Hindawi
Security and Communication Networks
Volume 2022, Article ID 5203217, 12 pages
https://doi.org/10.1155/2022/5203217

Research Article

Deep Neural Embedding for Software Vulnerability Discovery: Comparison and Optimization

Xue Yuan,[1] Guanjun Lin,[2] Yonghang Tai,[1] and Jun Zhang[1]

[1]*School of Physics and Electronic Information, Yunnan Normal University, Kunming 650000, China*
[2]*School of Information Engineering, Sanming University, Sanming, Fujian 365004, China*

Correspondence should be addressed to Yonghang Tai; taiyonghang@ynnu.edu.cn and Jun Zhang; junzhang@ynnu.edu.cn

Received 1 September 2021; Revised 31 October 2021; Accepted 6 November 2021; Published 18 January 2022

Academic Editor: Weizhi Meng

Due to multitudinous vulnerabilities in sophisticated software programs, the detection performance of existing approaches requires further improvement. Multiple vulnerability detection approaches have been proposed to aid code inspection. Among them, there is a line of approaches that apply deep learning (DL) techniques and achieve promising results. This paper attempts to utilize CodeBERT which is a deep contextualized model as an embedding solution to facilitate the detection of vulnerabilities in C open-source projects. The application of CodeBERT for code analysis allows the rich and latent patterns within software code to be revealed, having the potential to facilitate various downstream tasks such as the detection of software vulnerability. CodeBERT inherits the architecture of BERT, providing a stacked encoder of transformer in a bidirectional structure. This facilitates the learning of vulnerable code patterns which requires long-range dependency analysis. Additionally, the multihead attention mechanism of transformer enables multiple key variables of a data flow to be focused, which is crucial for analyzing and tracing potentially vulnerable data flaws, eventually, resulting in optimized detection performance. To evaluate the effectiveness of the proposed CodeBERT-based embedding solution, four mainstream-embedding methods are compared for generating software code embeddings, including Word2Vec, GloVe, and FastText. Experimental results show that CodeBERT-based embedding outperforms other embedding models on the downstream vulnerability detection tasks. To further boost performance, we proposed to include synthetic vulnerable functions and perform synthetic and real-world data fine tuning to facilitate the model learning of C-related vulnerable code patterns. Meanwhile, we explored the suitable configuration of CodeBERT. The evaluation results show that the model with new parameters outperform some state-of-the-art detection methods in our dataset.

1. Introduction

Software vulnerability has long been a severe but crucial research issue in cybersecurity [1–3]. These security vulnerabilities threaten the IT infrastructure of organizations and government sectors. There are increasingly more vulnerabilities being discovered. Multiple vulnerabilities released in the Common Vulnerabilities and Exposures were approximately 4,600 in 2010. However, it grows to approximately 153,955 in 2021. Software vulnerability [4–6], as a threat, is increasing in frequency, scale, and severity, which are similar to natural disasters; it may lead to unintended and severe consequences. Once vulnerability in a key system is exploited by attackers, millions of computer systems may be affected [7].

Despite the efforts that have been invested in pursuing the low probability of mistake when programming, software vulnerabilities remain and will continue to be a high-profile problem [8]. Recent years have witnessed a tremendous change in defending against vulnerabilities, from primarily reactive detection towards attempting to actively predict the code snippet whether it contains a vulnerability.

Deep learning-based vulnerability detection has attracted much attention recently. These techniques have been applied in the field of communications and networking and have achieved promising outcomes [9, 10]. For vulnerability detection tasks, neural networks are applied for automated feature extraction, which helps to improve the generalization ability of a model being capable of extracting high-level and latent features automatically. Therefore, researchers are

motivated to improve the usefulness of deep learning-based vulnerability detection solutions from various aspects. The process of applying deep learning techniques in the context of vulnerability detection can be divided into four steps: data collection, data preparation, model building, and evaluation/test. Some existing vulnerability detectors with traditional embedding solutions such as Word2Vec, GloVe, and FastText often incur low precision and recall [11]. The bimodal feature of code [12] demands a model to be able to handle long-distance contextual dependencies. However, conventional embedding methods adopted by existing studies, such as Word2Vec, can only generate a unique embedding for a given code token and are unable to produce embeddings based on different contexts. Hence, we consider using CodeBERT as a code embedding and feature generator.

CodeBERT is capable of producing different embeddings based on different contexts. On the one hand, CodeBERT is based on a bidirectional transformer which can capture long-distance dependencies of code sequences. It can preserve the relationship between contexts, capture latent vulnerable code patterns, and minimize the loss of information. On the other hand, CodeBERT inherits the structure of multihead attention, which makes the model focus on multiple key points of a code sequence. When there is a loop in a code fragment, the value of a variable is constantly changing according to the loop condition; however, using the noncontextual embedding methods, the vector corresponding to this variable is constant. This means that the generated vector representation fails to represent the change of the value that happened to the variable. That is, the noncontextual embedding models can only generate a fixed representation for one word, incapable of producing different representations according to the difference of code contexts. This can be one of the reasons that the work [11] that uses Word2Vec as the code embedding method yielded relatively low precision.

In this paper, we fill this gap by incorporating a CodeBERT-based embedding solution for vulnerable function detection, and the frequently used acronyms in this paper are summarized in Table 1. Recent research has achieved impressive results on embedding source code by applying natural language techniques such as Word2Vec, GloVe, and FastText. CodeBERT is based on the structure of the bidirectional transformer, which can help the understanding of code semantics in a relatively large context. In addition, CodeBERT inherits the mechanism of multihead attention, employing 12 parallel attention heads, which allows the model to jointly attend to vulnerable features from different representation subspaces at various positions. Therefore, as an extractor, CodeBERT has the potential of generating more rich and meaningful code embeddings compared with noncontextual embedding methods such as Word2vec, GloVe, and FastText. CodeBERT achieved a promising result in many code processing/analysis tasks such as clone detection, defection detection, and natural language code search. However, it has not been used in the context of C language vulnerability detection. In summary, the contributions of the paper are three-fold:

(i) We perform a systematic evaluation on several mainstream code embedding solutions including Word2vec, GloVe, FastText, and CodeBERT. We discover that utilizing CodeBERT as a code embedding solution yielded the best performance in terms of vulnerability detection in C open-source projects.

(ii) We apply synthetic vulnerability data derived from SARD (Software Assurance Reference Dataset) to fine tune the parameters of CodeBERT and to address the data imbalance problems frequently faced in vulnerability detection in practice. Experiments demonstrate that applying synthetic vulnerability data can further improve the usefulness of codeBERT and achieve optimized results in vulnerability detection.

(iii) We examine important parameters of CodeBERT in terms of code feature extraction and evaluate the optimal parameters identified for effective vulnerability detection.

The rest of this paper is organized as follows. Section 2 presents some existing studies for deep learning-based vulnerability detection. In Section 3, the research framework, the code representation learning, the process of fine tuning, and the impact of various sequence-length for fine tuning CodeBERT are presented. Section 4 is our evaluation and the analysis of experimental results. In Section 5, we conclude the present paper and discuss the limitations of the proposed scheme and open problems for future research.

2. Related Work

In the field of software vulnerability detection, various techniques have been proposed. There have been several survey articles providing systematic reviews of many approaches in this field from various perspectives [5, 28–32].

Meanwhile, we review several existing deep learning-based vulnerability detection studies on different neural networks (see Table 2). Many deep learning architectures include convolutional neural network (CNN) [13–16], deep belief network (DBN) [17], multilayer perceptron (MLP) [18, 19], long short-term memory (LSTM) [20–22], and gated recurrent unit (GRU) [11]. In addition, a large body of studies focuses on employing different embedding techniques for generating vector representations as input for the training process. A summary of the reviewed studies is shown in Table 3. Pradel and Sen [23] used Word2Vec for generating code vectors derived from the custom Abstract Synthetic Trees (ASTs)-based contexts. These vectors were used to train deep learning models to detect vulnerabilities in JavaScript code. The Word2Vec was applied for making vector representations from C/C++ source code [13].

Instead of using Word2Vec, Henkel et al. [24] applied the GloVe model to produce vectors learned from the Abstracted Symbolic Traces of C programs. Furthermore, FastText was used in FastEmbed [25] for vulnerability prediction based on ensemble machine learning models.

Table 1: List of the acronyms involved in the manuscript.

Acronyms	Definition	Acronyms	Definition
DL	Deep learning	NLP	Natural language processing
ML	Machine learning	MLM	Masked language modeling
SARD	Software assurance reference dataset	RTD	Replace token detection
DBN	Deep belief network	NVD	National vulnerability dataset
LSTM	Long short-term memory	CVE	Common vulnerabilities and exposures
BiLSTM	Bidirectional long short-term memory	GRU	Gated recurrent unit
ELMo	Embeddings from language models	CBOW	Continuous bag-of-words model
CuBERT	Code understanding BERT	CNN	Convolutional neural network
MLP	Multilayer perceptron		

Table 2: Reviewed studies which applied various neural networks for software vulnerability detection.

Neural network	Paper
Convolutional neural network	Harer et al. [13], Lee et al. [14], Russell et al. [15], and Wu et al. [16]
Deep belief network	Wang et al. [17]
Multilayer perceptron	Lin et al. [18] and Shar and Tan [19]
Long short-term memory	Li et al. [20], Lin et al. [21], and Lin et al. [22]
Gated recurrent unit	Lin et al. [11]

Table 3: Reviewed studies which applied various embedding techniques for software engineering.

Paper	Type of data	Embedding model	Whether to consider contextual information
Pradel and Sen [23]	150,000 JavaScript files collected from various open-source projects	Word2Vec	No
Harer et al. [13]	C/C++ packages distributed with the Debian Linux distribution C/C++ functions collected from github	Word2Vec	No
Henkel et al. [24]	19,000 API-usage analogies extracted from the Linux kernel	GloVe	No
Fang et al. [25]	Projects are extracted from open-source intelligence data such as NVD	FastText	No
Kanade et al. [26]	150k Python files from github	CuBERT	Yes
Karampatsis and Sutton [27]	150,000 JavaScript files consisting of various open-source projects	SCELMo	Yes

Recently, several contextualized embedding models have been applied for generating code representations. Kanade et al. [26] proposed CuBERT (Code Understanding BERT), which generates contextual embeddings by training a BERT model on software source code. Karampatsis and Sutton [27] proposed a model named SCELMo to generate contextual code representations. Both studies have proved that contextualized embedding models are effective for various code analysis tasks. In our work, we utilize CodeBERT, which is also a pretrained contextualized model on six programming languages, for a specific code analysis task which is vulnerability detection.

3. Methodology

3.1. Research Framework. We collect several dissimilar software projects and create the ground truth dataset; meanwhile, we select multiple synthetic vulnerabilities at the function level. Figure 1 presents our workflow. In detail, it includes three stages. Firstly, the source code is loaded and transformed to JSON format that is recognized by CodeBERT. For conventional models, the source code files are loaded and processed to generate sequence data and labels. The data is then passed to the next stage to be transformed into the code embedding vectors. These vectors will then be partitioned and fed to the GRU neural network for the training process. We utilized a system to train code vulnerability detectors for evaluating three traditional embedding methods. The system was built based on the open-source API benchmark proposed by [11]. Secondly, we feed CodeBERT with the synthetic data to obtain a fine-tuned model; the vulnerable probabilities of the test set were generated correspondingly. Thirdly, based on fine tuning, we evaluated the impact of the various parameters such as the length of the input sequence, batch size, epoch, and learning rate. A suitable input sequence length for extracting vulnerability features from programs written in C is finally determined. In order to verify the proposed method is successful in producing the desired result, we perform the tasks towards answering the following three research questions. We will address each question with the results.

> Research Question 1 (RQ1): how effective is CodeBERT when compared with other embedding methods? This research question is meaningful because one may argue that CodeBERT cannot be used to extract features of programs written in C. For

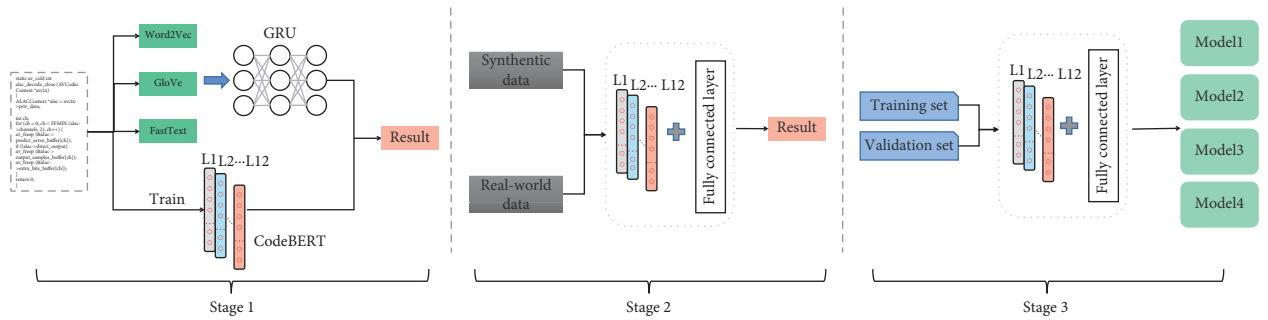

Figure 1: The workflow consists of three stages. In stage 1, we compare the performance of four models. In stage 2, a synthetic dataset derived from SARD projects is poured into the real-world dataset to fine tune the parameters of CodeBERT. In stage 3, we examine important parameters of CodeBERT in terms of code feature extraction.

answering this question, we will compare CodeBERT with other approaches, including Word2vec, GloVe, and FastText.

Research Question 2 (RQ2): how to improve the effectiveness of CodeBERT for vulnerability detection? To answer this question, we apply a real-world dataset and a synthetic dataset to fine tune CodeBERT and conduct comparative experiments.

Research Question 3 (RQ3): how does the input sequence length affect CodeBERT? We compare the performance of the fine-tuned model with various sequence lengths on the same classification tasks.

3.2. Code Representation Learning. Word2Vec, GloVe, FastText, and CodeBERT are embedding models for converting textual tokens to meaningful vectors; they have some similarities and differences. We compare four models from the following perspective.

From the perspective of model structure, in general, CodeBERT has a complex structure, while the other three models have simple structures. Word2Vec contains two models: Skip-gram and CBOW. Skip-gram employs middle words to predict nearby words, and CBOW applies context words to predict middle words. Both models have three layers, which are the input layer, mapping layer, and output layer. The hidden layer has a linear structure, so it is very fast to train. Word2Vec generates a unique vector for each word in the corpus; however, it ignores the connection between words, for example, "apple" and "apples." The two words are similar, that is, their internal morphology is close. However, both words are converted by Word2Vec; this internal form will be ignored. In order to avoid this problem, FastText uses character-level n-grams to represent a word. The FastText model also has three layers: an input layer, a hidden layer, an output layer, and the network structure is relatively simple. GloVe uses the matrix factorization method, and the training speed is also very fast. However, it only focuses on co-occurrence; the generated word vector contains limited semantic information and is only suitable for limited tasks such as similarity calculation. Compared with the three models mentioned above, the model structure of CodeBERT is much more complicated. CodeBERT is based on a multilayer two-way transformer. Specifically, CodeBERT contains 12 layers, each layer has 12 self-attention heads, the size of each self-attention head is set as 64, and the hidden dimension is set as 768. These differences in model structure and computer system will affect the detection of vulnerabilities.

From the perspective of outcome, for a word, CodeBERT can generate different vectors according to context information, however, Word2Vec, GloVe, and FastText cannot. This perspective is important because a target word may express multiple contents, meaning that an embedding model that is noncontextual would be too limited.

Context information is crucial for analyzing vulnerabilities of many different types. A motivating example is shown in Figure 2. In the vulnerable code (see Figure 2(a)), both a and b are short types, and $b = b + a$ may cause b to exceed its range. In the corrected function (see Figure 2(b)), the range of a is limited so that it can only be less than 0, which can avoid this error. In the two functions, the range of a is different. For different contexts, a certain variable has various meanings.

This paper applies CodeBERT for extracting high-level code representations for detecting vulnerable functions written in C. Traditional models such as Word2Vec, GloVe, and FastText are usually applied to convert a word into a fixed vector [33], regardless of the variation of the values of these variables. This will affect the correct understanding of code semantics. In CodeBERT, a pretrained model for natural language and programming language, two objectives are used for training. The first objective is Masked Language Modeling (MLM) and the second is to replace token detection (RTD). RTD further uses a large number of unimodal data, such as source codes without paired natural language data. For vulnerability detection, we take advantage of the second objective. CodeBERT is considered as an embedding model proposed in [34] which is based on the concept of transformer [35]. Furthermore, we leverage the component of the encoder; the encoder consists of a multihead attention layer and feedforward network. Designed to improve the ability to retain useful information, residual connections are used for two layers. CodeBERT has not learned the patterns of C language; by using transfer learning, CodeBERT makes practical and effective use of the relevant syntactic and semantic information learned from other programming languages.

```
1  short add (short a)
2  {
3      short b = 32767;
4      b = b + a;
5      return b;
6  }
```

(a)

```
1  short add (short a)
2  {
3      short b = 32767;
4      if (a<0){
5          b = b + a;
6      }
7      return b;
8  }
```

(b)

Figure 2: The example of C functions. (a) A vulnerable function. (b) Tthe revised function.

For many embedding tasks, checking the contiguous information is generally insufficient to generate semantically rich vector representations [36], vulnerable patterns usually contain declarations, assignments, control flow, and other operational logic. Hence, it is necessary to pay attention to multiple key points in the vulnerable functions. Multihead attention enables the algorithm to focus on multiple key points, which facilitates the capture of potentially vulnerable program patterns. In addition, the pattern of vulnerabilities is long-term dependent, meaning that long-distance contextual information is crucial for vulnerable function detection. For vulnerable functions, words, such as X_{i-n} and X_{i+n} of the word X_i, can also be useful. To obtain the dependencies of surrounding words of the word, the positional embedding is designed to serve this purpose. A positional encoding layer is added to the input embedding layer, and the dimension of positional encodings d_{model} is the same as the embeddings so that both parameters can be summed. Due to the source code structure being according to the rules of logic or formal argument and relating to meaning in language or logic, it is closely connected and tightly coupled. Hence, the occurrence of a vulnerable code fragment is usually having constituent parts linked or connected to either previous or subsequent code, or even to both. A tiny vulnerable code snippet usually holds multiple lines of code that can be distributed across a function block [37]. For traditional embedding models such as Word2Vec [38], it is needing much effort or skill to exactly pinpoint which line of code gives rise to software programs; however, with the structure of the transformer, it is presenting few difficulties to confirm the distance between each word. Hence, the structure of CodeBERT can make it easier for the model to detect a long-term dependency of both forward and backward by offering services, which can effectively capture the vulnerable programming patterns. CodeBERT has not been trained in C language; however, by using transfer learning, CodeBERT is capable of utilizing the relevant patterns from other programming languages to be applied to a related detection task.

We explored four embedding models for feature extraction from source code. Word2Vec: the dimension of the feature vector is set at 100 and the maximum distance between the current and predicted word within a sentence is 5. If a word appears less than 5 times, the word will be ignored. GloVe: dimensionality of the output word vectors is set at 100; the maximum distance between the current and predicted word within a sentence is 5. The number of iterations over the corpus is set at 40. The learning rate for training is set as 0.001. FastText: the dimension of the feature vector is set as 100. The maximum distance between a current and predicted word within a sentence is 5. Ignore the words with a total frequency lower than 5. The number of epochs is set at 20. CodeBERT: the number of epochs is set at 5. The default sequence length is 400. The number of data samples captured in one training session is 4, and we used the Adam optimizer throughout. The default learning rate is $2e-5$. The architecture of the CodeBERT-based embedding model is illustrated as follows:

Input layer: it feeds the open-source code to this model

Embedding layer: source codes are transformed into a low dimension vector

Encoder layer: 12 encoder layers are employed to learn high-level feature representation

Fully connected layer: only used for fine tuning

Output layer: it delivers the learned high-quality description of features for vulnerability detection

3.3. Synthetic Data Fine Tuning. In this paper, a fine-tuned solution is provided to allow CodeBERT to learn the syntax and structure of the C programming language and capture the semantics of C code. CodeBERT is pretrained in 6 programming languages which does not include the C programming language. CodeBERT requires to be familiar with the code pattern of the C language. Through transfer learning, CodeBERT does not need to use C language data to train; it only requires following the fine-tuning strategy. Fine-tuning CodeBERT requires a large number of samples and corresponding labels. Manually picking the faults on the real vulnerable functions is time-consuming. It is difficult to obtain a lot of real data to fine tune the CodeBERT. The reasons for adding an artificial synthetic dataset are as follows. First of all, the synthetic dataset has sufficient quantity and diversity. It includes basic code patterns and syntax. In addition, it possesses accurate labels which are more beneficial to the training and optimization of the model. We are using the dataset named SARD; synthetic vulnerable functions are poured off the training set and validation set, respectively, so that, in the total dataset, the vulnerability functions are one-tenth of the total functions. In addition, this also solves the issue of data imbalance for us. In reality, for multiple classification tasks, such as financial fraud and fault diagnosis, the data are often

imbalanced. The model receives the constraints of the data distribution and learns more of the features of the majority class, while ignoring the features of the minority class. This leads to a decrease in the classification performance of the model. After adding artificially synthesized vulnerability data, the ability of the model to extract vulnerability features can be improved.

Regarding the mixed dataset settings [39], we divide the real-world dataset into the training set, validation set, and test set according to the ratio of 6 : 2 : 2; the vulnerable functions in these datasets are 1189, 395, and 399, respectively. Among the synthetic dataset, 7486 and 2495 vulnerable functions are selected to pour into the training set and validation set, respectively, making sure that the vulnerable functions account for 1/10 of total functions. No synthetic vulnerable function was added to the test set.

3.4. Evaluate the Impact of Fine-Tuned CodeBERT with the Various Sequence Length. Source code functions have varying lengths when they are converted to sequences, and initial experiments suggest that using different sequence lengths exerted tremendous influence on detection results. When mapping code to vectors, the over-long codes are needed to truncate when converting codes to vectors of the fixed length for balancing between excessively long vectors and information loss. If a code sequence of a function is not long enough, it is padded with 1 s. The sequence length affects the performance of the CodeBERT model significantly. When the input sequence is too short, a large number of functions will be truncated, resulting in loss of information. The model cannot fully learn the features of functions. What it learns may only be the declaration or the definition of the variables. When the sequence is too long, a large number of functions will be filled with 1 at the end, and useless information occupies most of the space of the input sequence. In addition, there are also long-distance dependencies within the programs, given that some vulnerable functions may lie many sentences away from their locus of attention. Due to the bidirectional structure of CodeBERT, when the distance is too long, previous information may slip from the model's memory. Therefore, an appropriate input sequence length is required, and it will make the model focus exactly on the feature of the vulnerability.

The lengths of functions in the real-world dataset are depicted in Table 4; the entire real-world dataset includes 132,018 functions. We observed that approximately 44.4% of samples are within 128 elements in length. There are 90,696 functions less than 256, accounting for 68.7% of the total. Although the highest proportion is elements within 128, 128 is not necessarily the most suitable input sequence length. We need to make a trade-off between the different lengths of function code sequences.

We fine-tuned CodeBERT using different sequence lengths (block size) of 128, 256, 384, and 512, respectively. The model was firstly fine tuned, and the Adam optimizer with a learning rate of $2e-5$ is applied. Considering the limitation of GPU memory, the batch size is set to 4. Besides, a relatively small batch size helps the generalization of the model. The configuration is one of the many hyperparameters we tuned to obtain the optimal parameters. CodeBERT has been pretrained on six different programming languages, having weights to be initialized. Therefore, fine tuning is needed to allow the model to fit the specific task.

4. Experiment and Evaluation

4.1. Dataset. We believed that vulnerabilities are generally reflected in the pattern of source code, particularly at the function level. Hence, we focus on function-level vulnerabilities in this paper. The statistics of the aforementioned datasets are presented in Table 5.

4.1.1. Real-World Dataset. The real-world dataset used for evaluation consists of 12 popular open-source software projects and libraries. There are Asterisk, Httpd, ImageMagick, LibPNG, LibTIFF, OpenSSL, Pidgin, qemu, samba, VLC Player, and Xen. This dataset was built by Lin et al. [11] and was further extended to form a dual-granularity vulnerability detection dataset, providing vulnerabilities at the file and function level. For vulnerable files and functions, labels were manually attached based on the records and description of the National Vulnerability Dataset (NVD) and Common Vulnerabilities and Exposures (CVE). For the experiments, we acquire 1,983 vulnerable functions and 130,035 nonvulnerable functions from the dataset. The real-world vulnerability dataset enables classifiers to learn real-world vulnerable patterns.

4.1.2. Synthetic Dataset. The synthetic vulnerability dataset contains function samples derived from the Software Assurance Reference Dataset (SARD) project. Samples contain artificially constructed code fragments based on currently known vulnerable source code patterns. Meanwhile, each test fragment comprises one main function to guarantee the fragment of code is compilable. The synthetic dataset enables classifiers to learn the simplified and straightforward vulnerable patterns.

4.2. Experiment Settings. The implementation of CodeBERT is based on Pytorch (1.7.1) backend, and we implement the Word2Vec, GloVe, and FastText in Python using Keras with TensorFlow (1.14.0). We carry out experiments on a machine with NVIDIA GeForce GTX 1070 GPU and an Intel Core i7-6700k CPU operating at 4.00 GHz.

The comparison of four models: the real-world dataset is applied to evaluate the effectiveness of code features extracted by four embedding models, namely, Word2vec, FastText, GloVe, and CodeBERT. There are a total of 132,018 functions across 12 open-source projects, of which 1,983 functions are vulnerable. The selected source code function samples are divided into three sets, training set, validation set, and test set, with a ratio of 6 : 2 : 2. Fine-tuning stage: a contrast experiment was conducted to demonstrate the effectiveness of the proposed method that

Table 4: The statistics on code lengths for the total real-world dataset and test set involved in experiments. The functions are divided into five categories according to length.

Length of functions	<128	≥128 and <256	≥256 and <384	≥384 and <512	≥512
No. of samples (% of total sets)	58,556 (44.4%)	32,140 (24.3%)	15,052 (11.4%)	8,065 (6.1%)	18,205 (13.8%)
No. of samples (% of test set)	11,735 (44.4%)	6,446 (24.4%)	2,944 (11.1%)	1,630 (6.2%)	3944 (13.9%)

Table 5: The vulnerable functions and nonvulnerable functions are elaborated in this table. The datasets are derived from 12 open-source projects written in C programming language and the Software Assurance Reference Dataset (SARD) project which contains artificially constructed test cases. In the real-world dataset, the vulnerable functions are labeled based on the description of CVE and NVD. The first column lists the name of the dataset, the second column lists the projects, and the last two columns list the number of vulnerable functions and nonvulnerable functions, respectively.

Data source	Dataset/collection	No of functions used/collected	
		vulnerable	Nonvulnerable
Test cases from the SARD projects	C source code samples	83710	52290
Real-world open-source projects	Asterisk	94	17620
	FFmpeg	249	5549
	Httpd	57	3843
	ImageMagic	344	2361
	LibPNG	45	577
	LibTIFF	123	726
	OpenSSL	159	7004
	Pidgin	29	8547
	qemu	143	36063
	samba	26	32819
	VLC Player	44	6013
	Xen	670	8913
	Total	1983	130035

helps to fine-tune CodeBERT. Among the functions listed in Table 6, we used the training set and validation set to fine-tune CodeBERT; subsequently, we fed the model with the test set to obtain a CSV file that sorts functions according to the probability of being vulnerable. Evaluation of various parameters: we employed the previously used three sets in the fine-tuning stage as the dataset for evaluating the optimal parameters.

4.3. Evaluation Metrics. For classification tasks, precision and recall are both mainstream evaluation metrics. However, there are significantly more nonvulnerable functions than vulnerable ones. The proportion is approximately ninety six to one. The severe data imbalance may let the classifier focus on the majority class, while ignoring the minority one during the training process. To correctly monitor the performance, we apply the top-k percentage precision ($P@K\%$) and top-k percentage recall ($R@K\%$) to evaluate the effectiveness of the proposed methods. This standard of measurement is widely used in the research of information retrieval systems of top-k retrieved documents.

$P@K\%$ alludes to the data in the test set, that is, the vulnerable data, and has been successfully identified by the vulnerability detector. $R@K\%$ represents the percentage of $K\%$ data in the test set, that is, the data of vulnerability. They can be calculated by the following two mathematical expressions:

$$P@K\% = \frac{TP@K\%}{TP@K\% + FP@K\%},$$
$$R@K\% = \frac{TP@K\%}{TP@K\% + FN@K\%}. \quad (1)$$

4.4. Results and Analyses. There are various studies dedicated to addressing the issue of detecting vulnerabilities, such as the system proposed in [11] and an open-source detector named Flawfinder [40]. These systems are applied as the baselines because we have full access to the code and dataset, and Flawfinder is a well-known open-source tool which is widely used in practice. We structure the assessment by completing three research questions step by step.

RQ1: selecting the suitable embedding method can be a critical task since it can affect the performance of the vulnerability detectors. Thus, we compare the effectiveness of CodeBERT with other traditional embedding models to determine the most viable embedding model for vulnerability detection. We here report the results of applying four embedding models in detecting vulnerable functions written in the C programming language. Figure 3(a) elaborates the comparison of precision obtained from four embedding models, which are 46%, 28%, 41%, and 61%, respectively, when retrieving the top 1% of most probably vulnerable functions. Figure 3(c) shows that

Table 6: The number of vulnerable functions and nonvulnerable functions when fine tuning the parameters of CodeBERT. In the training set and verification set, aiming to make vulnerable functions account for 1/10 of the total number of functions, we added synthetic data to the original dataset.

Dataset	No of vul. Functions (real-world | SARD)	No. of total functions
Training set	8675 (1189 | 7486)	86759
Validation set	2891 (395 | 2495)	28919
Test set	399 (399 | 0)	26425

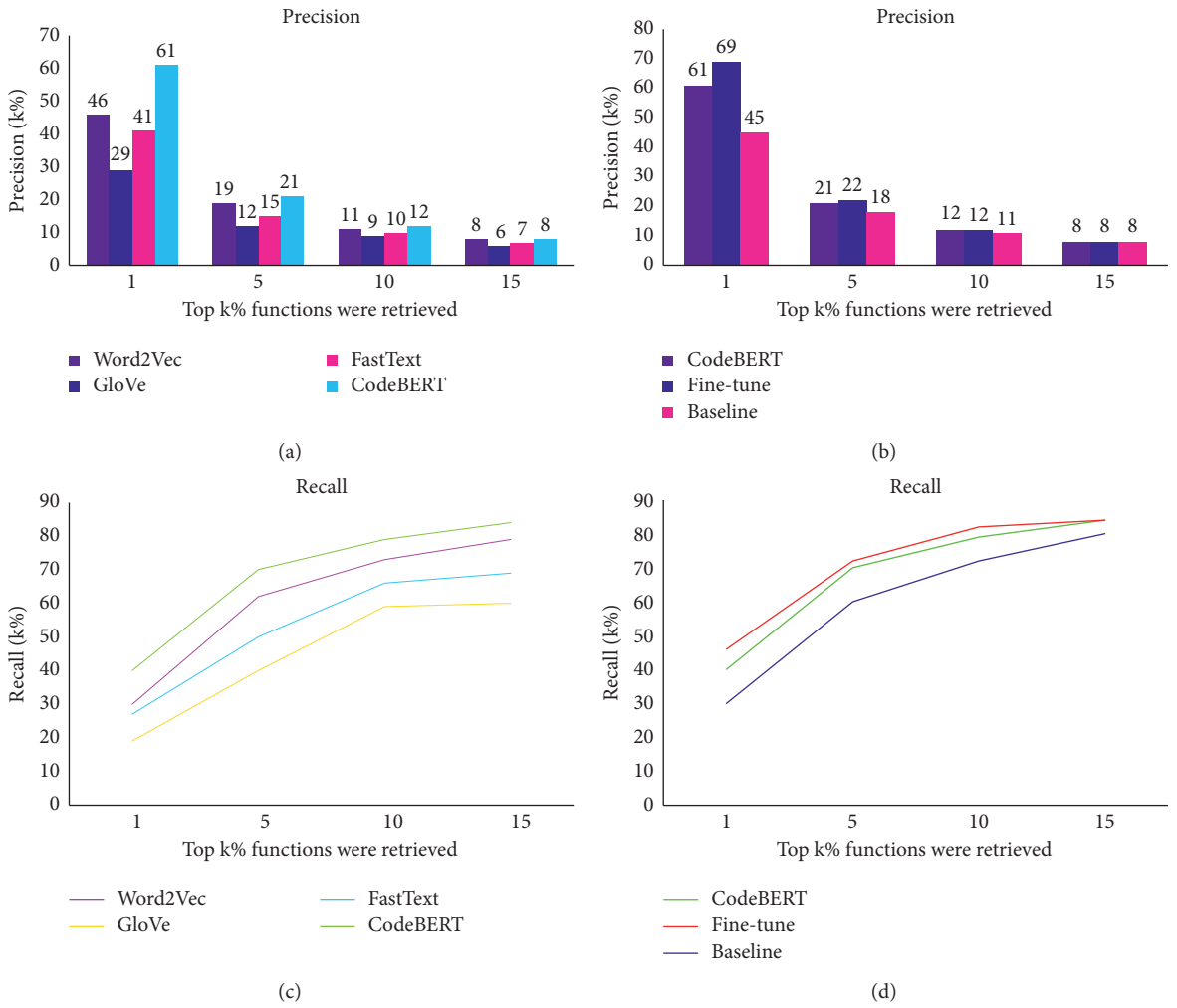

Figure 3: Results of two comparative experiments. (a) and (c) The precision and recall of several embedding methods, respectively; (b) and (d) the precision and recall of three models.

the green line which is the recall obtained by CodeBERT lies above the other three lines being the recall achieved by Word2Vec, GloVe, and FastText. CodeBERT could identify 40% of total vulnerable functions when retrieving 1% of functions. With Word2Vec, only 30% of total vulnerable functions were found. When using the FastText and GloVe models, only 27% and 20% of actual vulnerable functions could be found. This indicates that compared with the other three models, code embeddings generated by CodeBERT facilitate vulnerability detection.

In addition, we considered the computational complexity of CodeBERT. To directly measure the computational complexity of CodeBERT is a challenging task because the implementations of the encoder structure of transformer and the multihead attention mechanism are encapsulated by the deep learning framework (e.g., PyTorch). Therefore, we chose to compare the training and test time of CodeBERT with those of other embedding methods and measured the efficiency of CodeBERT and other embedding methods. By comparison, how computationally complex

codeBERT it is can be evaluated. Table 7 summarizes the training (measured in one epoch) and test time of CodeBERT and the other three models, respectively. We observe that it took CodeBERT 6720 seconds to complete one epoch during the training phase. In contrast, the training times of the three embedding models were similar, which did not exceed 300 seconds. The noncontextual model (Word2Vec, GloVe, and FastText) outperforms CodeBERT in terms of training time. This can be explained by the fact that the CodeBERT model structure is complex.

RQ2: to evaluate whether the performance of CodeBERT can be improved via fine tuning. We add a fully connection layer at the bottom of the output. Meanwhile, to further fine tune the parameters, we apply a synthetic vulnerability dataset derived from software.

Assurance Reference Dataset (SARD) project: the synthetic dataset contains artificially defined test cases to simulate vulnerable code patterns. As shown in Figure 3(b) and Figure 3(d), when retrieving 1% of vulnerable functions, we observe that, after fine tuning, there is an 8% of precision improvement, and the fine-tuned model could find 84% of total vulnerable functions when returning 15% of potentially vulnerable functions. In addition, the fine-tuned model is more effective than previous models such as the vulnerability detector [11]. When retrieving 1% of total functions, the improvement in each of the metrics is substantial, that is, 24% in precision and 16% in recall. It was proved that the performance of the detectors was improved by using the fine-tuned approach.

RQ3: the purpose of this experiment is to understand how much suitable sequence length help. We construct experiments to demonstrate the effectiveness of suitable sequence lengths for vulnerability detection. Tables 8 and 9 show the results of models with various sequence lengths. The testing set contains 26,425 functions among which there are 399 vulnerable functions. For detecting software vulnerabilities written in C language, we can conclude that the most satisfying sequence length is 256. Because whether it is retrieving the top 1%, 5%, 10%, or 15%, the model works best when the sequence length is 256. When examining the top 1% of total functions, the model with a sequence length of 128 only incurs a precision of 26%.

For a fair comparison, we analyzed the results obtained above with a baseline. We observe that the output of Flawfinder is ranking the functions according to their vulnerable level. The results of the proposed method are ranking functions according to the probability of being vulnerable. Hence, we applied the abovementioned performance metrics to describe both detectors. We here report the comparison between their effectiveness in detecting real world.

Vulnerabilities: Tables 8 and 9 show the comparison results; the model with a sequence length of 256 substantially outperforms the FlawFinder. The model with a sequence length of 256 incurs an $P@1\%$ of 70% and $R@1\%$ of 47%; however, when we retrieved the top 1% of functions in the test set that contains 399 vulnerable functions, Flawfinder only achieved 3% precision and 2% recall. Retrieving 5%, 10%, and 15% of functions ranked by the vulnerable level, the results of the Flawfinder are also poor. This phenomenon further highlights the effectiveness of the CodeBERT-based embedding solution for vulnerability detection.

5. Discussion

This section discusses the balance between model effectiveness and model complexity. To directly measure the computational complexity of CodeBERT is a challenging task because the implementations of the encoder structure of transformer and the multihead attention mechanism are encapsulated by the deep learning framework. We take an epoch as an example to show the time overhead incurred by the process of training. The training time corresponding to these four models (CodeBERT, Word2Vec, GloVe, and FastText) was, respectively, 6720 s, 287 s, 285 s, and 286 s; it is clear that CodeBERT takes the longest. The underlying reason is that the structure of CodeBERT is more complex than the other three models, and CodeBERT has a large capacity; it may save a large body of information that other models have not captured, such as some potential code patterns and semantic features. In addition, the parameter is complex and requires a lot of calculation, so it takes more time. Even so, CodeBERT can improve the precision and recall of vulnerability detection tasks; it is still worth selecting.

In addition, when using CodeBERT extracting code features, different sequence lengths exhibited varying performances; the statistic is shown in Tables 8 and 9. In the current task, the balance between functions length and the input sequence length is also a challenge. We discuss the possible causes of performance behavior of different sequence lengths. Compared with other lengths, the sequence length of 256 outperformed the real-world dataset. The underlying reason is that the vulnerable functions contain information such as a head file, variable declaration, parameters, logic code, and return value. If the sequence length is too short, the obtained features may only include information such as header files and variable declarations. Vulnerability features are usually hidden in the logic code. Therefore, the features of the vulnerabilities may be cut off, and the model learns all the useless features. When the sequence length is longer than 256, performance is not satisfactory; the reason for this phenomenon may be that the length of most vulnerable functions is within 256; if the sequence length is set too long, the sequence will automatically be filled with 1. Too much irrelevant information will interfere with the model's judgment on the features of the vulnerability; meanwhile, models will focus on irrelevant information. In general, setting the sequence length to 256 can greatly reduce the redundancy and loss of information.

Table 7: The complexity of training and test when we compare four models.

Models	Training time per epoch (s)	Test time per epoch (s)	Number of epochs	Total training time with all epoch completed (s)
CodeBERT	6720	600	5	33600
Word2Vec	287	32	150	43050
GloVe	285	32	150	42750
FastText	286	32	150	42900

Table 8: Test precision of various sequence lengths against Flawfinder on the same classification tasks.

Precision calculated when top-k% functions were retrieved					
		1 (%)	5 (%)	10%	15 (%)
Different sequence lengths	Block size = 128	26	7	6	5
	Block size = 256	70	22	12	8
	Block size = 384	56	18	10	9
	Block size = 512	63	20	12	8
Flawfinder		3	3	7	7

Table 9: Test recall of various sequence lengths against Flawfinder on the same classification tasks.

Recall calculated when top-k% functions were retrieved					
		1 (%)	5 (%)	10 (%)	15 (%)
Different sequence length	Block size = 128	12	25	37	46
	Block size = 256	47	74	81	86
	Block size = 384	37	60	69	78
	Block size = 512	41	66	78	85
Flawfinder		2	2	45	45

6. Conclusions and Future Work

This paper proposes an embedding solution for vulnerability detection which is based on CodeBERT. CodeBERT is not familiar with the syntax and semantic of the C language; however, it has been pretrained on other programming languages and has great potential of learning effective features of the C language. We have employed C open-source projects and manually constructed functions written in C to fine-tune CodeBERT. Meanwhile, we have constructed a useful real-world dataset for evaluating and estimating the ability and quality of the solution and other deep learning-based vulnerability detectors that will be expanded in the future. The experimental results show that the proposed embedding solution can achieve precision that exceeded expectation when returning K (1, 5, 10, 15)% of total functions, indicating that the approach is capable of facilitating the detection of vulnerabilities in C open-source projects.

The present scheme can be further improved by addressing the following limitations. Firstly, CodeBERT consists of 12 encoder layers and has approximately 110 million parameters, which are expensive to train and deploy. Therefore, proposing a lightweight model could be valuable. Secondly, the present embedding solution for vulnerability detection is limited to dealing with the software code written in C# and C++. Further research could be conducted to adapt to more programming languages. Thirdly, there is still lack of a large real-world dataset providing multiple detection granularities. Further research effort could be developing an automated vulnerability data labeling solution to speed up the data collection process.

Data Availability

The public dataset is available online for research (https://cybercodeintelligence.github.io/CyberCI/).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

Xue Yuan and Guanjun Lin contributed equally to this paper.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant no. 62062069), Optoelectronic Information Technology Key Laboratory Open Project Fund of Yunnan Province, China, under Grant YNOE-2020-01, and Natural Science Foundation Project of Fujian Province, China, under Grant 2021J011131.

References

[1] M. Wang, T. Zhu, T. Zhang, J. Zhang, S. Yu, and W. Zhou, "Security and privacy in 6g networks: new areas and new challenges," *Digital Communications and Networks*, vol. 6, no. 3, pp. 281–291, 2020.

[2] Y. Miao, C. Chen, L. Pan, Q.-L. Han, J. Zhang, and Y. Xiang, "Machine learning-based cyber attacks targeting on controlled information," *ACM Computing Surveys*, vol. 54, no. 7, pp. 1–36, 2022.

[3] X. Chen, C. Li, D. Wang et al., "Android hiv: a study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2019.

[4] G. Lin, W. Xiao, L. Y. Zhang, S. Gao, Y. Tai, and J. Zhang, "Deep neural-based vulnerability discovery demystified: data,

model and performance," *Neural Computing and Applications*, pp. 1–14, Springer, New York, NY, USA, 2021.

[5] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: a survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1397–1417, 2018.

[6] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: a comparison of software vulnerability discovery processes," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 374–391, IEEE, San Francisco, CA, USA, May 2018.

[7] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM Transactions on Networking*, vol. 23, no. 4, pp. 1257–1270, 2014.

[8] S. Liu, G. Lin, L. Qu et al., "Cd-vuld: CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[9] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescape, "Mobile encrypted traffic classification using deep learning: experimental evaluation, lessons learned, and challenges," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 445–458, 2019.

[10] T. O'shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017.

[11] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: a benchmark," in *Proceedings of the International Conference on Information and Communications Security*, pp. 219–232, Springer, Beijing, China, December 2019.

[12] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the International Conference on Machine Learning*, pp. 2123–2132, PMLR, Lille, France, July 2015.

[13] J. A. Harer, L. Y. Kim, R. L. Russell et al., "Automated software vulnerability detection with machine learning," 2018, https://arxiv.org/abs/1803.04497.

[14] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect soft-ware weakness," in *Proceedings of the KSII the 9th international conference on internet (ICONI) 2017 symposium*, Vientien, Laos, December 2017.

[15] R. Russell, L. Kim, L. 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)*, pp. 757–762, IEEE, Orlando, FL, USA, December 2018.

[16] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proceedings of the 2017 3rd IEEE international conference on computer and communications (ICCC)*, pp. 1298–1302, IEEE, Chengdu, China, December 2017.

[17] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 297–308, IEEE, 2016.

[18] G. Lin, J. Zhang, W. Luo et al., "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2469–2485, 2021.

[19] L. K. Shar and H. B. K. Tan, ""Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the2012 27th IEEE/ACM international conference on automated software engineering*, pp. 310–313, IEEE, Essen, Germany, September 2012.

[20] Z. Li, D. Zou, S. Xu et al., "Vuldeepecker: a deep learningbased system for vulnerability detection," 2018, https://arxiv.org/abs/1801.01681.

[21] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. . Xiang, "Poster: vulnerability discovery with function representation learning from unlabeled projects," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2539–2541, Dallas, TX, USA, October 2017.

[22] G. Lin, J. Zhang, W. Luo et al., "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[23] M. Pradel and K. Sen, "Deep learning to find bugs," *TU Darmstadt, Department of Computer Science*, vol. 4, no. 1, 2017.

[24] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 163–174, Boise, ID, USA, November 2018.

[25] Y. Fang, Y. Liu, C. Huang, and L. Liu, "FastEmbed: p," *PLoS One*, vol. 15, no. 2, Article ID e0228439, 2020.

[26] A. Kanade, P. . Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the International Conference on Machine Learning*, pp. 5110–5121, PMLR, Shenzhen, China, February 2020.

[27] R.-M. Karampatsis and C. Sutton, "Scelmo: source code embeddings from language models," 2020, https://arxiv.org/abs/2004.13214.

[28] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "SOftware vulnerability analysis and discovery using deep learning techniques: a survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020.

[29] S. K. Singh and A. Chaturvedi, "Applying deep learning for discovery and analysis of software vulnerabilities: a brief survey," *Advances in Intelligent Systems and Computing*, vol. 1154, pp. 649–658, 2020.

[30] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: a survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1744–1772, 2019.

[31] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Computing Surveys*, vol. 53, no. 6, pp. 1–36, 2020.

[32] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, https://arxiv.org/abs/1301.3781.

[34] Z. Feng, D. Guo, D. Tang et al., "Codebert: a pretrained model for programming and natural languages," 2020, https://arxiv.org/abs/2002.08155.

[35] A. Vaswani, N. Shazeer, N. Parmar et al., "Attention is all you need," in *Proceedings of the Advances in neural information processing systems*, pp. 5998–6008, Long Beach, CA, USA, December 2017.

[36] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in

Proceedings of the International conference on learning representations, New Orleans, Louisiana, May 2019.

[37] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," 2018, https://arxiv.org/abs/1808.09588.

[38] J. Zhang, L. Pan, Q.-L. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: a survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 3, pp. 377–391, 2022.

[39] C.-L. Zhang, J.-H. Luo, X.-S. Wei, and J. Wu, "In defense of fully connected layers in visual representation transfer," in *Pacific Rim Conference on Multimedia*Springer, New York, NY, USA, 2017.

[40] FlawFinder: https://dwheeler.com/flawfinder/.