using local cnn and svm based on kernel combination," *Neurocomputing*, vol. 211, pp. 98–105, 2016.

[31] P. Liang, W. Li, D. Liu, and J. Hu, "Large-scale image classification using fast svm with deep quasi-linear kernel," in *Proceedings of the 2017 International Joint Conference on Neural Networks*, pp. 1064–1071, IJCNN), Anchorage, AL, USA, 2017.

[32] W. Zhang, M. Xia, and J. Zhu, "An efficient hierarchical identification method with kernel-based svm for equivalent systems of aircrafts," *IEEE Access*, vol. 7, pp. 83243–83250, 2019.

[33] Li Lu, C. Wang, W. Li, and J. Chen, "Hyperspectral image classification by adaboost weighted composite kernel extreme learning machines," *Neurocomputing*, vol. 275, pp. 1725–1733, 2018.

[34] L. Wolf, H. Yair, and N. Dershowitz, "Joint word2vec networks for bilingual semantic representations," *International Journal of. Computational Linguistics and Applications*, vol. 5, no. 1, pp. 27–42, 2014.

[35] V. Quoc, "Le and Tomas Mikolov. Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, Beijing, China, 2014.

[36] H. Yuan, B. Wang, and L. Niu, "Kernel extreme learning machine for learning from label proportions," *Lecture Notes in Computer Science*, vol. 400, p. 409, 2018.

[37] G. B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 2, pp. 513–529, 2012.

[38] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Computational Survey*, vol. 49, no. 4, p. 62, 2017.