Semantic Embeddings for Prediction of Vulnerable and Non-Vulnerable codes and their CWE Categories

Harsha Jalluri.
Z1935854
Northern Illinois University
Email: harshajalluri@gmail.com

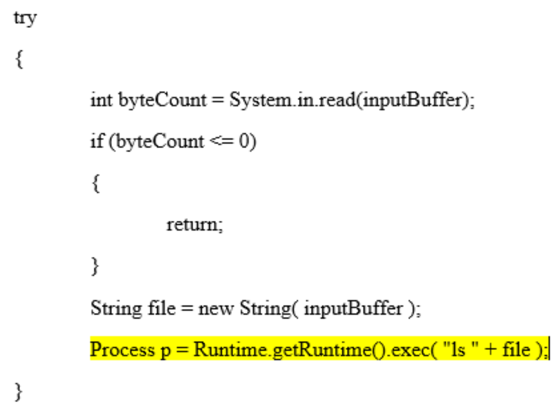
Satya Sree Varsha Ponnaganti
Z1907280
Northern Illinois University
Email: varshaponnaganti9726@gmail.com

Abstract- Today, a lack of software security costs the US economy billions of dollars. A vulnerable code is easily exploited by attackers and the increase in complexity of software codes needs primary prediction of code snippet if it contains a vulnerability. In this paper, we aimed to predict the vulnerable and non-vulnerable codes in java programming language using deep learning techniques. For this we utilized CodeBERT and Sentence BERT models as embeddings solution to facilitate the detection of good code and bad codes. Additionally, each vulnerable and non-vulnerable code is associated with Common Weakness Enumeration ID, a list of software and hardware weaknesses maintained by MITRE Corporation. We also focused on predicting these CWE-ID's using the BERT based embeddings. Classifying each class of bad, good and CWE types is done by using various classification algorithms. Further we evaluated the models and compared the results with accuracy, precision and recall score.

Index Terms – semantics, *vulnerable, non-vulnerable, CWE-ID, embeddings, transformers, SBERT, CodeBERT, classification models, scikit learn.*

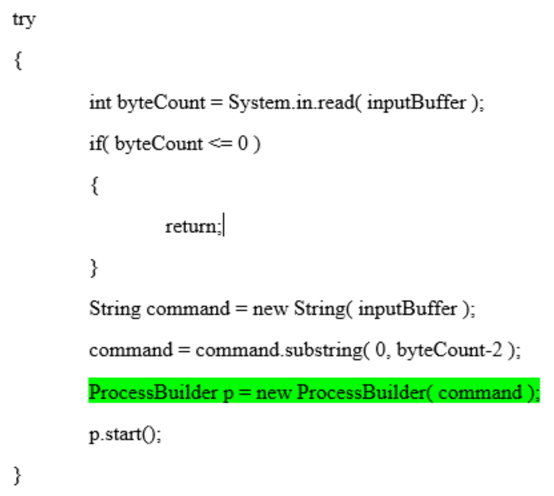
INTRODUCTION

The software business and the field of cybersecurity are becoming increasingly concerned about the steadily rising number of disclosed security vulnerabilities. Large amounts of software code are now readily available owing to the thriving open-source software community, which enables machine learning and data mining approaches to take advantage of the abundance of patterns found in software code. One such platform is SARD. The Software Assurance Reference Dataset (SARD) [1] is a large database of test cases and test suites that aids tool developers in enhancing their solutions and end users in finding the appropriate tools for their projects. It is a subject of National Institute of Standards and Technology (NIST). Software assurance tools look for errors in the code. We require programs with known defects as the ground truth in order to test such technologies. The Software Assurance Reference Dataset (SARD) is a widely accessible repository of more than 100,000 test cases written in several programming languages that cover numerous classes of flaws, comprising those included in the Common Weakness Enumeration (CWE).
Deep learning has recently made significant strides in the understanding of susceptible code patterns and semantics, which have been indicators of the features of vulnerable code and non-vulnerable codes. Data collection, data preparation, model construction, and evaluation are the four processes that make up the deep learning application process for vulnerability detection.
In this paper, we incorporate a BERT-based embedding solution for predicting vulnerable and non-vulnerable codes in JAVA programming language and classify the CWE types of the software vulnerabilities.
Java is so widely used programming language in developing software products and has been the default choice of scientific applications including Natural Language Processing (NLP). The main reason being its safety, portability and maintainability characteristics and it even has high-level concurrency tools than other programming languages like C++. It is currently used on more than 3 billion devices and comes with strong security protocols.
However, there are times where java versions offers no protection for users. There are at least eight zero-day-attacks [3] targeting the Java platform[4]. Sometimes the attacks require no user interaction, when an user simply visits a malicious we page based on java-enabled browser, systems are being attacked. Fortunately, most of the java vulnerabilities have ways to prevent them. Figures 1(a) and 1(b) show a vulnerable java code and a fix for it. These vulnerabilities are categorized as CWE ids in SARD. Therefore, we chose to work on java vulnerabilities and types of the vulnerability.
Some currently available vulnerability detectors with conventional embedding techniques, including Word2Vec, GloVe, and FastText, usually have low recall and precision [5]. Few works show identification of PHP code vulnerabilities using Graph Neural networks that yielded good result, however confined to only one programming language [6]. Therefore, we considered Sentence-BERT (SBERT) and CodeBERT as a code embedding and feature generator for java programming language. SBERT learns distributed representations and produces real-valued vector that encodes the semantics of the word such that the words are closer in the vector space are expected to be similar in meaning. In order to make the semantic distinctions between the vulnerable and non-vulnerable code snippets measurable, this model considers the code snippets as natural language and constructs the embeddings. On the other hand, CodeBERT, a bidirectional transformer is capable of capturing long-distance relationships between code sequences. It can capture latent vulnerable code patterns, maintain context relationships, and reduce information loss. It can improve in the interpretation of code semantics within a large environment. Therefore, when contrasted to noncontextual embedding techniques like Word2vec, GloVe, and FastText, CodeBERT has the potential to produce more complex and insightful code embeddings.



```
try
{
        int byteCount = System.in.read(inputBuffer);
        if (byteCount <= 0)
        {
                return;
        }
        String file = new String( inputBuffer );
        Process p = Runtime.getRuntime().exec( "ls " + file );
}
```

Figure 1(a). Vulnerable code where the string file is not validated before the execution.



```
try
{
        int byteCount = System.in.read( inputBuffer );
        if( byteCount <= 0 )
        {
                return;
        }
        String command = new String( inputBuffer );
        command = command.substring( 0, byteCount-2 );
        ProcessBuilder p = new ProcessBuilder( command );
        p.start();
}
```

Figure 1(b). Non-Vulnerable code where the function start checks the command is a valid operating system command.

We applied vulnerable and non-vulnerable codes derived from SARD on SBERT and CodeBERT to retrieve the embeddings. We focused on extracting the good and bad functions from an entire java class as most of the vulnerabilities and their fixes are found at function level. The embeddings of these functions are further passed to different classifiers, Decision Trees, Naïve Bayes, Random Forest and KNearest Neighbors to predict the good and bad codes and to classify the CWE types. We further performed a systematic evaluation on several classification algorithms using accuracy score, precision score, recall score, and Area Under the ROC curve (ROC AUC) score. Frequently used acronyms in this paper are summarized in Table 1.

METHODOLOGY

This project is divided into three parts, collecting the dataset, generating the embeddings from SBERT and CodeBERT, declaring a ground truth and passing the embeddings and predictable variables i.e., good, and bad code labels and CWE types to classifiers for prediction.

Acronyms	**Definition**
DL	Deep Learning
NLP	Natural Language Processing
ML	Machine Learning
SARD	Software Assurance Reference Dataset
SBERT	Sentence BERT
CWE	Common Weakness Enumeration
ROC AUC	Receiver Operating Characteristics, Area Under Curve
NB	Naïve Bayes
RF	Random Forest
DT	Decision Trees
KNN	KNearest Neighbours

Table 1. List of Acronyms

I. Building the Dataset:

Initially, we collected a test suite of java programming language from SARD open-source platform. Each test case folder of a CWE category has hundreds of java classes. Each java class has bad methods (vulnerable code) and good methods (non-vulnerable code). For each bad code, there are more than two good codes with associated fixes. goodB2G () is used for bad source and good sink whereas goodB2G () is used for good source and bad sink code snippets. As mentioned before, most of the vulnerabilities and their associated fixes reflect at function/method level, so we decided to extract all the methods using srcML [7] tool. We used a parser to extract all the functions from SARD source code. SrcML provides XML format for source code and is scalable for multiple languages. Firstly we defined functions in python to convert java to xml. Next, for parsing xml files, another function was used to extract all the methods and explode them separately into different rows. Finally, we converted all xml files back to java source code. All the functions in this process used srcML command. For classifying good and bad codes, we separated all the extracted methods and labelled them bad and good according to the method name. Additionally, we removed all the instances that appeared in both classes that were irrelevant to the vulnerability and that remained unchanged in the good methods as well. It is necessary to offer sufficient of a particular class to the deep learning model, so we discarded the CWE subsets of less than 10 samples. The final dataset contains 10,154 good codes, 4769 bad codes with 52 unique CWE-ID's. Table 2 represents the variables we have in the dataset.

CWE - ID	**Code**	**Label**
CWE - 129	public void goodG2B(..)…	good
CWE - 129	public void bad(..)...	bad

Table 2. Dataset Features

II. Generating Embeddings:

The data is then transferred to the following stage, where it will be converted into code embedding vectors. Word2Vec is one NLP model that generates a unique vector for each word in a word corpus. However, it neglects the relationship between terms, such as "apple" and "apples," for instance. The two words share a similar internal morphology, making them comparable. Unfortunately, Word2Vec converts both words, therefore this internal form will not be taken into account. In order to avoid this problem, SBERT processes two words as pairs using two almost identical BERT structures that use the same weights. Each BERT outputs pooled sentence embeddings that have to go through pooling technique where generalizing of features take place. The authors of SBERT research paper[8] finalized that the mean-pooling approach yielded better results. The two embeddings after pooling technique are now combined by SBERT, which will then subject them to a softmax classifier and train it with a softmax-loss function. Figure 2 shows the structure of SBERT. We used the pre-trained BERT model from sentence_transformers to load the model in our device. The code is then passed as sentences to the model for generating the embeddings.

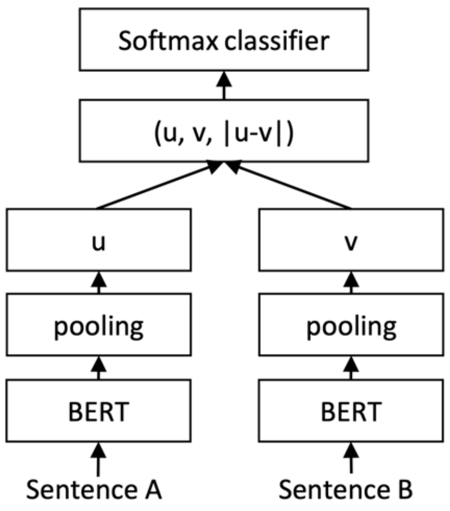


Figure 2. SBERT Architecture

As mentioned previously, FastText and GloVe are other embedding models for converting textual tokens to meaningful vectors. The former model has three layers, input layer, hidden layer and output layer and its structure is simple. GloVe employs the matrix factorization technique, and training progress is made quickly. The resulting word vector only has a limited amount of semantic information and is only useful for specific purposes like calculating similarity because it solely focuses on co-occurrence. On whole another level, CodeBERT captures the semantic connection between natural language and programming language and produces general-purpose representations that can broadly support NL-PL understanding tasks [9]. It's a multilayer two-way transformer. It comprises 12 layers, 12 self-attention heads on each layer, a size of 64 for each self-attention head, and a hidden dimension of 768. The detection of vulnerabilities will be impacted by these variations in model structure and computer architecture.

For the analysis of numerous types of vulnerabilities, context information is essential. In two functions, the range of values of variables can be different. Therefore, we decided to use CodeBERT model for generating the embeddings.
Since we do not focus on natural language code description, we utilized only the code tokens part of this model. Similar to SBERT, we used the pre-trained model of CodeBERT for embeddings generation. This model is originally trained on six programming languages, namely, java, python, ruby, PHP, Go, and JavaScript. Firstly this model converts the code to code tokens and then give each token an id. These token ids are then assed to model for producing context embeddings. The clear representation of CodeBERT with NL + PL is illustrated in Figure 3. With this embedding representation, we will addressing the following questions with results.

Research Question 1 (RQ1): Is Semantic information enough to predict Good and Bad codes?

Research Question 2 (RQ2): Can we classify CWE Categories from semantic information?

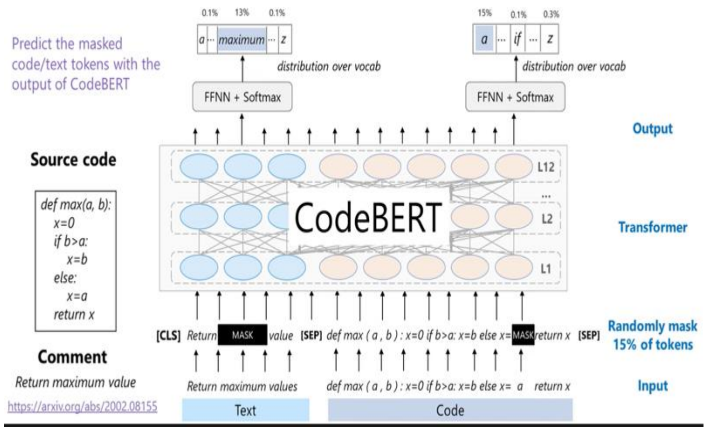


Figure 3. CodeBERT Architecture

III. Classification:

Within a machine learning model, an embedding can be utilized as a general free-text feature encoder [10]. Any machine learning model's performance will be enhanced by embeddings. They can also serve as a categorical feature encoder. It is more valuable if the categorical variables are meaningful and abundant. In this part, we split the dataset the contains embeddings and our predictable variables into training set and a testing set which will be used for classification. We implemented four classification models and compared their performances with visualization techniques like heatmap, ROC AUC curve etc.
The overview of this project's methodology is presented in figure 4.

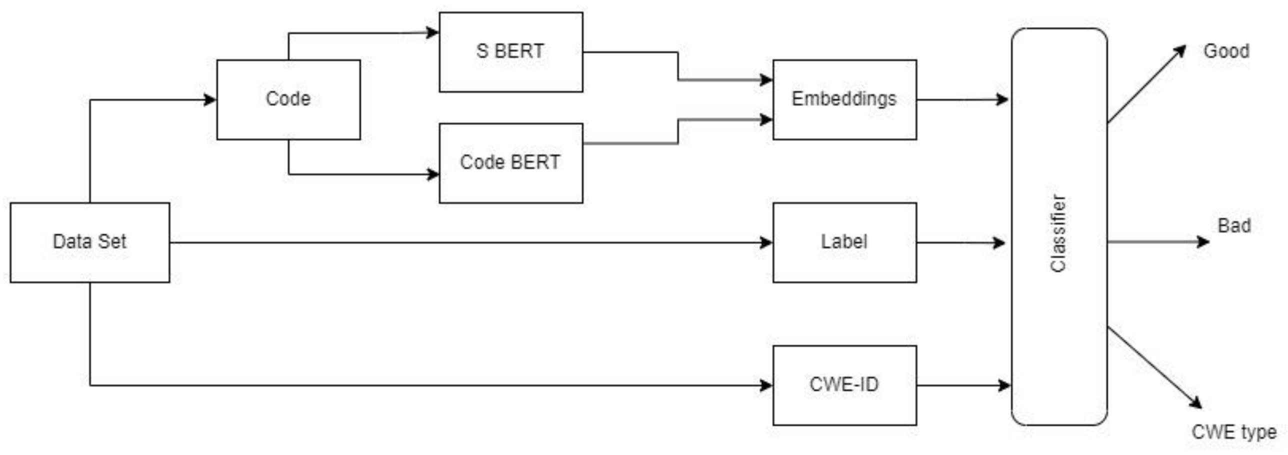


Figure.4 The workflow of the proposed model.

RELATED WORK

Deep learning-based vulnerability detection has attracted more attention in recent times. Various techniques have been proposed showing software vulnerability detection.

Hazim Hanif, Rishi Rabheru [11] proposed a DeepTective model for detecting vulnerabilities in PHP source code. They leveraged both syntactic and semantic information to discover vulnerabilities. The proposed model has two key components, Gated Recurrent Unit (GRU) and Graph Convolution Network (GCN). They used SARD vulnerable and non-vulnerable samples and real-world projects (GIT). Their proposed model achieved 99.95% accuracy on SARD dataset whereas on GIT data, the accuracy was 82.78%. In addition, other DL architectures include long short-term memory(LSTM). Fang. Y and Han. S[12] designed a static analysis for detection of PHP vulnerabilities based on DL technology. They combined a model using word2vec and LSTM that achieved 99% AUC and 97% accuracy. Choosing GloVe over Word2vec, Henkel and Lahiri [13] generated vectors that learn from Abstracted Symbolic Traces of C programs. They presented that GloVe learns word vectors as a dot-product of two vectors and is close to the logarithm of their probability of co-occurrence. They were able to achieve 93% accuracy. This paper also showed that semantic abstractions resulted in more accuracy than syntactic abstraction of embeddings. Additionally, FastText was employed in FastEmbed for ensemble machine learning model-based vulnerability prediction [14]. The authors provided a prediction model that is a combination of fastText and LightGBM algorithm. A Devign model comprising of Graph Neural network for graph level classification is designed by Zhou, Yaqin [15]. The models used in this paper were trained on a rich set of code semantics of C programming codes that secured 10% more accuracy than other state of the art models like BiLSTM. [16] describes a set of metrics at lower granularity levels and showcases evidence for vulnerability prediction. This paper is inclined towards Java programming language and used software metrics to predict vulnerable code methods. They divided class-level metrics and method-level metrics and achieved more than 70% recall and precision scores. A model named SCELMo was proposed by Karampatsis and Sutton [16] to generate contextual code representations. They trained ML systems on small corpus of programs for bug detection.
According to these studies, contextualized embedding models are efficient for a variety of code analysis tasks.

In our work, we use CodeBERT, which is also a pretrained model for contextualized embedding representation for specific code analysis task which is vulnerable and non-vulnerable detection.

IMPLEMENTATION AND EVALUATION

The selected source code function samples were passed to the SBERT embedder and CodeBERT model for producing the embeddings. The final dataset contains the generated embeddings, CWE categories, label as good or bad. We further converted the categorical variables to numerical variables, for instance, 1 for good code and 0 for bad code. We used to factorize () method from pandas library to convert all the unique CWE ids to numerical format.
In the classification task we predict the predefined categories given an input. We divided the dataset into 80% training set and 20% testing test. Firstly, we predicted the bad code and good code label based on embeddings of the codes from SBERT and CodeBERT. Our independent variable will be embeddings and our predictable variable is 0 or 1 for bad and good respectively.
To evaluate our classification models we used:

1. Accuracy: This shows if both true positive (TP) and true negatives (TN) are correctly classified.

$$ACC = \frac{tp + tn}{tp + fp + tn + fn}$$

2. Precision: The score is the ratio of true positives and total positives predicted.

$$P = \frac{TP}{TP+FP}$$

3. Recall: This is the ratio of true positives to all the positives in the ground truth.

$$R = \frac{TP}{TP+FN}$$

4. ROC AUC: ROC is the probability curve and AUC is the measure of separability. We can know how much the model is capable of distinguishing between classes.

RQ 1:
We used the following classification algorithms from sklearn library in python:

i. Naïve Bayes:

The Naive Bayes algorithm builds on Bayes' theorem that adopts the assumption that every pair of features is independent of one another. In many real-world scenarios, naive Bayes classifiers perform well. The number of parameters required for naive Bayes classifiers is linear in the number of variables (features/predictors) in a learning problem, making them extremely scalable. Naive Bayes has the benefit of requiring a limited set of training data to estimate the classification parameters.

SBERT: This classifier trained with embeddings from SBERT resulted in 97% accuracy, 100% precision score and 96.5% recall score. The ROC AUC score also achieved 99% (Figure 5.1).

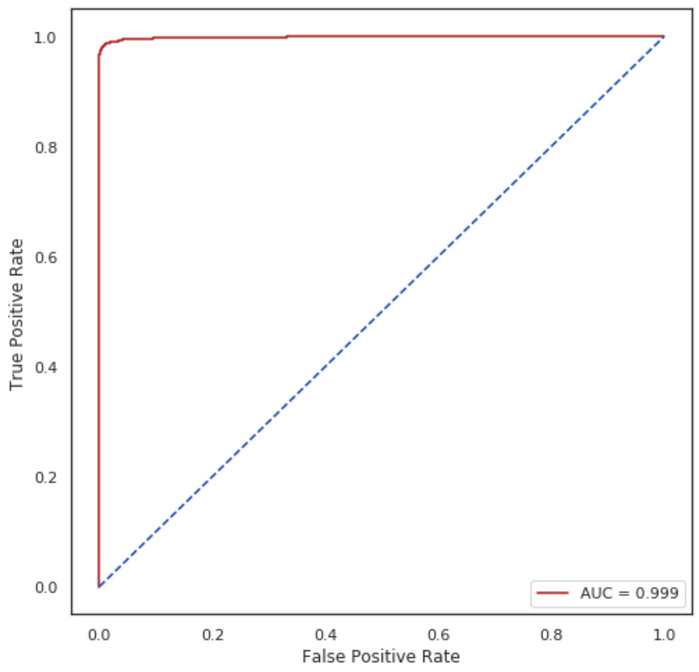


Figure 5.1: ROC AUC for SBERT – Naïve Bayes Classifier

CodeBERT: On the other hand, Naïve Bayes trained on CodeBERT embeddings gave 73% accuracy, 86.73% precision score, 72% recall score, and 79% ROC AUC (Figure 5.2).

A thorough comparison with different classification algorithms revealed that various methods, such boosted trees, outperformed Bayes classification [13].

ii. Decision Trees:

A decision tree generates a set of rules that can be used to categorize the data given a set of attributes and their classes. A decision tree incorporates a tree-like representation of decisions and their potential effects, such as utility, resource costs, and chance event outcomes. One technique to show an algorithm that solely uses conditional control statements is to use this method. It is simple to understand and visualize and moreover this needs little data preparation.

SBERT: The accuracy for this model from SBERT embeddings was 98.62%, precision and recall scores were 99% and 98% respectively and the ROC AUC score was also 98.5%. We plotted a graph against True Positive Rate and False Positive Rate (Figure 6.1).

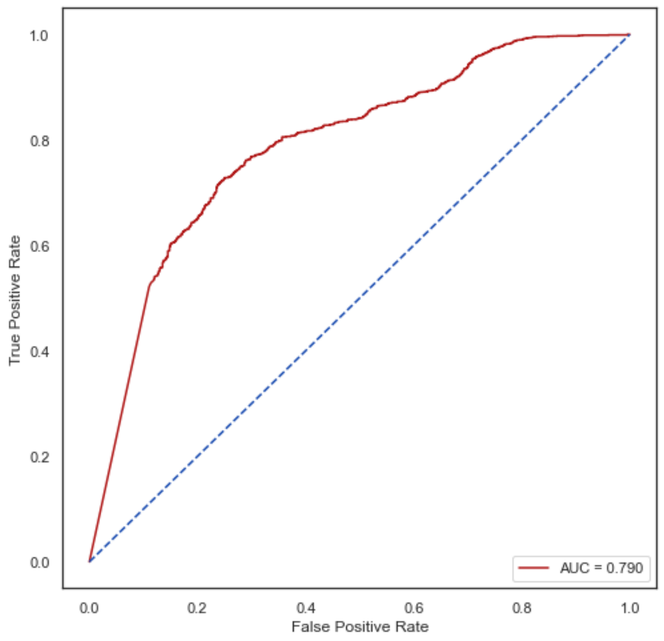


Figure 5.2 ROC AUC for CodeBERT – Naïve Bayes Classifier

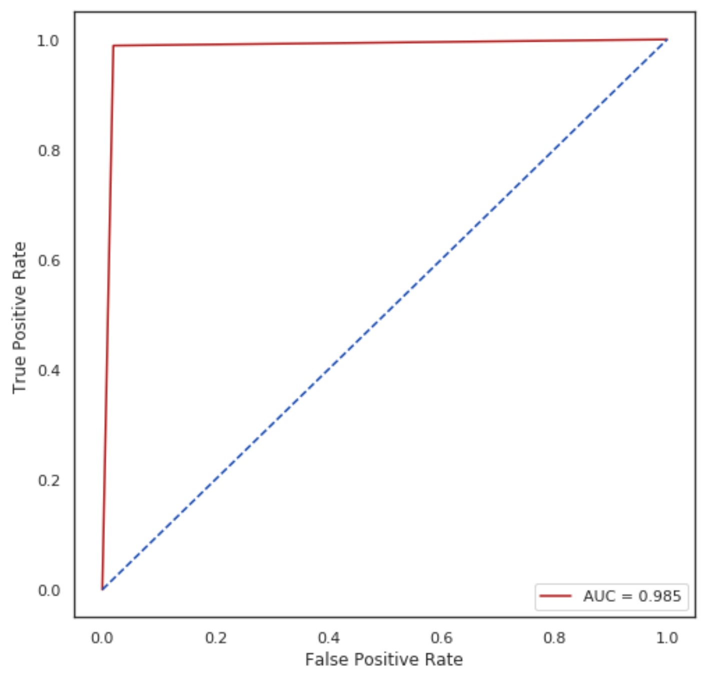


Figure 6.1: ROC AUC for SBERT – Decision Tree Classifier

CodeBERT: CodeBERT embeddings yielded 94% accuracy with 95% precision score, 96% recall score and 92.6% ROC AUC score. Figure 6.2 shows the ROC for this model.

iii. K-Nearest Neighbours

Neighbours-based classification is a form of lazy learning because it doesn't try to build a broad internal model; instead, it just keeps examples of the training data. The k nearest neighbours of each point have a simple majority vote to determine the classification. This algorithm is easy to use, tolerant to noisy training data, and efficient with huge amounts of training data.

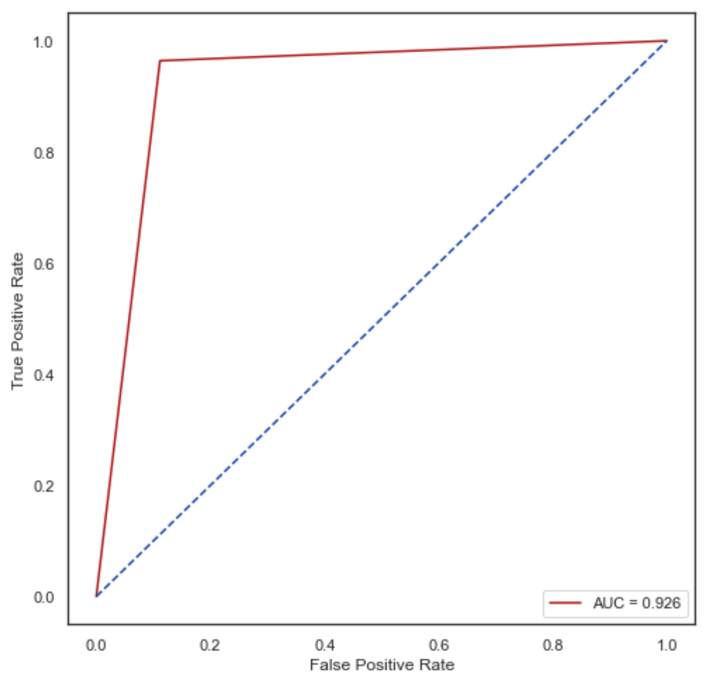


Figure 6.2 ROC AUC for CodeBERT – Decision Tree Classifier

SBERT: Here, compared to Naïve Bayes, the accuracy, precision score and recall achieved 99% and ROC AUC score attained 100 % score. (Figure 7.1)

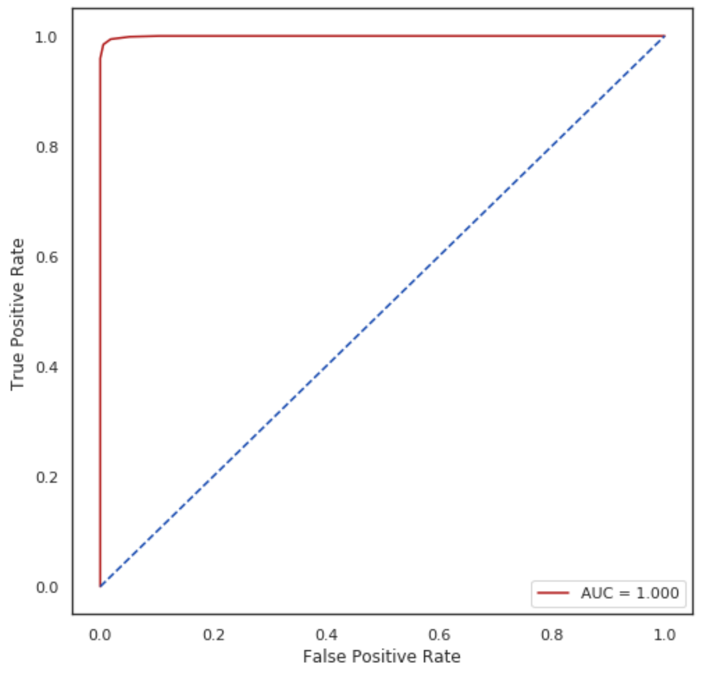


Figure 7.1: ROC AUC for SBERT – KNN Classifier

CodeBERT: This model achieved comparatively better results than previous classifiers with 93.4% accuracy, 94.5% precision, 96% recall and 97% ROC AUC (Figure 7.2).

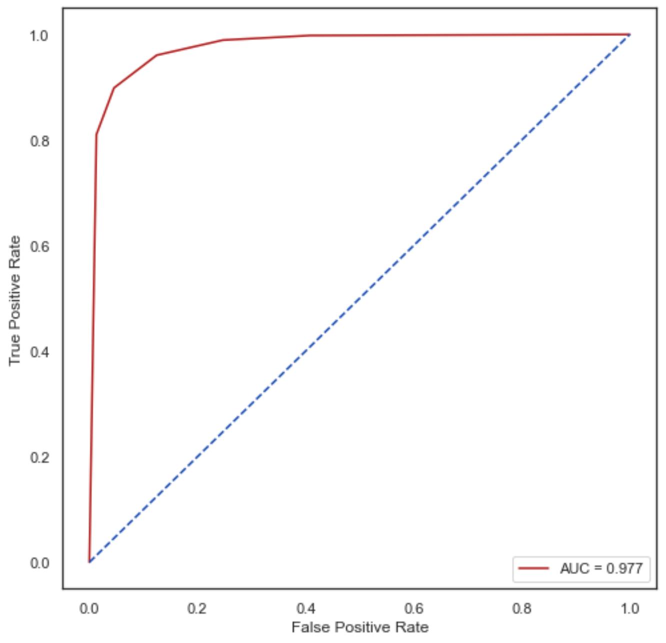


Figure 7.2 ROC AUC for CodeBERT – KNN Classifier

Figure. 8 illustrates a heatmap of all classifiers predicting bad and good codes with SBERT embeddings.

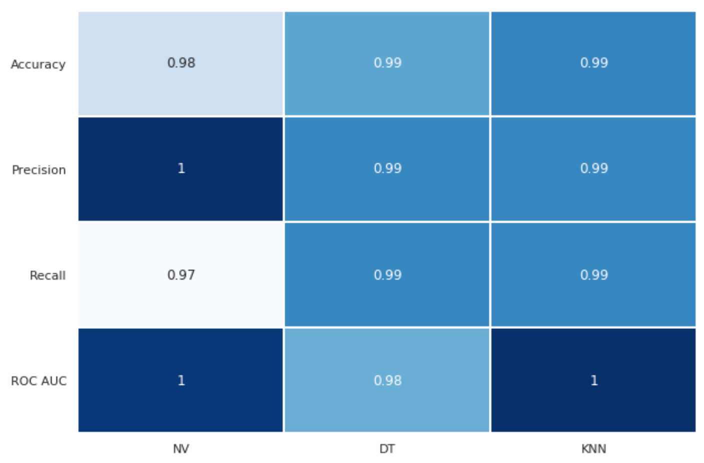


Figure. 8. Heat Map for SBERT – Bad/Good Classification

iv. Random Forest

We implemented Random Forest classifier for CodeBERT embeddings to get better accuracy score as it reduces over-fitting and this classifier is more accurate than Decision Trees in most cases. It fits a set of decision trees on various sub-samples of datasets. Although the samples are drawn via replacement, the sub-sample size is always the same as the original input sample size. As expected, this classifier yielded 98.5% accuracy with 99% ROC AUC score Figure(9).

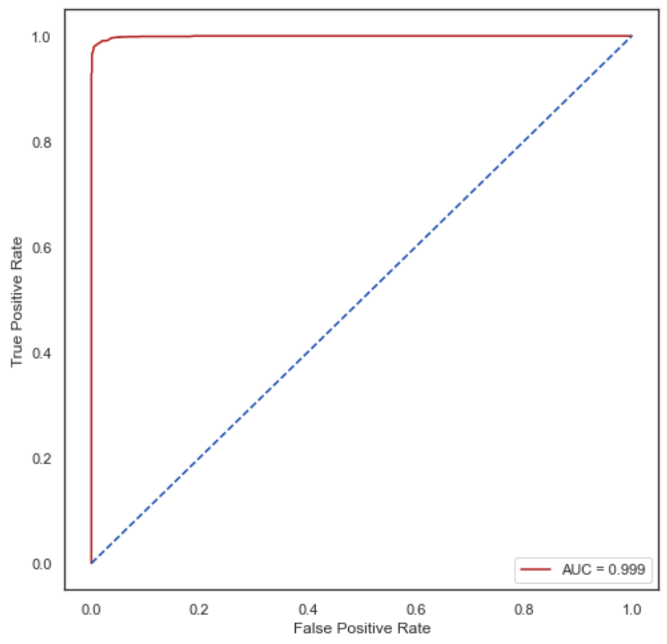


Figure 9. ROC AUC for CodeBERT – RF Classifier

Figure 10. illustrates a heatmap of all classifiers predicting bad and good codes with CodeBERT embeddings.

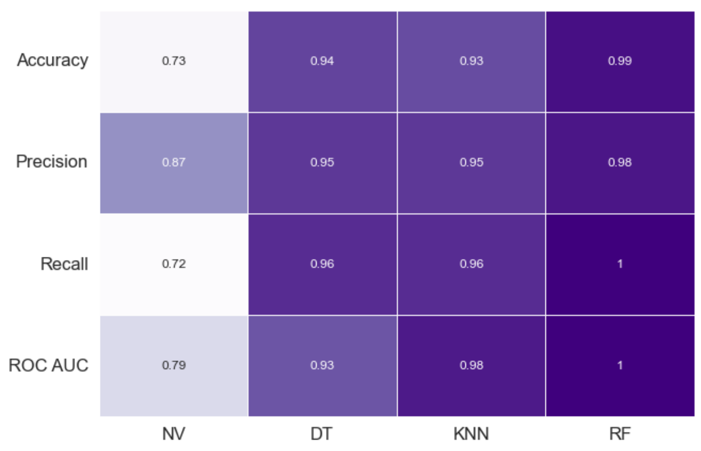


Figure. 10. Heat Map for CodeBERT – Bad/Good Classification

RQ 2:

We predicted the CWE Categories based on embeddings of the codes from SBERT and CodeBERT. There are about 52 unique CWE ids in dataset. Our independent variable for the classifiers will be embeddings and our predictable variables are CWE types.

SBERT:

We used the same classification models for predicting CWE types as well because most of them are known to classify multi-class.

Figure 11 illustrates the results of each classifier for predicting the CWE ids.

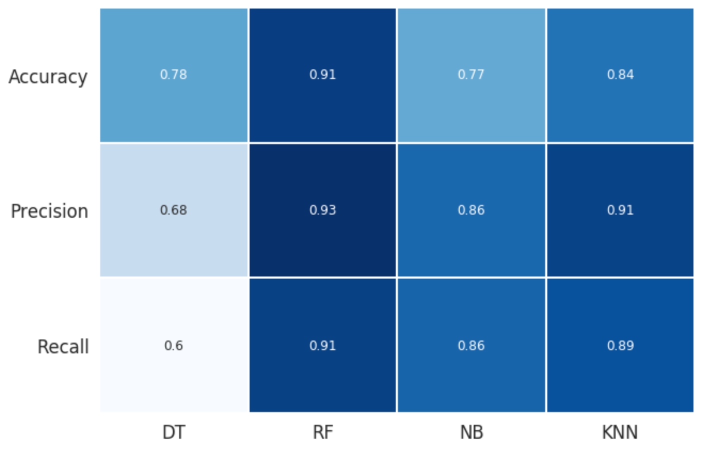


Figure. 11. Heat Map for SBERT – CWE Category Classification

Results of **CodeBERT** embeddings for predicting the CWE categories from the classifiers is represented in Figure. 12.

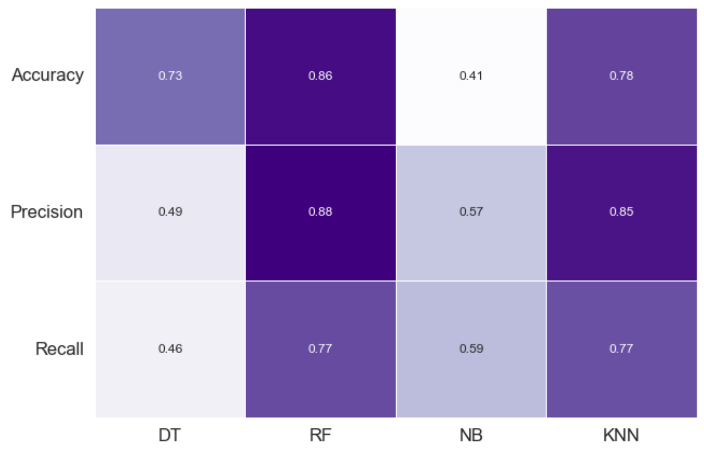


Figure. 12. Heat Map for CodeBERT – CWE Category Classification

Discussion:

In general, we observed that the semantic properties are distinctive and almost all the classifiers showed good performance. This indicates that the code embeddings generated by CodeBERT and SBERT facilitate vulnerability detection along with classifying CWE categories. As compared to SBERT, CodeBERT is more complex and has a large capacity. It might save a substantial amount of data that the other models have missed, like certain possible code patterns and semantic properties. In addition to this, as a lot of calculation is required to understand the complexity of parameters, it took a long time to generate embeddings for our dataset records. Moreover, vulnerability features are usually found at the logic code. Therefore the vulnerability features may be cut off and the model learns all some unwanted features. On the other hand, SBERT is a faster NLP model than other BERT models. It leverages high-quality labelled sentence pairs. Based on the outputs, it was high speed embeddings generating model and was notably good given its simplicity and word independence assumptions. Figures 13.1 and 13.2 represents the outputs of all classification models between SBERT and CodeBERT against the accuracy score.

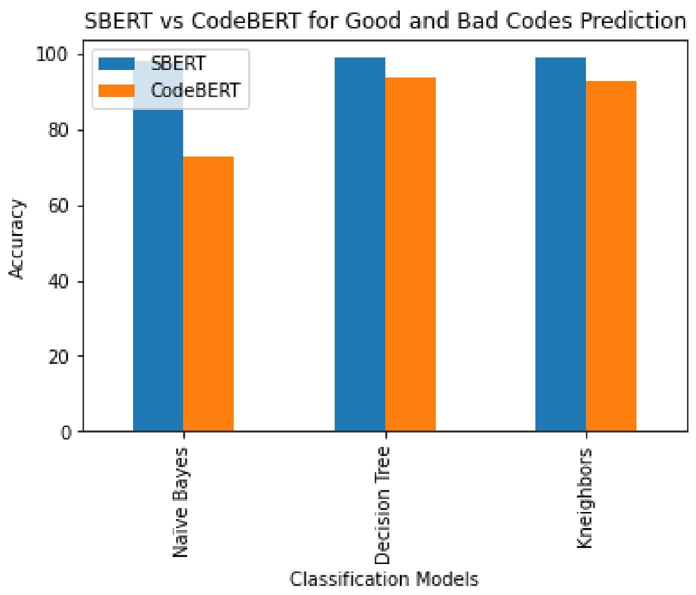


Figure. 13.1. Comparison of SBERT and CodeBERT

SBERT performed surprisingly better that CodeBERT. This maybe because of complexity of CodeBERT model or simplicity of SBERT. In the later model, the code is considered as sentences and this may affect the semantic understanding between code snippets. Binary Classification of good and bad codes from Decision Trees and KNN models achieved better results that Naïve Bayes for both the NLP models.

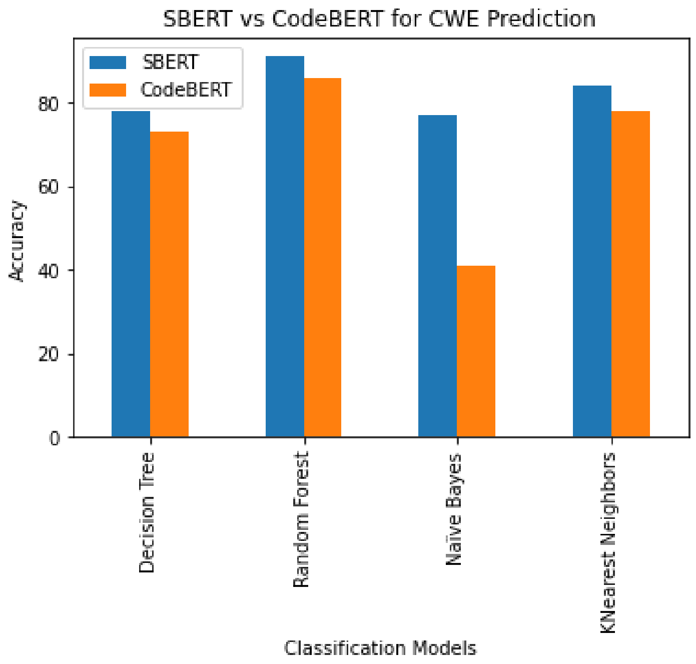


Figure. 13.1. Comparison of SBERT and CodeBERT

CONCLUSION AND FUTURE WORK

This study suggests a BERT-based embedding technique for vulnerability detection. SBERT and CodeBERT are proven to be performing well when it comes to embeddings for classification problem. SBERT is not familiar with syntax and semantics of Java language, however, if pretrained on the code language, it will have great potential of learning productive features of programming codes. Further research effort could be developing this Sentence BERT model to adapt to programming language. Meanwhile, CodeBERT already trained on java language, shows scalable results on each classification algorithms. However, the 12 encoder layers and over 110 million parameters in CodeBERT make it an expensive system to train and implement. A lightweight model of CodeBERT proposal could be useful. In addition to this, SARD dataset contains safe and unsafe samples and are designed to test security tools. This project can be extended with a real world dataset to test the models. Although a real world dataset is not labelled as vulnerable or non-vulnerable code, based on CWE descriptions or commit messages on GIT repository can be used for the labelling process. We can speed up the data collection process if a solution for automated vulnerability is proposed. Nevertheless, our experiments results show that this embeddings-generation as a solution approach is capable of facilitating the detection of vulnerabilities in Java source codes. While classifying the multi-class CWE categories

REFERENCES

[1] Black, Paul E. "Sard: a software assurance reference dataset." (2017).

[2] Black, Paul E. "A software assurance reference dataset: Thousands of programs with known bugs." *Journal of research of the National Institute of Standards and Technology* 123 (2018): 1.

[3] "What is a Zero-Day Vulnerability?". *pctools*. Symantec. Archived from the original on 2017-07-04. Retrieved 2016-01-20.

[4] "Java Security Best Practices." *Java Security Best Practices Information Security Office,* https://security.berkeley.edu/education-awareness/java-security-best-practices.

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

[6] Fang, Yong, et al. "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology." *PloS one* 14.11 (2019): e0225196.

[7] srcML. srcml. https://www.srcml.org/, 2007. Accessed: 2021-10-20.

[8] Reimers, Nils, and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." *arXiv preprint arXiv:1908.10084* (2019).

[9] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." *arXiv preprint arXiv:2002.08155* (2020).

[10] *OpenAI API*. Available at: https://beta.openai.com/docs/guides/embeddings/what-are-embeddings (Accessed: December 5, 2022).

[11] Rabheru, Rishi, Hazim Hanif, and Sergio Maffeis. "DeepTective: Detection of PHP vulnerabilities using hybrid graph neural networks." *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021.

[12] Xu, Aidong, et al. "Vulnerability detection for source code using contextual LSTM." *2018 5th International Conference on Systems and Informatics (ICSAI)*. IEEE, 2018.

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

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

[15] Zhou, Yaqin, et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks." *Advances in neural information processing systems* 32 (2019).

[16] Sultana, Kazi Zakia, Vaibhav Anu, and Tai-Yin Chong. "Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach." *Journal of Software: Evolution and Process* 33.3 (2021): e2303.

[17] Karampatsis, Rafael-Michael, and Charles Sutton. "Scelmo: Source code embeddings from language models." *arXiv preprint arXiv:2004.13214* (2020).

[18] Caruana, R.; Niculescu-Mizil, A. (2006). *An empirical comparison of supervised learning algorithms*. Proc. 23rd International Conference on Machine Learning.

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

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

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

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

[23] L. K. Shar and H. B. K. Tan, "“Predicting common web application vulnerabilities from input validation and sanitiza- tion code patterns," in *Proceedings of the2012 27th IEEE/ACM international conference on automated software engineering*,

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

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

[26] Hanif, Hazim, et al. "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches." *Journal of Network and Computer Applications* 179 (2021): 103009.

[27] Reimers, Nils, et al. "Classification and clustering of arguments with contextualized word embeddings." *arXiv preprint arXiv:1906.09821* (2019).

[28] Lu, Zhibin, Pan Du, and Jian-Yun Nie. "VGCN-BERT: augmenting BERT with graph embedding for text classification." *European Conference on Information Retrieval*. Springer, Cham, 2020.

[29] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[30] . Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[31] K. Ramirez-Amaro, Y. Yang, and G. Cheng. A survey on semantic- based methods for the understanding of human movements. *Robotics and Autonomous Systems*, 119:31–50, 2019.