BBVD: A BERT-based Method for Vulnerability Detection

Weichang Huang[1], Shuyuan Lin[2]*, Chen Li[3]
College of Information Science and Technology / the College of Cyber Security, Jinan University, Guangzhou, China[1]
School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China[2,3]

***Abstract*—Software vulnerability detection is one of the key tasks in the field of software security. Detecting vulnerability in the source code in advance can effectively prevent malicious attacks. Traditional vulnerability detection methods are often ineffective and inefficient when dealing with large amounts of source code. In this paper, we present the BBVD approach, which treats high-level programming languages as another natural language and uses BERT-based models in the natural language processing domain to automate vulnerability detection. Our experimental results on both SARD and Big-Vul datasets demonstrate the good performance of the proposed BBVD in detecting software vulnerability.**

***Keywords*—Vulnerability detection; BERT; software security**

I. Introduction

Software vulnerability is security flaw that exists in software [1]. When developers develop software, security issues such as logic error, buffer overflow, array out-of-bound access and other errors can easily occur in the source code. Vulnerability can be exploited for privilege escalation, leakage of secret data, denial of service and many other types of attacks [2], which put the integrity and availability of the software and the computer system at risk. The size and complexity of modern software are increasing dramatically, which also diversifies the types and causes of vulnerability. Thus, the traditional vulnerability detection methods are facing new challenges.

Traditional source code vulnerability detection depends on manual code auditing [3], relying on the expertise of the security personnel to review the software source code. However, in the face of the massive amount of code in complex software, the workload of manual audit vulnerability is large and the effectiveness of detection relies strongly on the amount of a priori knowledge of security personnel. It is unrealistic to rely entirely on manual audits to discover existing types and possible variants of vulnerability types. Now, automated vulnerability detection is becoming an important supplement to manual audits.

There are many similarities between high-level programming languages and natural language [4], [5]. High-level programming languages such as C/C++ that inherit the syntax of natural language have a defined syntax and semantics. Long sentences in natural language consist of words and phrases. A programming language also consists of a series of instructions. If these instructions are treated as words in a natural language, different combinations of instructions produce more complex operations, which correspond to long sentences in natural language. The choice of different words, the difference in word placement and the contextual connection all affect the semantics of sentences in natural language. In the case of programming language, the placement and combination of instructions also affect the logic of the programming language. Therefore, considering these similarities, we can use language models in NLP for software vulnerability detection.

The BERT-based model allows for parallelized computation in the model. Also, multiple experimental results show that BERT-based models [6]–[9] have outperformed existing models in natural language processing, including recurrent neural network (RNN) architectures. Therefore, BERT-based language models are more effective than RNNs and we choose to use BERT-based models to detect software vulnerability. Our contributions in this paper are as follows:

- We propose a method called BBVD for detecting vulnerability in C/C++ source code by using BERT-based models.
- We verify that the BERT-based language models used in the BBVD approach can be migrated to areas outside of the natural processing domain such as vulnerability detection.
- Our experimental results on both SARD [10] and Big-Vul [11] datasets show that BBVD outperforms the state-of-the-art method such as SySeVR [12].

The rest of the paper is organized as follows: Section I presents the related works on BERT-based models and software vulnerability detection methods. Section III describes the proposed BBVD method and the flow for the software vulnerability detection. Section IV describes the experimental setup of datasets and BERT-based models used in our work. Section V reports the experimental results and performance of the BERT-based models. Section VI concludes our paper.

II. Background and Related Work

A. BERT beyond NLP

In recent years, some classical methods have yielded good performance using model optimization strategies to process data containing noise and outliers, such as graph-based [13], [14] and information- and cue-based [15], [16] strategies. Furthermore, neural networks can be more efficient by fusing attention mechanisms when processing large amounts of input information. These networks using attentional mechanisms have been used in a wide range of tasks such as image text matching [17], visual sentiment analysis [18], video question

*Corresponding authors.

answering [19], generative adversarial task [20], multi-task travel route planning [21], etc.

Attention mechanisms are also used in BERT to focus each element in the input sequence on other elements. The paper [6] demonstrates that BERT pre-training facilitates almost all types of NLP tasks (except generative models). The question-and-answer (QA) domain is an important area of NLP. BERT in the field of QA is more effective than the previous methods, and the experimental results of some papers [22]–[25] in question retrieval and answer determination show that a significant improvement can be achieved by using BERT or attention mechanisms. The application of BERT to the reading comprehension task also had a huge impact on the various original techniques [26]. The effectiveness of works [27] using BERT in document retrieval tasks similarly demonstrates the usefulness of BERT in the field of information retrieval. In addition to these domains, BERT has also shown significant enhancements in areas such as text summarization [28], text classification [29], [30] etc.

The BERT can also be migrated to domains other than natural language processing. Inspired by pre-training models in the BERT model, Kanade et al. [31] proposed the CuBERT (CodeUnderstandingBERT) model designed with two pre-training tasks of predicting masked tokens and whether two logical lines of code are related to each other in a contextual sentence. The CodeBERT model proposed by Feng et al. [4] used bimodal data of natural and programming languages. They pre-trained CodeBERT in six programming languages and showed optimal performance on the NL-PL downstream task. The above two code pre-training models only consider codes as token sequences, ignoring the structural information in the codes. Guo et al. [32] proposed the GraphCode-BERT model using the data flow information of codes in the pre-training process, which significantly improved the performance of four downstream tasks, including code search, code clone detection, code translation and code summarization.

There have been studies using the BERT model to detect software vulnerability. Ziems et al. [33] fine-tuned the 100,000 C/C++ source code files on a pre-trained model known as BERT base and tested the fine-tuned model with over 100 types of vulnerability. Their work shows the feasibility of BERT for vulnerability detection. Although our approach is also based on BERT related model, there are several differences compared to their approach: (i) Their input is source code files without comments. Ours is the code slice that can better represent the syntax and semantic information of the program; (ii) They use pre-trained model which based on an English Wikipedia dataset containing 2.5 billion words, we use the C/C++ code slice to pre-train and then generate a pre-trained model; (iii) They consider only BERT, we have experimented on other BERT-based models; (iv) They only experimented on the synthetic dataset SARD, we do experiments on both the synthetic dataset SARD and the real dataset Big-Vul.

B. Vulnerability Detection Methods

Source code vulnerability detection methods can be divided into two categories: static and dynamic methods. Static methods do not require the execution of source code and can detect vulnerability by combining information from source code files, control flow graph [37], program dependency graph [42], LLVM IR [38], etc. Dynamic methods obtain the execution path and data flow of program crashes by executing the code file to obtain the heap, stack, registers and other information of the actual running program. Typical dynamic methods are fuzz testing and taint analysis, these methods applied to large project source code files have problems such as low path coverage and path explosion, which require a lot of computational resources. Therefore, we focus on static methods.

Static detection methods can be divided into two categories: traditional methods and machine learning based methods. Traditional detection methods are mainly based on code similarity and pattern match. Code similarity-based detection is suitable for vulnerability caused by code clone. It has a high false negative when the cloned code itself is not vulnerable but other code fragments are flawed. There are four types of code similarity-based detection methods: text-based, lexicon-based [34]–[36], syntax-based [39]–[41] and semantic-based [42]–[45]. Text-based and lexicon-based methods are simple to implement and can detect the source code of almost all programming languages. However, recognizing only textual information or lexical information will lose the syntactic and semantic information of the source code, which leads to a low accuracy rate. Syntax-based and semantic-based methods can identify syntactic and semantic information of programs with high detection accuracy. But as program size increases, the subtree matching algorithm for detecting abstract syntax tree similarity and the subgraph matching algorithm for detecting program dependency graph similarity grow dramatically in time and space complexity.

In contrast, pattern-based approaches require a large number of experts to define rules for matching vulnerability manually. The subjectivity of experts may lead to high false positives and high false negatives. For a specific vulnerability, the corresponding vulnerability patterns can be extracted. The pattern matching method describes these vulnerability patterns in a specific syntax. Then the source code is compared and detected after a series of processing. Methods that use pattern matching to detect vulnerability include PMD, Coverity Prevent [46], etc.

Researchers have started applying machine learning to detect software vulnerability due to increased computer computing power. Yamamoto et al. [47] applied machine learning algorithms such as naive bayes model to the vulnerability descriptions of the National Vulnerability Database (NVD) [48] for vulnerability detection. Their proposed method shows good prediction performance on a dataset containing more than 6000 vulnerable source codes. Toloudis et al. [49] combine techniques related to text analysis and principal component analysis to process the vulnerability descriptions of the NVD and perform experiments on a large dataset containing 70,678 vulnerable source code. Spanos et al. [50] use classical machine learning methods on the same dataset to predict software vulnerability, the experimental results achieved an accuracy of about 80%.

As the number of public vulnerabilities grow, it becomes possible to use deep learning models to extract abstract feature representations from vulnerability samples. Not all information in the source code file is helpful for software vulnerability detection. Too much irrelevant information may interfere with

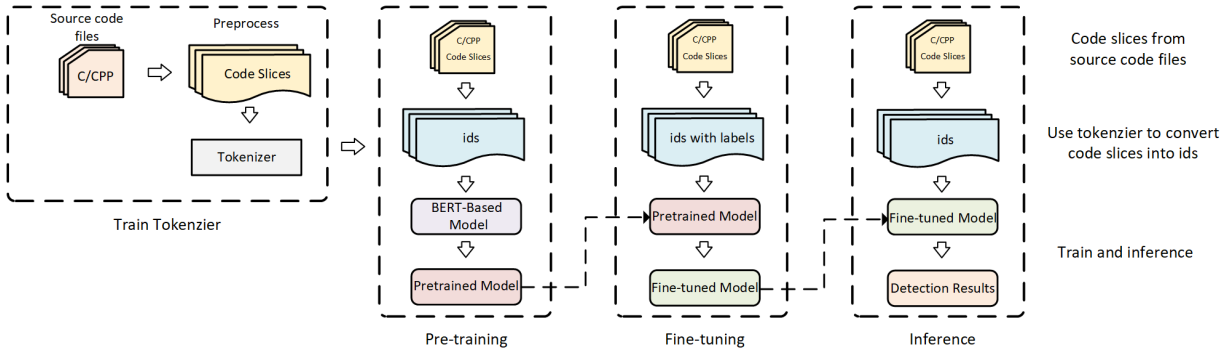

Fig. 1. Framework of BBVD

prediction. Program slicing techniques are widely used in vulnerability detection tasks which can obtain code fragments related to vulnerability. Li et al. [5] proposed the concept of code gadgets to serve as an intermediate representation of source code for obtaining semantic information related to vulnerability. They predicted the existence of vulnerability related to API and library function calls in programs based on code gadgets and BiLSTM. Li et al. [12] proposed code slices based on code gadgets combined with program dependency graphs. The code slices can represent the vulnerability syntax and semantic features of the program and fed into the BiGRU model to obtain vulnerability detection results.

In addition to using models in NLP to detect software vulnerability, some studies use graph neural networks to detect vulnerability. The Devign model proposed by Zhou et al. [51] uses an abstract syntax tree as a skeleton. The model incorporates the control flow and data flow information to generate a joint graph representation. A gated graph neural network and a convolutional neural network model are used for graph classification to detect vulnerable code. Chakraborty et al. [52] proposed ReVeal using a combination of gated graph neural networks and multilayer perceptrons. Their experiments showed that adding multilayer perceptron modules to gated graph neural networks helps in vulnerability detection.

III. Design of BBVD

In this section, we present a brief overview of the BBVD method. Fig. 1 illustrates the detection pipeline which trains from scratch. We divide the whole process into four phases: training the tokenizer, pre-training phase, fine-tuning phase and inference phase. In the first stage, we extract code slices from the source code files and then train the tokenizer based on the tokens in the code slices. The second stage converts the code slices into ids using the tokenizer and pretrains on the ids using a BERT-based model. The ids are token indices, numerical representations of tokens building the sequences that will be used as input by BERT-based models. The third stage labels the code slices and defines the downstream task as the classification for fine-tuning. The fourth stage uses datasets not identified by the model for inference.

A. Train Tokenizer Phase

In the stage of training tokenizer, we need to extract code slices from the source code files. Specifically, code slices are syntax-based vulnerability candidates (SyVCs) and semantics-based vulnerability candidates (SeVCs) which proposed by Li et al. [12]. In order to get the code slices, we need to use the open-source code analysis tool joern to parse the source code files and obtain the corresponding control flow graph and program dependency graph. Then, parsing out the call graph of function based on the CFG and PDG to extract array-related, pointer-related, api-related and arithmetic-related code slices. Furthermore, keeping the original name of keywords in the programming language and mapping the user-defined variable and function names one-to-one to symbolic names (e.g. “variable_0”, “variable_1”, “func_0”, “func_1”) to reduce the interference of user-defined function names and variable names. Fig. 2 shows different types of code slices extracted from the same source file. Finally, a tokenizer is generated based on these code slices.

B. Pretrain Phase

During the pre-training phase, the code slices of the pre-training dataset are transformed into ids using the tokenizer generated in the first phase. Fig. 3 shows an example of converting code slices and ids to each other. The ids are filled or truncated according to the max position embeddings in the model network parameter config. That is, when the length of ids is less than the max position embeddings, special token ids (e.g. “$\langle pad \rangle$” correspond id is 1) are used to fill in the right side of ids. When the length of ids exceeds the max position embeddings, the redundant part to the right of the ids is truncated. The BERT-based model trained with unsupervised learning to learn the context of code slices and their syntax and semantics. We use language masking model approach to train the BERT-based model. The language masking model masks some of the input locations based on probabilities (the original token is replaced with “$\langle mask \rangle$”) and trains the model to predict the masked parts using the remaining parts. Meanwhile, the model is optimized based on the original masked tokens and the predicted ones.

C. Finetune and Inference Phase

After pre-training is completed, we need to perform the downstream task on the pre-trained model. Since vulnerability detection can be regarded as a binary classification, we define the downstream task as a classification task. Similarly, the code slices extracted from the fine-tuned dataset are transformed

Fig. 2. Code slices

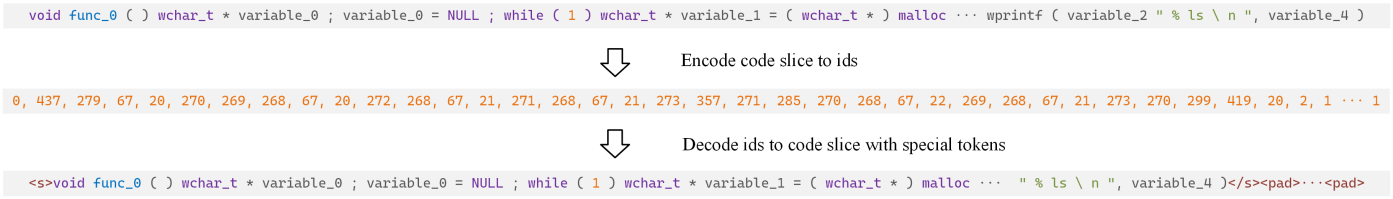

Fig. 3. Interconversion between the ids and code slices

into ids using the trained tokenizer. In order to make the classification task work properly, we need to add the appropriate labels to the ids at this phase. Label 0 means the code slice is a normal sample and label 1 means a vulnerable sample. Then, the fine-tuning dataset is split into a training set and an evaluation set. The pre-trained model is fine-tuned on the training set and evaluated on the evaluation set. Finally, generating the fine-tuned model. Using the fine-tuned model can inference whether the source code file is vulnerable or not. Repeat the steps of extracting code slices and transforming them into ids. Input the ids into the fine-tuned model can get the detection results.

IV. Experiment Setup

We designed follow three Research Questions (RQs).

- RQ1: Do the BERT-based models work better than the RNN-based models for vulnerability detection?
- RQ2: Do the use of irrelevant datasets in the pre-training and fine-tuning phases interfere with the detection performance of the model?
- RQ3: Do the different value of max embedding lengths (e.g., the max lengths of ids) have an influence on the detection effect of BERT-based models?

We implement BERT-based models by using the huggingface [53] with PyTorch 1.8.0. The computer running the experiment has an NVIDIA GeForce GTX 2080TI GPU and 16 Intel(R) Xeon(R) Gold 5117 CPUs running at 2.00 GHz.

A. Datasets

We experiment on two vulnerability datasets: SARD and Big-Vul. SARD is a synthetic dataset in which most of the testcases are written for academic and research purposes to reproduce specific vulnerability. SARD contains a batch of testcases composed of code in C/C++. These testcases are classified into different CWEs (Common Weakness Enumeration) according to the associated vulnerability types. They are classified into good, bad, and mixed types according to whether the code file is a vulnerability, among which mixed means that the code file contains both the vulnerability and the fix code. For the testcases of bad and mixed types containing vulnerability, the additional information about the line number where the vulnerability is located is also given. The entire sard dataset consists of 10,682 cpp source code and 24,633 c source code files.

Big-Vul dataset is a real dataset which collects testcases from the commit records related to CVEs(Common vulnerability and Exposures) that have appeared in the git repositories of 310 well-known C/C++ open source projects between 2002 and 2019 including Linux, Chrome, etc. All the data is provided in the form of CSV and each testcase containing information, such as the project name, associated CVE number, description, code content, and function differences before and after the commit. All vulnerability are categorized into ten types such as denial of service attacks, memory corruption, and privilege escalation. Each testcase labeled as vulnerable is one of the combinations of these types. To facilitate extracting

code slices, the samples in Big-Vul need to be filtered. The function differences in commit contain the corresponding code lines which change after the patch. We select the data where the number of code lines change by less than 100. The total number of filtered data is 16,608, of which 16,436 are c source code files and 172 are cpp source code files. Due to the complexity of the vulnerability types in the Big-Vul dataset, we called the memory corruption-related vulnerability data from filtered Big-Vul dataset as Big-Vul-MemCorr and the entire filtered Big-Vul dataset as Big-Vul-All.

B. Models

We conducted experiments on five models. Specifically, this experiment contains two RNN-based models (BiGRU and BiLSTM) and three BERT-based models (RoBERTa, DistilBERT and MobileBERT). Table I shows some important network structure-related related parameters of BERT-based models used in this paper.

TABLE I. NETWORK STRUCTURE RELATED PARAMETERS

Model	Hidden Size	Hidden Layers	Attention Heads	Intermediate Size	Max Position Embeddings
RoBERTa	768	12	12	3072	514
DistilBERT	768	6	12	3072	512
MobileBERT	512	24	4	512	512

RoBERTa. This model mainly based on BERT with several adjustments: longer training time with larger batch size and more training data; longer training sequence; dynamic adjustment of masking mechanism.

DistilBERT. It's proposed for the most popular BERT pre-training model with a 40% reduction in model size and 60% faster inference operations while retaining 97% of the performance.

MobileBERT. It compresses the BERT model which reduces the model size by a factor of three to four and increases the speed by a factor of four to five with little loss of effect, allowing a variety of NLP applications to be easily deployed on mobile.

C. Evaluation Metrics

The effectiveness of vulnerability detection can be evaluated using the following metrics: accuracy (*A*), precision (*P*), recall (*R*), F1-score (*F1*) and Mathews Correlation Coefficient (*MCC*). True positive (*TP*) indicates the number of samples with vulnerability detected. False Positive (*FP*) indicates the number of samples without vulnerability but detected as such. True Negative (*TN*) indicates the number of samples without vulnerabilit detected. False Negative (*FN*) denotes the number of samples with vulnerability but detected as not having vulnerability. The metric $A = \frac{TP+TN}{TP+FP+TN+FN}$ is the number of correctly predicted samples out of all the samples. The metric $P = \frac{TP}{TP+FP}$ represents all the samples that are declared to be vulnerable but what percentage of them are actually vulnerable. The metric $R = \frac{TP}{TP+FN}$ represents all the samples that are actually vulnerable but what percentage declared vulnerable. The metric $F1 = 2 * \frac{P*R}{P+R}$ is used to measure test accuracy, which is a weighted average of the precision and recall. The *F1* score is 1 when it's best and on 0 when it's worst. The $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$ returns a value between negative one and positive one. A coefficient of positive one represents a perfect prediction, zero means no better than random prediction and negative one indicates total disagreement between prediction and actual label.

V. EXPERIMENTS AND RESULTS

In this section, we conduct experiments on SARD, Big-Vul-All, Big-Vul-MemCorr datasets based on the design of BBVD described in section III. Comparative experiments are conducted using the models and evaluation metrics described in section IV.

A. Preprocess

1) Code slices:: For the SARD, Big-Vul-All and Big-Vul-MemCorr datasets, we extracted four types of code slices, corresponding to the API function call, array usage, pointer usage and arithmetic expression, respectively.

- The API function call related code slices: We extract 135,145 from SARD, 68,686 from Big-Vul-All and 6,381 from Big-Vul-MemCorr.
- Array usage related code slices: We extract 56,346 from SARD, 11,566 from Big-Vul-All and 4,953 from Big-Vul-MemCorr.
- Pointer usage related code slices: We extract 318,912 from SARD, 161,596 from Big-Vul-All and 4,525 from Big-Vul-MemCorr.
- The arithmetic expression related code slices: We extract 7,810 from SARD, 16,842 from Big-Vul-All and 5,192 from Big-Vul-MemCorr.

Since there are still duplicates in the above extracted code slices, we need to de-duplicate these data. For the SARD dataset, the API function call related code slices are reduced from 135,145 to 69,432; the array usage related code slices are reduced from 56,346 to 24,680; the pointer related code slices are reduced from 318,912 to 128,492; the arithmetic related code slices are reduced from 7,810 to 6,956. The total number of non-duplicate data in the SARD dataset is 229,560, of which 44,447 are vulnerable and 185,113 are non-vulnerable. For the Big-Vul-All dataset, the API function call related code slices are reduced from 68,686 to 29,865; the array usage related code slices are reduced from 11,566 to 6,151; the pointer related code slices are reduced from 161,596 to 101,804; the arithmetic related code slices are reduced from 16,842 to 15,476. The total number of non-duplicate data in the Big-Vul-All dataset is 153,287, of which 19,908 are vulnerable and 133,379 are non-vulnerable. Table II describes the number of extracted code slices for the SARD, Big-Vul-All and Big-Vul-MemCorr datasets.

2) Tokenizer:: Tokenization is an important step in natural language processing which breaks long texts such as sentences, paragraphs, and articles down into word-based data structures for subsequent processing and analysis work. In order to translate code slices into vectors that can be recognized by the BERT-based models, we need a tokenizer to translate them into vectors which can be recognized by BERT-based models.

TABLE II. NUMBER OF CODE SLICES EXTRACTED FROM DATASETS

Datasets	Slices Type	Total Slices	De-duplicated Slices	Vul Slices	Non-Vul Slices
SARD	API Related	135145	69432	12786	56646
SARD	Array Related	56346	24680	7195	17485
SARD	Pointer Related	318912	128492	11680	116812
SARD	Arithmetic Related	7810	6956	12786	56646
Big-Vul-All	API Related	68686	29865	4919	24946
Big-Vul-All	Array Related	11566	6151	325	5826
Big-Vul-All	Pointer Related	161596	101804	13406	88398
Big-Vul-All	Arithmetic Related	16842	15476	1258	14191
Big-Vul-MemCorr	API Related	6381	2064	538	1526
Big-Vul-MemCorr	Array Related	4953	1067	204	863
Big-Vul-MemCorr	Pointer Related	4525	3154	657	2587
Big-Vul-MemCorr	Arithmetic Related	5192	2911	440	2471

Specifically, we used two tokenizers in experiments, one based on byte-pair encoding and the other based on wordpiece. DistilBERT and MobileBERT model use tokenizer based wordpiece. RoBERTa use tokenizer based byte-pair encoding.

- Tokenizer based byte-pair encoding (BPE). BPE is an algorithm for encoding based on byte pairs. The algorithm is described as a cascading iterative process in which the most frequent pair of characters in a string is replaced by a character that does not appear in the string.
- Tokenizer based wordpiece. The wordpiece method is very similar to BPE in general, except that when selecting characters for merging, BPE uses the highest frequency, while wordpiece uses the highest probability.

B. Experiments for RQ1

To answer RQ1, we use SySeVR to perform the vulnerability detection on the SARD and Big-Vul datasets, respectively. We use the detection results as a benchmark. The BERT-based models are then used to perform the detection on the same datasets and the detection results are compared with the benchmark. In detail, the BERT-based models used in RQ1 is the same as that described in section IV. All BERT-based models used in this paper are pretrained for *10 epochs* with *learning_rate=1e-04* and fine-tuned for *10 epochs* with *warmup_steps=1000, learning_rate=1e-05, weight_decay=0.1*. The loss functions used for both pre-training and fine-tuning phase are cross-entropy loss.

TABLE III. DETECTION RESULTS OF SARD (THE METRICS UNIT: %)

Model	Dataset	A	P	R	F1	MCC
BGRU(SySeVR)	SARD	95.16	**89.63**	86.36	87.97	84.96
BLSTM(SySeVR)	SARD	95.06	88.65	86.99	87.81	84.72
RoBERTa	SARD	**95.42**	85.25	**93.90**	**89.37**	**86.63**
DistilBERT	SARD	95.39	88.01	89.73	88.86	85.97
MobileBERT	SARD	95.02	84.56	92.56	88.38	85.36

Table III shows the detection results of 2 RNN-based models and 3 BERT-based models on the SARD dataset. The entire SARD dataset is divided into a training set and a test set after randomly sorted, with four-fifths of the training set and one-fifth of the test set. The RNN-based model is trained with the training set and tested with the test set. The BERT-based model is pre-trained with the entire SARD dataset, fine-tuned with the training set, inferred with the test set, and the inference results are compared with the benchmark.

In terms of F1, and the MCC metric, all the three BERT-based models outperform RNN-based models. Among the BERT-based models, RoBERTa has the best result. The corresponding F1 and the MCC metric is 93.90%, 89.37%, and 86.63%, respectively. All the BERT-based models outperform RNN-based models in accuracy metrics except MobileBERT. As for the precision metric, although the RNN-based model exceeds the BERT-based models, the recall rate of BERT-based models outperform RNN-based models. That is, the BERT-based model has a lower miss detection rate. Also, considering the metrics F1 and MCC, the overall detection effect of the BERT-based model is better than that of the RNN-based model.

TABLE IV. DETECTION RESULTS OF BIG-VUL-ALL (THE METRICS UNIT: %)

Model	Dataset	A	P	R	F1	MCC
BGRU(SySeVR)	Big-Vul-All	70.69	**13.09**	**14.30**	**13.67**	-3.94
BLSTM(SySeVR)	Big-Vul-All	72.41	12.90	12.18	12.53	-3.82
RoBERTa	Big-Vul-All	83.96	9.97	9.32	9.63	0.85
DistilBERT	Big-Vul-All	**84.23**	11.16	10.32	10.72	**2.09**
MobileBERT	Big-Vul-All	83.97	9.24	8.46	8.83	0.07

Table IV shows the detection results on the Big-Vul-All dataset. Compared to the detection results on the synthetic dataset SARD, the detection results show a huge drop. This means that real vulnerabilities have complex syntax and semantic infomations and it is impractical to experiment on Big-Vul-All, which covers multiple vulnerabilities. Therefore, we focus on one vulnerability type called memory corruption and perform experiments on the Big-Vul-MemCorr dataset.

TABLE V. DETECTION RESULTS OF BIG-VUL-MEMCORR (METRICS UNIT: %)

Model	Dataset	A	P	R	F1	MCC
BGRU(SySeVR)	Big-Vul-MemCorr	69.90	42.55	26.08	32.34	15.11
BLSTM(SySeVR)	Big-Vul-MemCorr	70.50	44.73	29.56	35.60	18.12
RoBERTa	Big-Vul-MemCorr	71.94	48.87	37.82	**42.64**	**24.82**
DistilBERT	Big-Vul-MemCorr	66.78	39.46	**38.26**	38.85	16.06
MobileBERT	Big-Vul-MemCorr	**72.18**	**49.13**	24.78	32.94	19.39

Table V shows the results of several models on Big-Vul-MemCorr. Because the vulnerability types in Big-Vul-MemCorr dataset all fall into one category, the detection results are much better than those for Big-Vul-All. Of these five models, the one works best is the RoBERTa model. Its accuracy reaches 71.94%, precision reaches 48.87%, recall reaches 37.82%, F1 reaches 42.64% and MCC reaches 24.82%. Such results also illustrate that, despite the complexity of the real vulnerability types, it is feasible to use the BERT-based model for detection if only one vulnerability type is targeted.

C. Experiments for RQ2

To answer RQ2, we use different combinations of three datasets (SARD, Big-Vul-All and Big-Vul-MemCorr) described in section IV in the pre-training phase and fine-tuning phase. Similarly, the BERT-based model is also shown in sectionIV. The combination of datasets has the following five combinations (COMBs).

- COMB1: Pre-training phase and fine-tuning phase using SARD dataset.

- COMB2: Pre-training phase using Big-Vul-MemCorr dataset and fine-tuning phase using Big-Vul-MemCorr dataset.
- COMB3: Pre-training phase using Big-Vul-All dataset and fine-tuning phase using Big-Vul-MemCorr dataset.
- COMB4: Pre-training phase using SARD dataset and fine-tuning phase using Big-Vul-MemCorr dataset.
- COMB5: Pre-training phase using Big-Vul-All dataset and fine-tuning phase using SARD dataset.

TABLE VI. DETECTION RESULTS OF COMBS (THE METRICS UNIT: %)

Max Length	COMBs	Metrics	RoBERTa	DistilBERT	MobileBERT
512	COMB1	A	**95.42**	95.39	95.02
		P	85.25	**88.01**	84.56
		R	**93.90**	89.73	82.56
		F1	**89.37**	88.86	88.38
		MCC	**86.63**	85.97	85.36
512	COMB2	A	71.94	66.78	**72.18**
		P	48.87	39.46	**49.13**
		R	37.82	**38.26**	24.78
		F1	**42.64**	38.85	32.94
		MCC	**24.82**	16.06	19.39
512	COMB3	A	**69.42**	67.38	69.18
		P	**43.16**	39.60	42.19
		R	34.34	**34.78**	31.73
		F1	**38.25**	37.03	36.22
		MCC	**18.49**	15.21	16.73
512	COMB4	A	**70.02**	64.14	62.82
		P	**42.64**	35.44	32.75
		R	25.21	**36.52**	33.04
		F1	31.69	**35.97**	32.90
		MCC	**14.88**	11.08	7.19
512	COMB5	A	**95.34**	95.09	95.16
		P	**94.11**	92.62	90.54
		R	82.41	82.59	**85.29**
		F1	**87.87**	87.32	87.84
		MCC	**85.29**	84.50	84.88

Table VI shows the results of using different combinations of datasets in the pre-training and fine-tuning phases. For the comparison of COMB1 and COMB5 or the comparison of COMB2 and COMB4, we can see that using the same dataset for the pre-training and fine-tuning phases is helpful for the vulnerability detection. All three BERT-based models have slightly higher F1 on COMB1 than on COMB5 and on COMB2 than on COMB4. In the comparison between COMB2 and COMB3, the two COMBs use different datasets in the pre-training phase. The COMB2 uses Big-Bul-MemCorr, while COMB3 uses Big-Vul-All. Both the RoBERTa and DistilBERT models have better F1 and MCC metrics on COMB2 than COMB3. This illustrates that the dataset used in the pre-training phase can affect the detection results. Big-Vul-All covers multiple vulnerability types, while Big-Vul-MemCorr covers only one vulnerability type. If the model learns too many vulnerability features in the pre-training phase it may affect the detection results of a single vulnerability.

D. Experiments for RQ3

To answer RQ3, we set the maximum sequence length for the BERT-based model to two lengths: 512 and 1024.

Table VII shows the results of the model for vulnerability detection using different maximum sequence lengths. For SARD, the total number of tokens in code slices less than or equal to 512 is 142,303 and less than or equal to 1024 is 144,526. For Big-Vul-MemCorr, the total number of tokens in code slices less than or equal to 512 is 2,430 and less than or equal to 1024 is 3,135. If the maximum sequence length is set to 512, the code slice will be truncated when the number of tokens in the code slice is greater than 512. As a result, these code slices lose some of their syntactic and semantic information. However, the results on COMB2 show that it is not the case that the larger the maximum sequence length, the better the detection. The model does not detect as well at a maximum sequence length of 1024 as it does at a maximum sequence length of 512. At a maximum sequence length of 1024, the F1 of the RoBERTa model on COMB2 is about 5 percent lower than at a maximum sequence length of 512.

TABLE VII. DETECTION RESULTS WITH DIFFENENT MAX LENGTHS OF IDS (THE METRICS UNIT: %)

Max Length	COMBs	Metrics	RoBERTa	DistilBERT	MobileBERT
512	COMB1	A	**95.42**	95.39	95.02
		P	85.25	**88.01**	84.56
		R	**93.90**	89.73	82.56
		F1	**89.37**	88.86	88.38
		MCC	**86.63**	85.97	85.36
1024	COMB1	A	95.40	**95.57**	95.25
		P	93.33	**95.34**	93.72
		R	**83.52**	82.41	82.35
		F1	88.16	**88.40**	87.67
		MCC	85.52	**86.04**	85.02
512	COMB2	A	71.94	66.78	**72.18**
		P	48.87	39.46	**49.13**
		R	37.82	**38.26**	24.78
		F1	**42.64**	38.85	32.94
		MCC	**24.82**	16.06	19.39
1024	COMB2	A	**69.30**	68.22	64.62
		P	**43.36**	40.93	39.05
		R	36.95	34.34	**50.43**
		F1	39.90	37.35	**44.02**
		MCC	**19.58**	16.39	19.10

VI. CONCLUSION

In this paper, we propose BBVD by studying BERT-based models for software vulnerability detection. Specifically, the proposed BBVD uses C/C++ code for pre-training, fine-tuning, and inferencing to detect vulnerability. Our results show that these BERT-based models outperform existing RNN-based models such as BiGRU and BiLSTM for vulnerability detection. There are still issues that can be investigated, such as detection results on real datasets are not as good as on synthetic datasets. Future work should use more real vulnerability datasets to improve the effectiveness of the BERT-based models in detecting vulnerability.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of P. R. China (Nos. 62002068).

REFERENCES

[1] Dowd M, McDonald J, Schuh J. The art of software security assessment: Identifying and preventing software vulnerability[M]. Pearson Education. 2006: 1-1129.

[2] Plate H, Ponta S E, Sabetta A. Impact assessment for vulnerability in open-source software libraries[C]. IEEE International Conference on Software Maintenance and Evolution. 2015: 411-420.

[3] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review[C]. International Conference on Software Engineering. 2013: 712-721.

[4] Feng Z, Guo D, Tang D, et al. CodeBERT: A pre-trained model for programming and natural languages[C]. Findings of the Association for Computational Linguistics. 2020: 1536-1547.

[5] Li Z, Zou D, Xu S, et al. VulDeePecker: A deep learning-based system for vulnerability detection[J]. Network and Distributed System Security Symposium. 2018: 1-15.

[6] Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. In Proceedings of NAACL-HLT. 2019: 4171-4186.

[7] Liu Y, Ott M, Goyal N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.

[8] Sun Z, Yu H, Song X, et al. Mobilebert: A compact task-agnostic bert for resource-limited devices[J]. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020: 2158-2170.

[9] Sanh V, Debut L, Chaumond J, et al. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter[J]. arXiv preprint arXiv:1910.01108, 2019.

[10] Accessed Oct 15, 2022. Software assurance reference dataset (SARD). https://samate.nist.gov/SARD/

[11] Fan J, Li Y, Wang S, et al. A C/C++ code vulnerability dataset with code changes and CVE summaries[C]. International Conference on Mining Software Repositories. 2020: 508-512.

[12] Li Z, Zou D, Xu S, et al. SySeVR: A framework for using deep learning to detect software vulnerability[J]. IEEE Transactions on Dependable and Secure Computing. 2021: 1-15.

[13] Lin S, Xiao G, Yan Y, Suter D, Wang H. Hypergraph optimization for multi-structural geometric model fitting[C]. In Proceedings of the AAAI Conference on Artificial Intelligence. 2019, 33(1): 8730-8737.

[14] Lin S, Luo H, Yan Y, Xiao G, Wang H. Co-clustering on Bipartite Graphs for Robust Model Fitting[J]. IEEE Transactions on Image Processing. 2022, 31: 6605-6620.

[15] Lin S, Wang X, Xiao G, Yan Y, Wang H. Hierarchical representation via message propagation for robust model fitting[J]. IEEE Transactions on Industrial Electronics. 2020, 68(9): 8582-8592.

[16] Yang H, Lin S, Cheng L, Lu Y, Wang H. SCINet: Semantic Cue Infusion Network for Lane Detection. IEEE International Conference on Image Processing. 2022, 1811-1815.

[17] Huang F, Zhang X, Zhao Z, et al. Bi-directional spatial-semantic attention networks for image-text matching[J]. IEEE Transactions on Image Processing. 2018, 28(4): 2008-2020.

[18] Xu J, Li Z, Huang F, et al. Visual sentiment analysis with social relations-guided multiattention networks[J]. IEEE Transactions on Cybernetics. 2020, 52(6): 1-13.

[19] Liu Y, Zhang X, Huang F, et al. Cross-Attentional Spatio-Temporal Semantic Graph Networks for Video Question Answering[J]. IEEE Transactions on Image Processing. 2022, 31: 1684-1696.

[20] Huang F, Jolfaei A, Bashir A K. Robust multimodal representation learning with evolutionary adversarial attention networks[J]. IEEE Transactions on Evolutionary Computation. 2021, 25(5): 856-868.

[21] Huang F, Xu J, Weng J. Multi-task travel route planning with a flexible deep learning framework[J]. IEEE Transactions on Intelligent Transportation Systems. 2020, 22(7): 3907-3918.

[22] Yang W, Xie Y, Lin A, et al. End-to-end open-domain question answering with BERTserini[C]. Conference of the North American Chapter of the Association for Computational Linguistics. 2019: 72-77.

[23] Li C, Pang B, Liu Y, et al. Adsgnn: Behavior-graph augmented relevance modeling in sponsored search[C]. Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021: 223-232.

[24] Pang B, Li C, Liu Y, et al. Improving Relevance Modeling via Heterogeneous Behavior Graph Learning in Bing Ads[C]. Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 2022: 3713-3721.

[25] Bi S, Li C, Han X, et al. Leveraging Bidding Graphs for Advertiser-Aware Relevance Modeling in Sponsored Search[C]. Findings of the Association for Computational Linguistics: EMNLP. 2021: 2215-2224.

[26] Qu C, Yang L, Qiu M, et al. BERT with history answer embedding for conversational question answering[C]. International ACM SIGIR conference on research and development in information retrieval. 2019: 1133-1136.

[27] Zhan J, Mao J, Liu Y, et al. An analysis of BERT in document ranking[C]. International ACM SIGIR Conference on Research and Development in Information Retrieval. 2020: 1941-1944.

[28] Grail Q, Perez J, Gaussier E. Globalizing BERT-based transformer architectures for long document summarization[C]. Conference of the European chapter of the association for computational linguistics: Main volume. 2021: 1792-1810.

[29] Munikar M, Shakya S, Shrestha A. Fine-grained sentiment classification using BERT[C]. Artificial Intelligence for Transforming Business and Society. 2019, 1: 1-5.

[30] Sun C, Qiu X, Xu Y, et al. How to fine-tune bert for text classification?[C]. China National Conference on Chinese Computational Linguistics. 2019: 194-206.

[31] Kanade A, Maniatis P, Balakrishnan G, et al. Learning and evaluating contextual embedding of source code[C]. International Conference on Machine Learning. 2020: 5110-5121.

[32] Guo D, Ren S, Lu S, et al. Graphcodebert: Pre-training code representations with data flow[J]. arXiv preprint arXiv:2009.08366, 2020.

[33] Ziems N, Wu S. Security Vulnerability Detection Using Deep Learning Natural Language Processing[C]. Conference on Computer Communications Workshops. 2021: 1-6.

[34] Baker B S. A program for identifying duplicated code[J]. Computing Science and Statistics. 1993: 49-49.

[35] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code[C]. International Conference on Software Maintenance. 1999: 109-118.

[36] Gitchell D, Tran N. Sim: A utility for detecting similarity in computer programs[J]. ACM Sigcse Bulletin. 1999, 31(1): 266-270.

[37] Allen F E. Control flow analysis[J]. ACM Sigplan Notices. 1970, 5(7): 1-19.

[38] VenkataKeerthy S, Aggarwal R, Jain S, et al. Ir2vec: Llvm ir based scalable program embeddings[J]. ACM Transactions on Architecture and Code Optimization. 2020, 17(4): 1-27.

[39] Kim Y C, Cho Y Y, Moon J B. A plagiarism detection system using a syntax-tree[C]. International Conference on Computational Intelligence. 2004, 1: 23-26.

[40] Jiang L, Misherghi G, Su Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C]. International Conference on Software Engineering. 2007: 96-105.

[41] Baxter I D, Yahin A, Moura L, et al. Clone detection using abstract syntax trees[C]. International Conference on Software Maintenance. 1998: 368-377.

[42] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems. 1987, 9(3): 319-349.

[43] Liu C, Chen C, Han J, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C]. International Conference on Knowledge Discovery and Data Mining. 2006: 872-881.

[44] Pham N H, Nguyen H A, Nguyen T T, et al. Complete and accurate clone detection in graph-based models[C]. International Conference on Software Engineering. 2009: 276-286.

[45] Sheneamer A, Roy S, Kalita J. A detection framework for semantic code clones and obfuscated code[J]. Expert Systems with Applications. 2018, 97: 405-420.

[46] Kim Y, Kim M, Kim Y J, et al. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE[C]. International Conference on Software Engineering. 2012: 1143-1152.

[47] Yamamoto Y, Miyamoto D, Nakayama M. Text-mining approach for estimating vulnerability score[C]. International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security. 2015: 67-73.

[48] Accessed Oct 15, 2022. National Vulnerability Database. https://doi.org/10.18434/M3436

[49] Toloudis D, Spanos G, Angelis L. Associating the Severity of vulnerability with their Description[C]. International Conference on Advanced Information Systems Engineering. 2016: 231-242.

[50] Spanos G, Angelis L, Toloudis D. Assessment of vulnerability severity using text mining[C]. Proceedings of the Pan-Hellenic Conference on Informatics. 2017: 1-6.

[51] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in Neural Information Processing Systems. 2019, 32: 1-11.

[52] Chakraborty S, Krishna R, Ding Y, et al. Deep learning based vulnerability detection: Are we there yet[J]. IEEE Transactions on Software Engineering. 2021, 48(9): 3280-3296.

[53] Wolf T, Debut L, Sanh V, et al. Transformers: State-of-the-art natural language processing[C]. Conference on Empirical Methods in Natural Language Processing: System Demonstrations. 2020: 38-45.