VulDeBERT: A Vulnerability Detection System Using BERT

Soolin Kim*†, Jusop Choi*, Muhammad Ejaz Ahmed†, Surya Nepal† and Hyoungshick Kim*†
* *Department of Electrical and Computer Engineering, Sungkyunkwan University, Republic of Korea*
{soolinkim, cjs1992, hyoung}@skku.edu
† *Data61, CSIRO, Australia*
{ejaz.ahmed, surya.nepal}@data61.csiro.au.

***Abstract*—Deep learning technologies recently received much attention to detect vulnerable code patterns accurately. This paper proposes a new deep learning-based vulnerability detection tool dubbed VulDeBERT by fine-tuning a pre-trained language model, Bidirectional Encoder Representations from Transformers (BERT), on the vulnerable code dataset. To support VulDeBERT, we develop a new code analysis tool to extract well-represented abstract code fragments from C and C++ source code. The experimental results show that VulDeBERT outperforms the state-of-the-art tool, VulDeePecker [1] for two security vulnerability types (CWE-119 and CWE-399). For the CWE-119 dataset, VulDeBERT achieved an F1 score of 94.6%, which is significantly better than VulDeePecker, achieving an F1 score of 86.6% in the same settings. Again, for the CWE-399 dataset, VulDeBERT achieved an F1 score of 97.9%, which is also better than VulDeePecker, achieving an F1 score of 95% in the same settings.**

***Index Terms*—Vulnerability Detection, Code Gadget**

I. Introduction

Detecting security vulnerabilities is a well-known fundamental problem in software security because they can potentially be abused for security attacks. Therefore, many static analysis techniques have been proposed to identify vulnerable parts in source code [2], [3], [4], [5]. However, these techniques mainly rely on a database of known vulnerable code patterns or rules, requiring significant human expertise to find effective code patterns and rules. Consequently, they are ineffective in detecting new vulnerable code patterns when they have a slight change in some expressions.

Recently, machine learning-based techniques [1], [6] have been proposed to overcome these limitations. For example, Li et al. [1] introduced a deep learning-based vulnerability detection system dubbed VulDeePecker and showed that VulDeePecker could achieve fewer false negatives (with reasonable false positives) than other approaches.

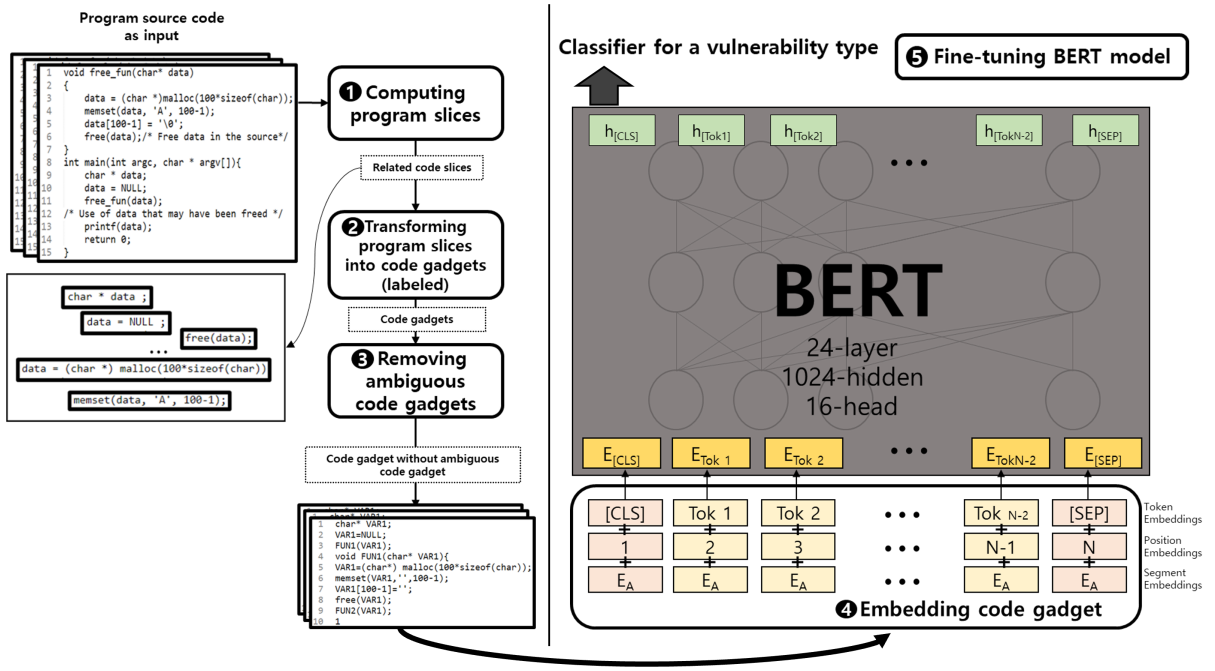
This paper presents a more effective and accurate deep learning-based vulnerability detection system dubbed VulDeBERT using Bidirectional Encoder Representations from Transformers (BERT) [7], which is known as a model for natural language processing. Because the program source code is composed of meaningful sequences of multiple instructions and contains highly repeated instruction sequences, BERT has been applied to various tasks in analyzing program code and achieved high performance [8]. Thus, we use BERT to develop VulDeBERT for processing program source code as inputs and detecting vulnerable code fragments.

BERT was originally pre-trained with unlabeled data extracted from BooksCorpus and English Wikipedia for natural language processing. Therefore, to adapt BERT to the vulnerable code detection task in C and C++ source code, we found that it is important to conduct a fine-tuning process with code gadgets. Each code gadget represents a sequence of multiple (abstract) program statements that are semantically related to each other in terms of data and control dependencies. Generating well-structured code gadgets from source code is essential for VulDeBERT. Thus, we develop a new code gadget generation method that can effectively be used for VulDeBERT and the other deep learning-based vulnerable detection models. Our code gadget generation method can properly handle the code containing nested function calls that cannot be processed by VulDeePecker [1]. Our contributions are summarized below:

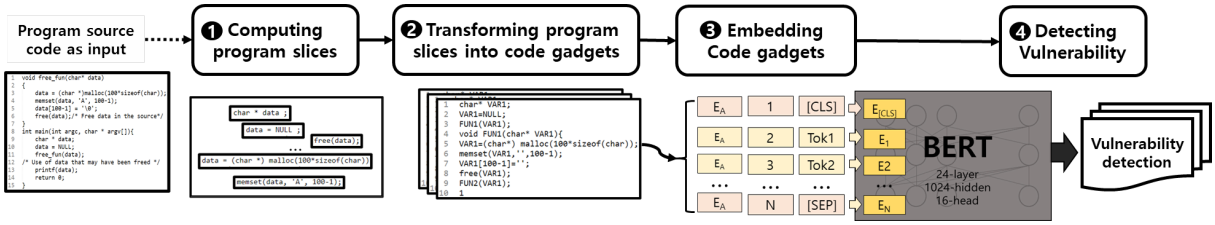
- We propose a vulnerability detection system called VulDeBERT using BERT model in C and C++ source code. We will make our code available at https://github.com/SKKU-SecLab/VulDeBERT.git (see Section II).
- We develop a new code gadget generation tool that can be used for static analysis in C and C++ (see Section III).
- Our experimental results show that VulDeBERT outperforms the state-of-the-art method [1] for two security vulnerability types (CWE-119 and CWE-399) on the SARD [9] and NVD [10] datasets (see Section IV).

II. Overview of VulDeBERT

In this section, we describe the overview of VulDeBERT, which is designed as a classification model to detect security vulnerabilities in C and C++ source code. VulDeBERT has the training and detection phases, as shown in Figure 1. In the training phase, VulDeBERT takes program source codes as input, generates labeled code gadgets representing safe



(a) Training phase



(b) Detection phase

Fig. 1: Overview of VulDeBERT framework.

and vulnerable abstract program code fragments, and fine-tunes a pre-trained BERT model with those code gadgets (see Section II-A). In the detection phase, VulDeBERT takes a target source code fragment as input, generates its code gadget, and checks whether it is vulnerable or not with the fine-tune BERT model (see Section II-B).

A. Training phase

The training phase aims to train a pre-trained BERT model for detecting a vulnerability by receiving ground-truth data. This phase comprises five stages, as shown in Figure 1a.

Given a set of program source codes, VulDeBERT computes program slices related to system function calls (❶). Next, VulDeBERT transforms program slices into (labeled) code gadgets by abstracting program slices (e.g., replacing variable and function names with symbolic representations) (❷). Section III provides a more detailed explanation of generating code gadgets. The code abstraction process can occasionally generate *ambiguous* code gadgets as a side effect because the identical code gadgets can be transformed from both safe and vulnerable program slices simultaneously. Therefore, we remove ambiguous code gadgets to avoid training with such code gadgets (❸). Finally, we should embed the generated code gadgets for BERT input representation to train a BERT model. VulDeBERT appends the special [CLS] token to indicate the start of the input vector and the special [SEP] token to indicate the end of the input vector (❹). After encoding code gadgets, we input them into the BERT model and fine-tune all model parameters in the BERT model to detect some target vulnerabilities. The output vector is fed to a binary classifier to detect a specific vulnerability type (❺).

B. Detection phase

In the detection phase, VulDeBERT uses the fine-tuned BERT model to detect the vectors corresponding to the vulnerable code gadgets generated from a target source code.

Given a target source code, VulDeBERT performs the same steps except for the step of removing ambiguous code gadgets (❸ in Figure 1a) to obtain the vector corresponding to the target source code's code gadgets, as shown in Figure 1b. When a vector is classified as a vulnerable one, the vulnerability's

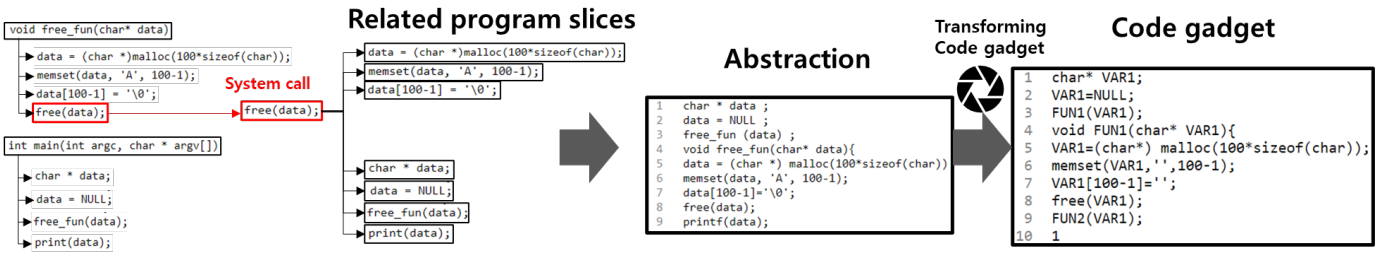


Fig. 2: Example of code gadget generation. We collect the code gadget when it is triggered by the call of system function in function *free_fun*. If we adjust the depth of the callee/caller functions to the unlimited, the code gadget is composed of all functions.

location is reported. Otherwise, the vector is classified as a safe one.

III. Code Gadget Generation

To use a deep learning model with program source codes, we need to generate input vectors representing program source codes, which can be fed into the deep learning model. To achieve this goal, we use an abstract representation of the program source code fragment, dubbed *code gadget* representing a sequence of multiple (abstract) program statements that are semantically related to each other in terms of data and control dependencies. The previous work [1] introduced the idea of using code gadgets for vulnerability detection. This paper also uses this idea with a new code gadget generation method.

We are particularly interested in generating code gadgets from code slices related to the parameters of system function calls because our main research interests are to find security vulnerabilities related to system function calls. Therefore, a code gadget is created by collecting the code slices related to a system function call from its caller and callee parts. The code gadget generation of VulDeBERT is composed of three stages: computing program slices related to system function calls (see Section III-A), transforming program slices into code gadgets (see Section III-B) and, removing ambiguous code gadgets (see Section III-C).

A. Computing program slices

In VulDeBERT, the first stage of the code gadget generation computes the program slices related to system function calls extracted from a given source code. Figure 2 shows an example of the code gadget generation. The first step to computing program slices is removing non-ASCII characters and comments. Then VulDeBERT starts collecting program slices related to a system function call, *free(data)*. We perform backward program slicing from this function call to discover the statements contributing to the value of the *data* variable. The code lines which are related to the *data* variable are in the *free_fun* function and also in the *main* function. We extracted those code slices. After repeating such backward program slicing processes, we can collect the program slices related to *free(data)*. We combine those slices into one program code according to the function call order.

VulDeePecker [1] used a similar way to compute program slices related to a system function call. However, we found that the VulDeePecker's implementation cannot properly handle *nested function calls*. To address this problem, we develop a new code gadget generation tool to handle k level of nested function calls from the main function by tracing nested function calls iteratively.

B. Transforming code gadgets

In this stage, VulDeBERT transforms collected program slices into code gadgets. We need to transform concrete program code in program slices into generalized and abstract symbolic representations for being fed into the deep learning model. For program analysis, user-specific functions and variables, comments, strings, and constant values may not be important. Therefore, we aim to capture only core semantic information needed to analyze the structure of vulnerable code. The first step to transforming program slices into code gadgets is replacing user-defined functions and variables with abstract symbolic representations. For example, as shown in Figure 2, the user-defined function ("free_fun") or variable name ("data") are replaced with abstract symbols ("FUN1" and "VAR1") representing a function and a variable, respectively. We only consider the structure of statements and their relationship. To achieve this goal, we employ the following rules.

- The ith ordered variable name is replaced with "VARi."
- The ith ordered function name is replaced with "FUNi." However, we preserve the function names for system function calls (e.g., `malloc`, `fread`, and `memset`).

Next, we label the generated code gadgets into vulnerable or patched ones with the corresponding program source code information, which can be collected with source code from NVD and SARD. Sometimes, vulnerable and patched codes can generate the same code gadgets. Therefore, in VulDeBERT, we exclude such code gadgets as ambiguous code gadgets. Finally, all generated code gadgets are classified into vulnerable or patched ones.

Interestingly, we found that the VulDeePecker code gadget dataset contains several mislabeled code gadgets, which is consistent with the observation in the previous study [11].

Therefore, we develop our own gadget method to avoid such cases.

C. Removing ambiguous code gadgets

We manually examined the generated code gadgets and found that code abstraction (i.e., replacing variable and function names with symbolic representations) can occasionally cause false alarms as a side effect specifically for short source code. A few lines of abstracted code would be too ambiguous to determine whether it matches a piece of vulnerable or patched code. We need to remove such code gadgets from the dataset to avoid false alarms.

IV. Evaluation

In this section, we conduct experiments to show the feasibility of VulDeBERT and evaluate its detection accuracy compared to the state-of-the-art solution, VulDeePecker [1]. We first explain the implementation details and the dataset used in experiments and present the evaluation results to discuss the effectiveness of VulDeBERT.

A. Experimental settings

Setup. Our experiment settings are as follows: we use Intel(R) Xeon(R) 2.10 GHz CPU with 256.0 GB RAM and NVIDIA GeForce Titan for building the machine learning models used in experiments. We used Pytorch to implement VulDeBERT. To find the optimized VulDeBERT, we trained VulDeBERT with a learning rate of 0.00001 and three epochs. We set the random seed to 2022 and batch size to 24. For VulDeBERT, we use the BERT-base model with 24 layers of transformer blocks, 1024 hidden layers, and 16 self-attention heads. Furthermore, we use the Adam optimizer and cross-entropy for the classification loss. To show the superiority of VulDeBERT, we used VulDeePecker [1] as a baseline model. However, the source code of VulDeePecker is not publicly available. Therefore, we implemented the bidirectional LSTM (BiLSTM) classification model used for VulDeePecker [1] with the same model architecture and parameters described in [1]. We used Keras [12] to implement BiLSTM. The model's hyperparameters are exactly the same as those in VulDeePecker [1]. The model was trained with a learning rate of 0.00001 and four epochs; the number of tokens in the vector representation of code gadgets was set to 50; the dropout was set to 0.5; the batch size was set to 64; the number of epochs was set to 4; the mini-batch gradient descent together with ADAMAX [13] was used for training; 300 hidden nodes were chosen.

TABLE I: Code gadgets description.

Dataset		# of code gadgets	# of vulnerable code gadgets	# of patched code gadgets
CWE-119	VulDeePecker code gadgets	39,753	10,440	29,313
	Our code gadgets	26,702	14,206	12,496
CWE-399	VulDeePecker code gadgets	21,885	7,285	14,600
	Our code gadgets	14,807	11,555	3,252

Dataset. We generated our code gadgets from two program source codes maintained by the National Institute of Standards and Technology (NIST): the NVD [10] and the Software Assurance Reference Dataset (SARD) project [9]. We focus on two types of CWE (i.e., buffer error vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399)) to evaluate the performance of VulDeBERT. For CWE-119, our code gadget generation method generated 14,206 vulnerable code gadgets and 12,496 patched code gadgets, respectively, while 10,440 vulnerable and 29,313 patched code gadgets exist respectively at the VulDeePecker code gadget dataset (https://github.com/CGCL-codes/VulDeePecker). For CWE-399, our code gadget generation method generated 11,555 vulnerable code gadgets and 3,252 patched code gadgets, respectively, while 7,285 vulnerable and 14,600 patched code gadgets exist, respectively, in the VulDeePecker code gadget dataset. Table I summarizes the dataset used for experiments.

For CWE-119, we initially obtained 34,805 vulnerable and 24,457 patched code gadgets, including ambiguous ones. However, when we remove ambiguous code gadgets described in Section III-C, we finally have 14,206 vulnerable and 12,496 patched code gadgets, respectively. In contrast, for CWE-399, the number of ambiguous code gadgets is relatively smaller– we initially obtained 16,625 vulnerable and 3,368 patched code gadgets, and finally have 11,555 vulnerable and 3,252 patched code gadgets after removing ambiguous ones.

Training and test code gadgets. We randomly selected 80% of the code gadgets as training code gadgets and used the remaining 20% as the test code gadgets. We used this setting for all experiments.

Evaluation metrics. For evaluating our VulDeBERT detection systems, we use the following metrics: false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), precision (P), and F1 score (F1) [14]. Let TP is the number of code gadgets with vulnerabilities detected correctly, FP is the number of code gadgets with false vulnerabilities detected, FN is the number of code gadgets with real vulnerabilities that are not detected, and TN is the number of code gadgets with no undetected vulnerabilities. The false positive rate metric is $\texttt{FPR} = \frac{\texttt{FP}}{\texttt{FP+TN}}$ which means false positive vulnerabilities to the whole code gadgets that are not vulnerable. The false negative rate metric $\texttt{FNR} = \frac{\texttt{FN}}{\texttt{TP+FN}}$ measures the ratio of false negative vulnerabilities to the whole code gadgets that are vulnerable. The true positive rate metric is $\texttt{TPR} = \frac{\texttt{TP}}{\texttt{TP+FN}}$ which means the ratio of true positive vulnerabilities to the whole code gadgets are vulnerable. The precision metric is $\texttt{P} = \frac{\texttt{TP}}{\texttt{TP+FP}}$ measures the correctness of the detected vulnerabilities. The F1 score metric is $\texttt{F1} = \frac{2\times\texttt{P}\times\texttt{TPR}}{\texttt{P+TPR}}$ which considers both accuracy and false negative rate in detecting vulnerabilities.

B. Results

Effectiveness of code gadget generation method. To demonstrate the effectiveness of our code gadget generation method (see Section II-A), we evaluate the performance of models with our own code gadgets and VulDeePecker code gadgets [1], respectively. Experiments were repeated ten times at each configuration. Table II and III summarize the evaluation

TABLE II: Results of detecting CWE-119 (μ: Mean, σ: Standard deviation).

CWE-119		FPR		FNR		TPR		P		F1 score	
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
BERT	Our code gadgets	2.1	0.4	**7.8**	1.3	**92.2**	1.3	**97.1**	0.7	**94.6**	0.9
	VulDeepecker code gadgets	**1.3**	0.1	10.9	0.6	89.1	0.6	96.0	0.2	92.4	0.3
BiLSTM	Our code gadgets	4.8	0.5	15.6	0.7	84.4	0.7	93.1	0.6	88.5	0.4
	VulDeepecker code gadgets	2.5	0.5	29.0	1.3	70.9	1.3	91.2	0.7	79.8	0.4
VulDeepecker results [1]		2.9		18.0		82.0		91.7		86.6	

TABLE III: Results of detecting CWE-399 (μ: Mean, σ: Standard deviation).

CWE-399		FPR		FNR		TPR		P		F1 score	
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
BERT	Our code gadgets	**0.3**	0.2	**4.0**	0.3	**96.0**	0.3	**99.9**	0.1	**97.9**	0.1
	VulDeepecker code gadgets	1.0	0.2	7.1	0.5	92.9	0.5	97.8	0.4	95.3	0.4
BiLSTM	Our code gadgets	3.2	0.3	10.8	0.8	89.2	0.8	98.9	0.1	93.8	0.5
	VulDeepecker code gadgets	2.4	3.0	25.1	6.2	74.9	6.2	95.6	2.1	83.8	2.6
VulDeepecker results [1]		2.8		4.7		95.3		94.6		95.0	

results (i.e., the mean and standard deviation of each evaluation metric).

Table II shows that deep learning models (BERT and BiLSTM) with our code gadgets overall achieve higher accuracy than those with VulDeePecker code gadgets for CWE-119. VulDeBERT with our code gadgets produced the best accuracy results, achieving the mean F1 score of 94.6% (standard deviation of 0.9), which is significantly better than VulDeePecker [1], achieving an F1 score of 86.6%. Although our own BiLSTM implementation's detection accuracy (79.8%) is not superior to VulDeePecker's detection accuracy (86.6%) reported in [1] when VulDeePecker's code gadgets are used, our BiLSTM implementation produced better detection accuracy (88.5%) than VulDeePecker's accuracy reported in [1] when our code gadgets are used. Table III shows the experimental results for CWE-399. Again, BERT and BiLSTM work better with our code gadgets than the VulDeePecker code gadgets. The best model configuration is BERT with our code gadgets, which achieves them mean F1 score of 97.9% (standard deviation of 0.1).

Effectiveness of deep learning models. Table II and III also show that BERT works better than BiLSTM for detecting vulnerable codes. For CWE-119, BERT always outperforms BiLSTM when the same code gadget dataset is used for both models. Similarly, for CWE-399, BERT always outperforms BiLSTM when the same code gadget dataset is used for both models. Based on those evaluation results, our recommended configuration is to use BERT with our code gadgets for both vulnerability types (CWE-119 and CWE-399).

V. Limitations

Programming language specific analysis. In theory, VulDeBERT can be implemented for any programming language. In practice, however, it is challenging to provide a code gadget generation tool suitable for a target programming language. Therefore, our current VulDeBERT implementation supports only program source code written in C and C++.

Lack of supporting diverse vulnerability types. Our current VulDeBERT implementation can discover the only vulnerabilities related to system function calls. Therefore, we must explore new static analysis techniques to compute program slices related to other vulnerability types (e.g., cryptographic misuses). In addition, BERT is also needed to fine-tune with new code gadgets related to such vulnerabilities.

Focusing on fine-tuning. In the current VulDeBERT implementation, we used a conventional BERT model pre-trained for natural language processing. To propose a more suitable model for vulnerability detection, we need to consider another pre-trained model, such as CodeBERT [8], which was trained with a program source code dataset.

VI. Related work

Our discussion of related work is grouped as follows.

Conventional vulnerability detection methods. Conventional vulnerability detection methods use a set of rules or a database of known vulnerable code patterns. Rule-based methods require manual effort to generate effective rules, and their performance can be varied with security experts' knowledge. For example, Flawfinder [15], RATS [16], and Checkmarx [17] are well-known tools, but they suffer from high false positives and false negatives [6]. Other approachs [2], [3], [4], [5] are to develop a static analysis tool based on a database of known vulnerable code patterns. This approach is quite useful in detecting known vulnerable code patterns but is not robust with new vulnerable code patterns with a slight change.

Machine learning-based vulnerability detection methods. Using machine learning is not new for developing static analysis tools. However, recent deep learning advancements have inspired the development of deep learning-based vulnerability detection tools. In general, this approach builds a classification model with vulnerable and safe codes and uses the model to detect (unlabeled) vulnerable code fragments. Li et al. [1] introduced the first deep learning-based vulnerability detection model dubbed VulDeePecker using a BiLSTM model feeding one LSTM network [18] with the program statements in the forward direction and another in the backward direction. To train program source codes effectively, they also introduced the concept of code gadgets to transform actual program codes into more generalized and normalized code fragments. The experimental results show that VulDeePecker outper-

forms the other detection methods such as VUDDY [5] and VulPecker [4] in detection accuracy. In addition, VulDeepecker was used to detect four vulnerabilities that are not reported in public vulnerability databases. Li et al. [6] extended VulDeePecker into a new tool dubbed SySeVR with a program dependency graph to capture the semantic meaning of program code more effectively. They built a BiGRU-based model [18], [19], [20] as SySeVR because their experimental results showed BiGRU outperformed other model architectures, including BiLSTM. Jeon et al. [21] proposed a deep learning-based vulnerability detection tool using BERT dubbed SmartConDetect for detecting security vulnerabilities in smart contracts on Ethereum. The experimental results showed that BERT could effectively be used to detect software vulnerabilities in program source code. We propose a novel deep learning-based vulnerability detection tool dubbed VulDeBERT for software vulnerabilities in C and C++ program codes. Our experimental results demonstrate that VulDeBERT produced better detection accuracy than VulDeePecker [1] for two security vulnerability types (CWE-119 and CWE-399).

BERT. Bidirectional Encoder Representations from Transformers (BERT) [7] has been proposed for pre-training deep *bidirectional representations* from unlabeled texts by jointly considering the forward and backward directions of contextual sentences. As a result, BERT was successfully adapted as a pre-trained representation for various applications (e.g., a question-answering task [22]). Furthermore, BERT is also used as a pre-trained model for programming analysis. For example, the CuBERT [23] was built with two pre-trained tasks such as predicting masked tokens and checking whether two logical lines of code are related to each other in a contextual sentence. Feng et al. [8] proposed the CodeBERT as a bimodal pre-trained model for natural language and programming language. The fine-tuned CodeBERT performed well for natural language code search and code-to-documentation generation.

VII. Conclusion

In this paper, we propose VulDeBERT as a novel vulnerability detection model for C and C++ source code. The experimental results demonstrated that VulDeBERT outperforms VulDeePecker [1] in detecting two well-known security vulnerability types (CWE-119 and CWE-399). For the CWE-119 dataset, VulDeBERT achieved an F1 score of 94.6%, which is significantly better than VulDeePecker, achieving an F1 score of 86.6%. For the CWE-399 dataset, VulDeBERT achieved an F1 score of 97.9%, which is also better than VulDeePecker, achieving an F1 score of 95%. VulDeBERT can be extended to detecting security vulnerabilities in other programming languages. Therefore, as a part of future work, we plan to extend VulDeBERT into a more generalized one with pre-training tasks for various programming language analyses. Furthermore, we plan to analyze not only the security vulnerabilities related to system function calls but also other types of security vulnerabilities.

Acknowledgment

The authors would thank anonymous reviewers. Hyoungshick Kim is the corresponding author. This work was supported by the Korean government's projects (No.2018-0-00532, No.2022-0-00995) and was carried out while the first and last authors worked at CSIRO Data61.

References

[1] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in Proceedings of 25th Annual Network and Distributed System Security Symposium (NDSS), 2018.

[2] J. Jang, A. Agrawal, and D. Brumley, "Finding unpatched code clones in entire os distributions," in Proceedings of IEEE Symposium on Security and Privacy, 2012.

[3] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 1157–1168.

[4] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis," in Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016.

[5] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in Proceedings of 38th IEEE Symposium on Security and Privacy (SP), 2017.

[6] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," IEEE Transactions on Dependable and Secure Computing, 2021.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," CoRR, vol. abs/1810.04805, 2018.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2020.

[9] "Software Assurance Reference Dataset," https://samate.nist.gov/SRD/index.php.

[10] "National Vulnerability Database," https://nvd.nist.gov/.

[11] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and Don'ts of Machine Learning in Computer Security," in Proceedings of the USENIX Security Symposium, 2022.

[12] "Keras," https://github.com/keras-team/keras/.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[14] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A Survey on Systems Security Metrics," ACM Computing Surveys (CSUR), vol. 49, no. 4, pp. 1–35, 2016.

[15] "FlawFinder," http://www.dwheeler.com/flawfinder/.

[16] "Rough Audit Tool for Security," https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[17] "Checkmarx," https://checkmarx.com/.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," arXiv preprint arXiv:1409.1259, 2014.

[20] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," Neural networks, vol. 18, no. 5-6, pp. 602–610, 2005.

[21] S. Jeon, G. Lee, H. Kim, and S. S. Woo, "SmartConDetect: Highly Accurate Smart Contract Code Vulnerability Detection Mechanism using BERT," in Proceedings of the KDD Workshop on Programming Language Processing, Virtual Conference, 2021.

[22] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," arXiv preprint arXiv:1606.05250, 2016.

[23] 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, 2020.