Python source code vulnerability detection based on CodeBERT language model

Kunpeng Zhao*
School of Cyberspace Security
Zhengzhou University
Zhengzhou, China
Email: zkunp@gs.zzu.edu.cn

Shuya Duan†
School of Cyberspace Security
Zhengzhou University
Zhengzhou, China
Email: duanshuyaa@gs.zzu.edu.cn

Ge Qiu†
School of Cyberspace Security
University of Information Engineering
Zhengzhou, China
Email: 3111896391@qq.com

Jinyuan Zhai†
School of Cyberspace Security
University of Information Engineering
Zhengzhou, China
Email: 1243251137@qq.com

Mingze Li†
School of Cyberspace Security
University of Information Engineering
Zhengzhou, China
Email: 311897521@qq.com

Long Liu†*
School of Cyberspace Security
University of Information Engineering
Zhengzhou, China
Email: d12_liu@163.com

2024 7th International Conference on Algorithms, Computing and Artificial Intelligence (ACAI) | 979-8-3315-2931-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ACAI63924.2024.10899694

***Abstract*—Programming language source code vulnerability mining is crucial to improving the security of software systems, but current research is mostly focused on the C language field, with little attention paid to scripting languages. Python code is currently widely used in software systems, and vulnerability detection in Python code is also very important. As an interpreted language, Python has a concise and clear grammatical structure and is closer to natural language, so Python source code vulnerability mining has greater advantages. Previous detection systems based on LSTM models have achieved certain results in detecting Python source code vulnerabilities, but due to the limitations of the model, the vulnerability dataset has low accuracy, lack of full consideration of the deep semantic features of the source code, and the vulnerability multi-classification process is relatively cumbersome and inefficient, and there is room for improvement in accuracy and recall. Therefore, this paper proposes a source code snippet multi-classification vulnerability detection system based on CodeBERT, which aims to automatically detect the security of software code snippets. First, by analyzing the differences between the new and old versions, a multi-label vulnerability dataset is constructed using diff files, and then the RoBERTa is used to obtain high-quality expressions of code keywords. Finally, the constructed multi-classification CodeBERT model is introduced for fine-tuning detection. By improving the accuracy of vulnerability datasets and introducing a high-performance model, this method has achieved significant improvements in source code vulnerability mining and vulnerability classification compared to traditional LSTM model-based detection methods. The average accuracy of the proposed model reaches 98%.**

***Keywords*-Diff; CodeBERT; Python source code; vulnerability mining**

I. INTRODUCTION

The rapid development of the Internet has profoundly changed people's production and lifestyle, bringing unprecedented convenience to individuals and society. However, with the significant increase in the number of software vulnerabilities and potential risks, the inefficiency of manual vulnerability detection methods can no longer meet the current urgent needs for software security. As an effective strategy to deal with network attacks, source code vulnerability detection has gradually occupied an important position in the field of network security research and has become the focus of academic circles.

There are many tools in the field of vulnerability mining, such as Scan Dal[1], HybriDroid[2] and PREfix[3]. These tools are mainly used to detect vulnerabilities in C language or Java. There is relatively little research on vulnerability mining in Python. There are mainly the following methods. Michelsen proposed a Python vulnerability detection tool based on static taint analysis - PyT[4]; Peng et al[5]. proposed a Python source code vulnerability detection method based on code similarity; Wartschinski et al[6]. proposed a deep learning based on LSTM Model method for detecting vulnerabilities in Python source code; Guo et al[7]. innovatively proposed a method for detecting and classifying vulnerabilities in Python source code based on BiLSTM and OVO SVMs models.

They do not fully consider the contextual semantic information of vulnerability datasets, and the models do not extract deep semantic features for the source code. In addition, in terms of vulnerability classification, a vulnerability type classifier is constructed by extracting relevant text features and combining it with the OVO SVMs algorithm, and finally gives code snippets of vulnerability types, which is cumbersome and inefficient. Although the above results have achieved certain results, there is still much room for improvement.

Based on the above situation, this paper proposes a new Python source code vulnerability mining and classification method based on the CodeBERT model, aiming to achieve highly accurate identification and precise classification of Python vulnerabilities.

The specific contributions of this study are outlined as

979-8-3315-2931-4/24/$31.00 ©2024 IEEE

follows:

1. Constructed a Python source code vulnerability data set from the open source community by comparing diff file differences and the keyword ”Security”.

2. A multi-label and multi-classification CodeBERT[8] model was constructed to mine and classify Python source code vulnerabilities, with an accuracy rate of 97%-99% and a recall rate of 94%-98%.

II. METHODOLOGY

This section presents a Python source code vulnerability detection method designed for the GitHub open-source community, comprising three main modules. The comprehensive framework of the system is depicted in Fig 1.

Initially, the Transformer model undergoes pre-training to develop a comprehensive understanding of the syntax and semantics inherent in Python source code. Following this, the multi-label source code vulnerability data, segmented into blocks, is inputted into the pre-trained Transformer model for fine-tuning post data preprocessing[9].

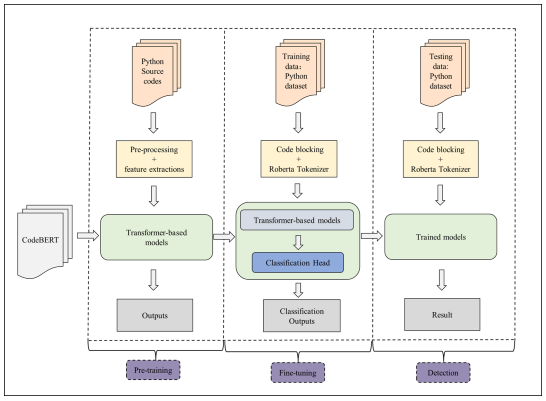

Figure 1: Frame structure.

A. Problem definition

In the context of the current version, a_i represents the code line of fragment ”i,” which is deemed non-vulnerable to attack. Conversely, p_i denotes the code line of fragment ”i” from the previous version, identified as vulnerable to attack. The equation $diff(a_i) = p_i$ signifies the process of comparing the code line from the diff file with that of the current version, thereby identifying the vulnerable lines of code from the previous version. This method is instrumental in compiling a vulnerability dataset.The vulnerability data set marked through diff files is shown in Fig 1.

Our objective is to identify the presence and types of vulnerabilities within the source code. In the initial step of the process, known as vulnerability dataset labeling, we approach the problem as a multi-classification task. Here, ”c” denotes the label category, where ”c” takes on values from 1 to 10, representing different vulnerability types. A label of ”c=0” signifies that the code fragment is deemed safe. Moving to the second step, fine-tuning of the vulnerability model takes place. This entails refining the pre-trained model using the labeled vulnerability dataset to develop the multi-classification detection model, denoted as $f()$. In the final step of vulnerability detection, the label of a given code fragment s_i is determined through the fine-tuned model $f()$. If $f(s_i)$ returns a value from 1 to 10, it indicates that s_i is vulnerable code, with the label number identifying the specific vulnerability type. Conversely, if $f(s_i) = 0$, it signifies that s_i is deemed safe code.

B. Data collection

The dataset utilized in this study was curated and structured based on the real-world vulnerability dataset collected and annotated by Wartschinski et al[6]. Furthermore, it was expanded from the original 7 types to encompass 10 types of vulnerabilities. Data collection involved crawling a substantial volume of source codes and diff files pertaining to Python files from the GitHub open-source community. Subsequently, irrelevant content was filtered out, and the differences between old and new versions were analyzed using the diff files to construct a vulnerability dataset $diff(a_i) = p_i$. The intricate process of determining vulnerable code fragments is elucidated in Fig 2.

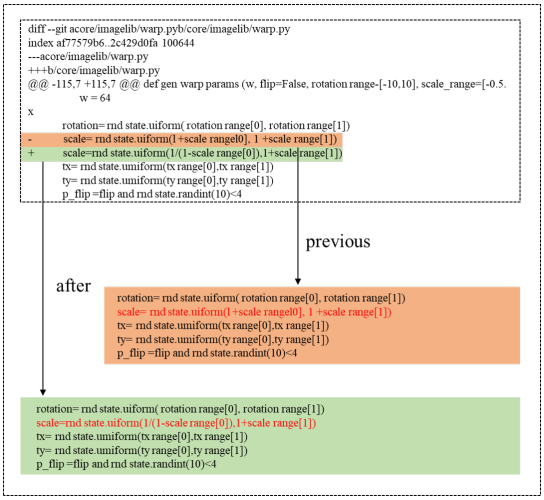

Figure 2: Source code diff file comparison fragment.

C. Data processing

The CodeBERT model is specifically designed to directly learn and analyze source code. In order to effectively leverage the capabilities of this model, it is imperative that the dataset contains contextual semantic information from the training data. Consequently, when labeling the dataset, careful consideration is given to the context surrounding each vulnerable code snippet. Furthermore, after identifying the vulnerable lines of code within the diff file, directly

marking the code snippets above and below as "vulnerable label numbers" may result in inconsistent block sizes for the remaining portions of the file. To ensure proper data processing and prevent imbalanced datasets[10], it is imperative to treat both vulnerable and non-vulnerable segments equally and proportionally during the labeling phase. Therefore, the proposed approach involves dividing the data into chunks of uniform length. If these chunks overlap with vulnerable code segments, they are assigned a vulnerable type label; otherwise, they are labeled as non-vulnerable. This strategy ensures consistency and balance in the dataset, as illustrated in Fig 3.

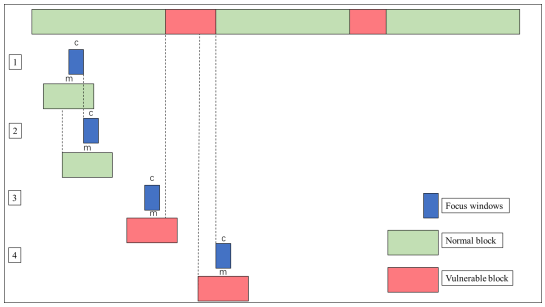

Figure 3: Source code chunking.

Utilize a focus window to systematically traverse the entire source code in increments of length "c." The focus window, denoted by a small blue box, commences and concludes at the characters that terminate individual tokens of Python code (e.g., colons, parentheses, or spaces) to prevent token fragmentation. Within this focus window, ascertain the surrounding context with an approximate length of "m" and regard it as a code block[11]. If the code block encompasses code statements bearing security risks, it is flagged as containing vulnerabilities; otherwise, it is designated as normal code.

D. Representation of source code

Liu et al[11]. argue for the necessity of Abstract Syntax Tree (AST) representation in mining patterns from code. Conversely, Russel et al[12]. and Hovsepyan et al[13]. have demonstrated that AST representation is not always essential, as code can also be effectively modeled using textual representations. Moreover, there are many similarities between language and source code, such as the inherent structure of language and source code, both of which are carriers for expressing meaning and conveying information.

Deep neural networks are adept at modeling diverse forms of sequence data, encompassing natural language, sensor data, and even music [citation needed], and have demonstrated remarkable performance in these domains. Consequently, they have found successful application in directly modeling source code as text.

E. Vulnerability detection model based on Transformer

1) Transformer: The Transformer is a deep learning model architecture grounded in the attention mechanism, originally introduced by Vaswani et al. A key characteristic of this model is its departure from the conventional recurrent neural network (RNN) structure, opting instead for a self-attention mechanism to capture dependencies across different positions within the input sequence[25]. Illustrated in Fig 4, the Transformer architecture primarily comprises two components: the encoder and the decoder. The encoder is tasked with transforming the input sequence into a contextually informed encoded representation, while the decoder utilizes this encoded representation to gen- erate the output sequence. Central to both components are key elements such as position encoding, multi-head attention, and fully connected layers.[14]. These operations are succinctly expressed in matrix form as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (1)$$

here d_k is the key vector's dimension, and Q, and K and V are matrices packed with all (multi-head) queries, (multi-head) keys and (multi-head) values, respectively.

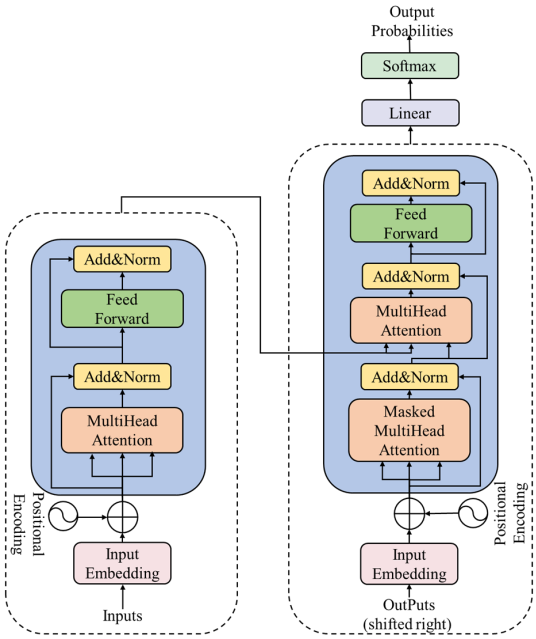

Figure 4: Transformer architecture.

2) CodeBERT: CodeBERT is a dual-mode pre-trained transformer model designed for programming and natural language tasks[8]. It undergoes training on bimodal data, consisting of both code and documents, sourced from languages such as Python, Java, JavaScript, Ruby, Go, and PHP.

The system employs BERT and CodeBERT models, which are constructed upon the Transformer architecture.

These models extend and refine the Transformer to acquire a more profound understanding of the semantic representation within the source code, thereby capturing essential information and semantics embedded in the code. This structural enhancement enables the models to exhibit heightened accuracy and superior generalization capabilities in detecting vulnerabilities within Python source code.

III. EMPIRICAL EVALUATION

A. Datasets

Currently, there exists a scarcity of open datasets specifically tailored for Python source code vulnerability detection, and those available are often lacking in comprehensiveness. Hence, this study utilizes a dataset founded upon the real-world vulnerability dataset assembled and annotated by Wartschinski et al[6]. The dataset is meticulously organized by scrutinizing the disparities between the old and new versions of diff files and employing a method that retrieves security-related keywords. Through this process, the dataset is expanded to encompass ten distinct categories.At the same time, in order to train the vulnerability detection model and the code vulnerability type classification model, we filtered the data and assigned labels. Among the 10 categories of code features, we have a total of 434,040 training data labels, including 309,682 vulnerability codes and 124,358 security codes.To further refine the dataset, a partitioning scheme is implemented, allocating the data into proportions of 0.7:0.15:0.15.

B. Evaluation metrics

Evaluation metrics play a pivotal role in machine learning assessment. Each metric underscores a different aspect, with the following four key concepts commonly forming the basis for evaluation in prediction and classification tasks: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). ”Positive” and ”negative” pertain to predictions, where a positive prediction denotes a ”fragile” outcome and a negative prediction signifies a ”not fragile” outcome.

Precision, defined as the ratio of true positives to all predicted positives, serves as a metric for assessing the accuracy of a model.

$$\text{precision} = \frac{TP}{TP + FP} \tag{2}$$

Recall, also referred to as sensitivity, quantifies the proportion of correctly identified positive instances relative to the total actual positive instances.

$$\text{recall} = \frac{TP}{TP + FN} \tag{3}$$

F1-Score, a balanced metric derived from the harmonic mean of precision and recall, provides a comprehensive assessment of a model's performance, incorporating both precision and recall into a single score.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

Accuracy, a fundamental metric in machine learning evaluation, represents the proportion of correct predictions relative to all predictions made by the model.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

C. Result & analysis

1) Experimental parameters: Hyperparameters exert significant influence on the predictive performance of deep learning models. This study investigated the variations in accuracy, precision, recall, and F1 Score evaluation metrics of the model across specific batch sizes and various training durations.

The number of training epochs significantly influences the predictive performance of the model. If the number of epochs is set too low, the model may fail to adequately capture the underlying features of the training data, leading to underfitting and poor predictive accuracy. Conversely, setting the number of epochs too high may result in the model memorizing the training data rather than learning generalizable patterns, resulting in overfitting and reduced generalization ability. Hence, this experiment examined training epochs ranging from 1 to 10 and analyzed their impact on accuracy, precision, recall, and F1 Score. The fluctuations in these metrics across the ten epochs are depicted in Fig 9 below.

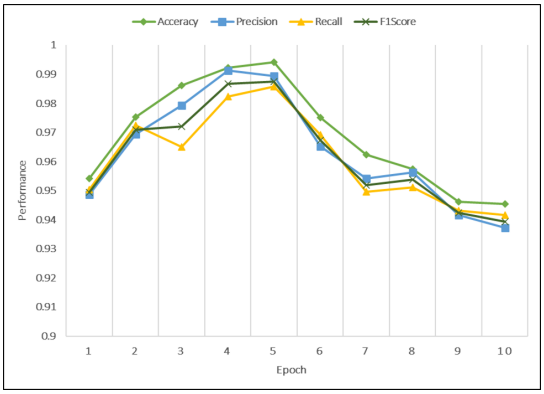

Figure 5: Changes in model indicators at different times of training.

The analysis of model performance across different training epochs, as illustrated in Fig 9, reveals that the model achieves optimal performance when trained for 5 epochs. During this period, the accuracy, precision, recall, F1 Score, and other metrics consistently stabilize at higher levels. As a result, it is recommended to set the number of training epochs for the model to 5.

Table I: Model hyperparameter settings

model	hidden size	hidden layers	attention heads	batch_size	Epoch
LSTM	100	1	0	128	100
BERT	768	12	12	16	10
CodeBERT	768	12	12	16	10

The analysis of model performance across different training epochs, as depicted in Fig 6, indicates that the model consistently achieves optimal performance when trained for 5 epochs. During this period, metrics such as accuracy, precision, recall, and F1 Score stabilize at higher levels, suggesting robust performance. Therefore, it is recommended to set the number of training epochs for the model to 5.

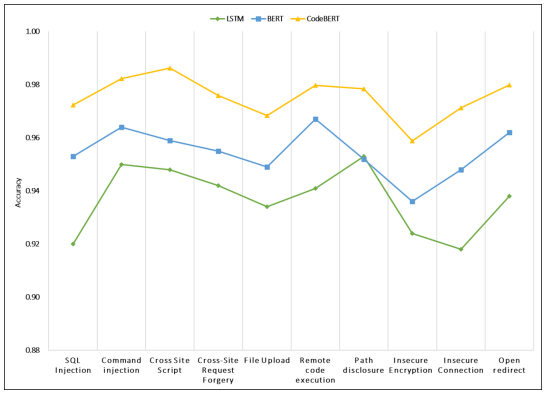

Figure 6: Accuracy indicator comparison.

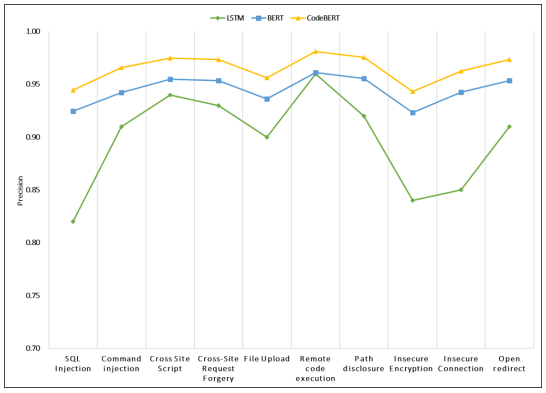

Figure 7: Precision indicator comparison.

Furthermore, experimental results demonstrate that varying the batch size has negligible impact on the accuracy, precision, and F1 Score of the model. Consequently, it is deemed appropriate to set the batch size to 16. Additionally, given the utilization of the BERT and CodeBERT models, parameters such as hidden layer size, number of hidden layers, dropout ratio, and activation function selection remain

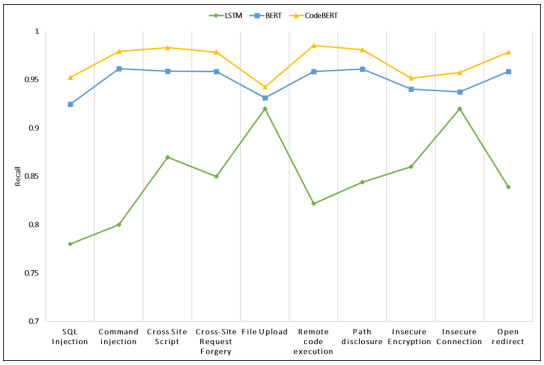

Figure 8: Recall indicator comparison.

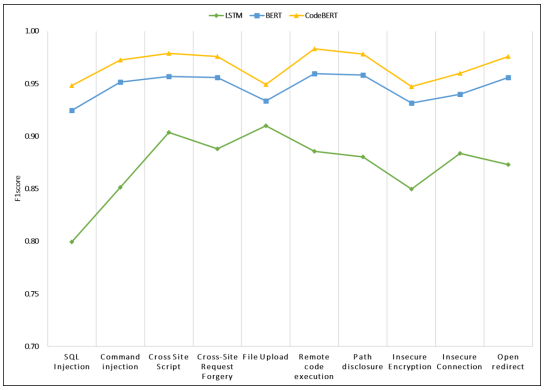

Figure 9: F1Score indicator comparison.

fixed. Thus, the various hyperparameter settings of the model are summarized in Table 1.

2) Model performance: To validate the effectiveness of the CodeBERT model, a comparative analysis was conducted against the BERT and LSTM deep learning models. This comparison was performed across 10 distinct vulnerability datasets, ensuring consistency in dataset selection, data preprocessing methodologies, and optimal hyperparameter configurations. The performance metrics of the three models were evaluated under varying dataset conditions, as depicted in Figures 7 through 10.

In the comparison of multiple metrics across different datasets, it is evident that both the CodeBERT and BERT models exhibit notable improvements over the LSTM model utilized by Wartschinski et al[6]. in source code vulnerability detection. Notably, there are enhancements observed in accuracy, F1Score, and recall metrics, with accuracy ranging 95.89% - 98.63%, and F1Score 94.74% - 98.33%. These findings underscore the efficacy of BERT and CodeBERT, built on the Transformer model, in source code vulnerability detection. Leveraging the attention mechanism, these models not only capture sequence features effectively but also prioritize key vulnerability characteristics, thus leading to enhanced detection performance compared to LSTM-based approaches.

Table II: CodeBERT multi-classification model evaluation

Type	Accuracy	Precision	Recall	F1Score
SQL Injection	0.97	0.94	0.95	0.95
Command injection	0.99	0.97	0.98	0.97
Cross Site Script	0.98	0.97	0.98	0.98
Cross-Site Request Forgery	0.97	0.97	0.98	0.98
File Upload	0.98	0.96	0.94	0.95
Remote code execution	0.98	0.98	0.99	0.98
Path disclosure	0.98	0.98	0.98	0.98
Insecure Encryption	0.96	0.94	0.95	0.95
Insecure Connection	0.97	0.96	0.96	0.96
Open redirect	0.98	0.97	0.98	0.98

In the vulnerability type classification experiment, since the code and data set of Guo et al[7]. have not been made public, comparative experiments cannot be conducted. As shown in Table 2, the performance of the CodeBERT model in multi-classification experiments is demonstrated. As can be seen from the table, among the 10 vulnerability types, the accuracy rate of CodeBERT multi-classification is about 96%-98%, the precision rate is 94%-98%, the recall rate is 94%-99%, and the F1-score is 95%-98%, all indicators show excellent results.

IV. CONCLUSION

This article builds a deep learning-based CodeBERT vulnerability mining model for Python source code. In the past, due to the lack of public Python source code data sets, this paper combined diff files and Security keywords to build a multi-label vulnerability data set based on contextual semantics. At the same time, in order to deeply learn semantic information and extract deep semantic structures, a multi-classification CodeBERT detection model was constructed. Compared with LSTM and BiLSTM, this model can extract and learn semantic information at a deeper level.

When performing multi-classification, we directly added corresponding classification headers in CodeBERT based on the multi-label vulnerability data set to determine the type of vulnerability. Since the code of Guo et al[7]. has not been made public, we cannot conduct comparative experiments, but according to the indicators provided in the paper, our method is more effective. Currently, the system can only detect whether there are vulnerabilities in the code, but cannot accurately locate the vulnerabilities. We will continue to work hard to achieve this goal.

REFERENCES

[1] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, vol. 12, no. 110, p. 1, 2012.

[2] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pp. 250–261, 2016.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[4] S. Micheelsen and B. Thalmann, "A static analysis tool for detecting security vulnerabilities in python web applications," *Aalborg University, Aalborg University, 31st May*, 2016.

[5] S. Peng, P. Liu, and J. Han, "A python security analysis framework in integrity verification and vulnerability detection," *Wuhan University Journal of Natural Sciences*, vol. 24, no. 2, pp. 141–148, 2019.

[6] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809, 2022.

[7] W. Guo, C. Huang, W. Niu, and Y. Fang, "Intelligent mining vulnerabilities in python code snippets," *Journal of Intelligent & Fuzzy Systems*, vol. 41, no. 2, pp. 3615–3628, 2021.

[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," *arXiv preprint arXiv:2002.08155*, 2020.

[9] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th Annual Computer Security Applications Conference*, pp. 481–496, 2022.

[10] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 99–108, IEEE, 2015.

[11] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[12] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762, IEEE, 2018.

[13] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*, pp. 7–10, 2012.

[14] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image transformer," in *International conference on machine learning*, pp. 4055–4064, PMLR, 2018.