Vulnerability Classification on Source Code using Text Mining and Deep Learning Techniques

Ilias Kalouptsoglou[1,2,*], Miltiadis Siavvas[1], Apostolos Ampatzoglou[2], Dionysios Kehagias[1], and Alexander Chatzigeorgiou[2]
[1]Centre for Research and Technology Hellas, Thessaloniki, Greece
[2]University of Macedonia, Thessaloniki, Greece
iliaskaloup@iti.gr, siavvasm@iti.gr, a.ampatzoglou@uom.edu.gr, diok@iti.gr, achat@uom.edu.gr
*corresponding author

Abstract—**Nowadays, security testing is an integral part of the testing activities during the software development life-cycle. Over the years, various techniques have been proposed to identify security issues in the source code, especially vulnerabilities, which can be exploited and cause severe damages. Recently, Machine Learning techniques capable of predicting vulnerable software components have appeared, among others, enhancing the automation of the demanding process of identifying security flaws. However, there is also a need for automating the process of labeling detected vulnerabilities in vulnerability categories. Traditionally, vulnerability categorization was conducted through experts-based labeling. Later, automated methods were proposed for determining the category of reported vulnerabilities in the National Vulnerability Database based on their textual descriptions. This work examines the vulnerability classification directly from the source code during the detection phase. This way, a vulnerability detection method will be able to provide complete information and interpretation of its findings. Leveraging the advances in the field of Artificial Intelligence and Natural Language Processing, we construct and compare several multi-class classification models for categorizing vulnerable code snippets. We also discuss our findings by examining the strengths and the weaknesses of the utilized techniques. The results highlight the importance of the context-aware embeddings of the Transformer architecture as well as the significance of transfer learning from a programming language-related domain.**

Index Terms*—*security testing; vulnerability categories; machine learning; word embedding; Transformer; context-awareness

1. Introduction

Software security is considered a significant characteristic of the software development life-cycle SDLC, based on the ISO/IEC 25010 International Standard on Software Quality [1]. A major concern of the software community from the aspect of software security is the identification, characterization, and mitigation of the software vulnerabilities that reside in the source code. Those vulnerabilities are weaknesses in software systems, which can be exploited by external threats [2][3]. The utilization of security testing and especially vulnerability detection tools is considered crucial for the deployment of any software product. Software enterprises can benefit from such mechanisms by allocating their limited time and fortification efforts to potentially vulnerable segments.

As modern software systems become more complex and interconnected, there is a constantly increasing number of new identified vulnerabilities [4]. Thus, there is a need for a classification scheme to group related and similar vulnerabilities. Therefore, the Common Weakness Enumeration (CWE) system [5] was introduced by MITRE to categorize weaknesses found in software systems. Each individual CWE represents a single vulnerability type. However, CWE is an expert-based system, where the founders of a vulnerability are often different from the categorizers of it [6]. Moreover, the aforementioned manual categorization usually is a time consuming procedure. According to Spanos et al. [7], there are delays between when a vulnerability is reported, when the technical description is conducted, and when the vulnerability is characterized with a CWE and a severity score.

Recently, approaches of employing Machine Learning (ML) algorithms for automating the procedure of categorizing vulnerabilities have appeared. To enhance the manual classification carried out by the security experts, Aivatoglou et al. used ML to categorize reported vulnerabilities into CWEs utilizing the technical descriptions provided from NVD for each vulnerability [8]. Additionally, Liu et al. classified vulnerabilities gathered from cybersecurity articles and websites into the groups of code injection, access issues, buff errors, and SQL injection [9]. However, although these approaches may automate the categorization of the already reported vulnerabilities, they still need a formal and accurate vulnerability-specific technical description to be provided first. In other words, none of these methods are part of the software testing, since they do not leverage information retrieved from the analyzed software itself.

In contrast with vulnerability prediction techniques, which are commonly built based on ML models that utilize software attributes to predict vulnerable hot-spots (e.g., files, classes, methods, etc.) in a software product [10][11][12], the procedure of labeling the predicted vulnerabilities is currently disconnected from the software testing phase, raising questions about the interpretability, and hence, the reliability of the findings of the testing activities. ML-enabled vulnerability prediction has grown significantly, but it still presents important drawbacks, which stand as obstacles for its adoption in practice. For instance, although it overcomes the limitations of static analysis by producing much fewer false positives and being able to identify not only coding violations of pre-defined rules but also complex vulnerabilities [13][14], it lacks the ability of static code analyzers to present specific lines and categories of the security alerts.

Recent research endeavors in the field of software vulnerability prediction have attempted to reduce the level of granularity of their predictions, focusing on localizing as much as possible the vulnerable lines of code per component [15][16]. For this purpose, they often employ explainable Artificial Intelligence (AI) methods (e.g., Attention mechanism) in order to recognize the parts of the code that were the most important in the model's prediction of a vulnerability. However, these techniques do not provide further information about the category of the vulnerability, since they try to explain the model's decision of classifying a file or a function as vulnerable in a binary classification scheme.

The purpose of this study is to present a mechanism of classifying detected vulnerabilities, providing this way developers with valuable insights into the kind of the security issues that exist in their software (e.g., command injection, deadlock, SQL injection, etc.) during the testing phase of the SDLC when the vulnerabilities are actually identified. To this end, we conduct an extensive comparative analysis of several Natural Language Processing (NLP) techniques applied in the textual format of vulnerable source code snippets.

In particular, we examine two different text representations methods (i) Bag-of-Words (BoW) and (ii) sequences of tokens as well as four different word embedding algorithms (i) Word2vec [17], (ii) fastText [18], (iii) Bidirectional Encoder Representations from Transformers (BERT) [19], and (iv) CodeBERT [20]. We also train different kinds of ML models (e.g., Random Forest and Transformer). Furthermore, special attention is paid on the adoption of transfer learning. Last but not least, we provide insightful observations for the strengths and the weaknesses of each examined algorithm based on the results.

The rest of the paper consists of five main parts. In Section 2, we provide a summary of the state-of-the-art approaches in the field of vulnerability classification to categories and in Section 3, we provide the necessary theoretical background of the examined text mining methods. In Section 4, we provide details of the utilized dataset, the examined ML algorithms, and the overall methodology of the current study. Section 5 presents the results of our evaluation scheme and finally, Section 6 concludes the study and provides future research directions.

2. Related Work

Research studies about the characterization of security vulnerabilities focused on the technical description provided for the reported software flaws. Initially, Neuhaus and Zimmermann analyzed vulnerability reports in the CVE database, represented as BoW, utilizing an unsupervised ML algorithm (i.e., Latent Dirichlet Allocation) to find the most usual types of vulnerability.

Then, Yamamoto et al. developed a method able to calculate vulnerability scores based on the natural language description provided by CVE [21], while Wen et al. proposed an automatic vulnerability categorization framework using text mining on the vulnerability descriptions in NVD [6]. Subsequently, Aghaei et al. presented ThreatZoom, a tool capable of classifying CVEs into CWEs from CVE descriptions using Artificial Neural Networks (ANNs) [22].

Furthermore, Yosifova et al. examined the performance of several baseline ML models on predicting the vulnerability type using as features the CVE descriptions [23]. In addition, Aivatoglou et al. proposed a text analysis and ML-based method to automate the process of vulnerability classification using NVD descriptions, showcasing the high prediction accuracy of tree-based models [8].

In the vulnerability prediction-related literature, where studies utilize software attributes to classify software components as vulnerable or not, text mining techniques have demonstrated encouraging results [11][24][25][26]. However, a very limited number of studies has dealt with the objective of classifying vulnerabilities to vulnerability categories as highlighted in a systematic mapping study on software vulnerability prediction [27].

Wartschinski et al. proposed VUDENC, which is a Deep Learning (DL) based vulnerability detection model [28] considering several different vulnerability categories. Although in their study they present also results per vulnerability category, they focus on identifying which components are vulnerable, by performing binary classification. Moreover, Kong et al. dealt with the problem of identifying multi-type vulnerabilities, using graph embeddings and graph neural networks, but, although they included several CWEs in their dataset, they focus on the discrimination between vulnerable and non-vulnerable components, as well [29]. In fact, they are more interested in exploring how best to capture the diverse code representations of different types of vulnerabilities.

One of the initial attempts to classify software vulnerabilities during the security testing phase is [30], where the authors noticed the need for pinpointing types of vulnerabilities during their detection phase. They proposed the use of DL, Bidirectional Long-Short Time Memory (BiLSTM) neural network specifically, to implement a multi-class vulnerability prediction model. They also noticed that training a separate model for each type of vulnerabilities and apply all of them to every single sample of the testing data is neither a scalable nor an effective enough solution.

Next, Mamede et al. explored the capabilities of Transformer-based models on the classification of software vulnerabilities [31]. Particularly, they trained several BERT variants for multi-label vulnerability classification using a synthetic dataset of Java source code. Moreover, Mazuera–Rozo et al. examined several different source code representations and ML classifiers in both binary and multi-class vulnerability prediction showing a very high accuracy drop in both cases when using real-world data instead of synthetic ones [32].

From the above analysis, we can argue that there is a lack of fine-grained vulnerability classification mechanisms, operating in the testing phase of the SDLC. Moreover, existing methods have focused primarily on C/C++ and secondary on Java vulnerabilities. The most promising work seems to be the study of Zou et al. but their study [30] has several limitations, since it: (i) identifies solely vulnerabilities related to API/library function calls, (ii) cannot localize the identified vulnerabilities, (iii) is based on a dataset that is largely synthetic, and (iv) deals only with C/C++ code.

In the present study, we propose a method that enhances the automation of the vulnerabilities' categorization as opposed to the traditional expert-based labeling, by leveraging AI and NLP algorithms. Concisely, the contributions of the mechanism presented in this study to the relevant literature can be summarised as follows: Firstly, this mechanism achieves faster classification of vulnerabilities (i.e. from the security testing phase) as opposed to other ML-based methods that classify reported vulnerabilities based on their description provided by NVD [8][9]. Secondly, the proposed scheme provides increased confidence in the security test findings by complementing them with the categories of vulnerabilities identified. Thirdly, it categorizes vulnerable lines of code, and hence, achieves a lower level of granularity of the vulnerability classification process (i.e., code snippet level), as opposed to studies that implement multi-class vulnerability prediction by predicting which files or methods contain vulnerabilities and of what kind [30][31][32].

Moreover, we classify real-world vulnerabilities (instead of synthetic ones), which belong to categories of vulnerabilities (e.g., command injection, path disclosure, open redirect, etc.) that are considered major security issues in software engineering. Furthermore, this study performs vulnerability classification on source code written in Python, which is one of the fastest growing and most popular programming languages [33]. Finally, a discussion is conducted regarding the relationship between the results obtained and the differences as well as the technological developments in the NLP techniques considered.

3. Theoretical Background

In this section, a description of the main text mining techniques utilized in the current study is provided. In particular, details about the main concepts that we adopted from the NLP domain to perform vulnerability classification are presented. To train ML models to classify vulnerabilities based on code snippets identified as vulnerable, we first needed to represent the source code in a numerical way. For this purpose, we used common textual representation methods. In particular, we used (i) Bag-of-Words (BoW) and (ii) sequences of tokens representations.

On the one hand, BoW is the simplest NLP method that allows training ML models on text data. In the BoW approach, code snippets are collected in a "bag" that contains the words of each snippet, without considering their sequential order. This method constructs a vocabulary from the words of the entire training dataset, with each code snippet represented as a vector aligned with the vocabulary. The values of this vector are the number of occurrences of each word in the code snippet. This way, the textual data of the code snippets are transformed to tabular data, allowing the code to be analyzed through a structured numerical format.

On the other hand, in the sequences of tokens approach, each code snippet is represented as a sequence of the words included in the snippet. Therefore, the sequential order of the tokens within the code snippets is preserved, giving a benefit to ML algorithms that are capable of capturing the structural dependencies hidden among the tokens of the source code. After constructing token sequences, a word embedding method has to be applied in order to feed ML models with numerical vectors as well as to represent better the meaning of the words.

In order to vectorize the sequences of tokens, we employed various sophisticated algorithms such as Word2vec, fastText and Transformers. These algorithms are DL models capable of learning semantic and syntactic relationships among the tokens and place them at the vector space based on their similarity and their actual position in the text (i.e., source code).

Word2vec, which was initially proposed by Mikolov et al. and Google [17], is one of the most popular and widely used techniques for vectorizing source code [34][35][36]. It operates under the premise that words sharing similar contexts also share similar meanings. This method employs two primary architectures, namely the Continuous Bag-of-Words and Skip-gram [17]. Although Word2vec has proved to be efficient in a variety of NLP tasks [37], it has several drawbacks, as well. Specifically, it neither handles the out-of-vocabulary (OOV) problem nor is able to capture contextual relationships.

To the contrary, fastText, which is another efficient word embedding algorithm proposed by Bojanowski et al. and Facebook AI [18], manages to handle the OOV problem by considering sub-word information. More specifically, fastText breaks words into smaller units, such as character n-grams, and therefore, is able to handle OOV words. Similarly to Word2vec, it has both Continuous Bag-of-Words and Skip-gram architectures. However, it still has difficulty in understanding the complex semantic relationships and multiple meanings of words. Therefore, both fastText and Word2vec embeddings are called static or global embeddings as they are unique per word and do not change based on the context.

A more evolved architecture namely the Transformer, which was originally proposed by Vaswani et al. and Google [38], managed to surpass the aforementioned issues of the traditional word embeddings techniques. Specifically, the Transformer architecture, which revolutionised the NLP field, introduced the positional embeddings that can capture relative positions of the tokens in the sequences. In other words, the Transformer, through its positional embeddings, can capture contextual patterns across the whole input sequence with positional information. Therefore, each word's embedding vector is not unique but depends on the context. As opposed to the static vectors, Transformer-based word embeddings are considered contextual embeddings.

One popular variant of the Transformer architecture is the Bidirectional Encoder Representations from Transformers (BERT) model that was proposed by Google [19]. It is an encoder-only Transformer, which has been pre-trained on the task of masked language modeling (MLM) using a large corpus consisted of English Wikipedia and BookCorpus [39]. More specifically, it was trained to predict the original tokens (i.e., words) in sentences where 15 % of them were randomly masked by a special *mask* token.

In a replication study of BERT, Liu et al. [40] observed that BERT was significantly undertrained, and therefore, they proposed a robustly optimized pre-training approach, called RoBERTa. Based on the RoBERTa model, Microsoft pre-trained a model called CodeBERT [20], which is a bimodal model that has been pre-trained not only on natural language but also on six programming languages (i.e., Java, Python, Go, Ruby, PHP, and JavaScript). In particular, Feng et al. pre-trained it in pairs of function-level source code and the corresponding documentation in natural language. During pre-training, CodeBERT learnt general-purpose representations that proved to be useful in tasks such as generation of code documentation and code search based on natural language queries [20].

4. Methodology

This section describes the overall methodology that we followed in order to predict the types of software vulnerabilities. In particular, we present the dataset utilized, the strategy of constructing ML models, and the evaluation procedure. Figure 1 illustrates all the steps of our implementation: (i) data collection and preparation, (ii) model selection, (iii) model training, parameterization and prediction, and (iv) model evaluation and comparison.

4.1 Dataset

For training and evaluating the examined vulnerability classification approaches, we utilized a dataset that consists of Python source code. This dataset is provided by Wartschinski et al. [28], who used a version control system (i.e., GitHub) as a data source for collecting source code components. To create a dataset of files signed with a label that declares if they are vulnerable or not, they scanned many commit messages in GitHub projects written in Python programming language.

In particular, they searched for commits, which contain vulnerability-fixing keywords in the commit message. They gathered a large number of Python source files included in such commits. The version of each file before the vulnerability-fixing commit (i.e., parent version) is considered as vulnerable, since it contains the vulnerability that required a security patch. They also gathered the *diff* files, which contain the differences between two consecutive commits. This way, they managed to extract the specific lines which were repaired, and hence, were considered as vulnerable.

All those collected vulnerable block of lines were also characterized by a unique vulnerability category. This specific labeling was conducted based on the keywords included in the commit messages of the fixing commits. The authors of [28] chose to include keywords indicative of seven common vulnerability types, taking into account the OWASP Top 10 list [41]. Specifically, the dataset contains 4530 code blocks

TABLE I
Dataset class distribution

Vulnerability category	No. of vulnerabilities
SQL injection	1431
XSRF	976
Command injection	721
Path disclosure	481
Open redirect	442
Remote code execution	334
XSS	145

categorized as SQL injection, Cross-Site Request Forgery (XSRF), command injection, path disclosure, open redirect, remote code execution, and Cross-Site Scripting (XSS) vulnerabilities. Table I presents the distribution of the classes (i.e., vulnerability categories) in the dataset.

4.2 Study Design

In the first step, of our methodology, after retrieving the vulnerability-related data, we pre-processed the code snippets and we replaced all the numeric constants (e.g., integers, floats, etc.) and String literals with two unique identifiers, ”numId$” and ”strId$” respectively. This replacement was necessary in order to make the code snippets more generic and free from application specific constants, which could affect the performance of the produced models.

Subsequently, in the model selection phase, we represented the source code in two formats namely as BoW and sequences of tokens. The former is in numerical form as it represents the source code in a table of words with their number of occurrences as features. For the latter, we transformed each sequence to a numerical vector called embedding, comparing several embedding methods (i.e., Word2vec, fastText, BERT, and CodeBERT). For the actual implementation of these techniques, we employed the algorithms provided by Gensim[1] and Hugging Face (HF)[2] libraries.

For the cases of Word2vec and fastText (i.e., global embeddings), since we deal with a programming language-related task, we trained word embedding vectors leveraging a large code corpus that is provided by Wartschinski et al. [42]. This corpus consists of functions written in Python programming language and it contains 11.5 million lines of code in total. Then we computed the mean among the embedding vectors that correspond to the different words in each input sequence, resulting in a single vector that represents the average of all the word vectors in the sequence.

On the other hand, for the case of Transformer-based models, there was no need for training embedding vectors as they are models already pre-trained on large datasets. Therefore, we utilized the pre-trained models provided by HF. In addition, in this case, we fed the sequences of tokens to the pre-trained models in inference mode and we extracted the sentence-level embedding vectors from the last hidden state of

[1]https://radimrehurek.com/gensim/
[2]https://huggingface.co/

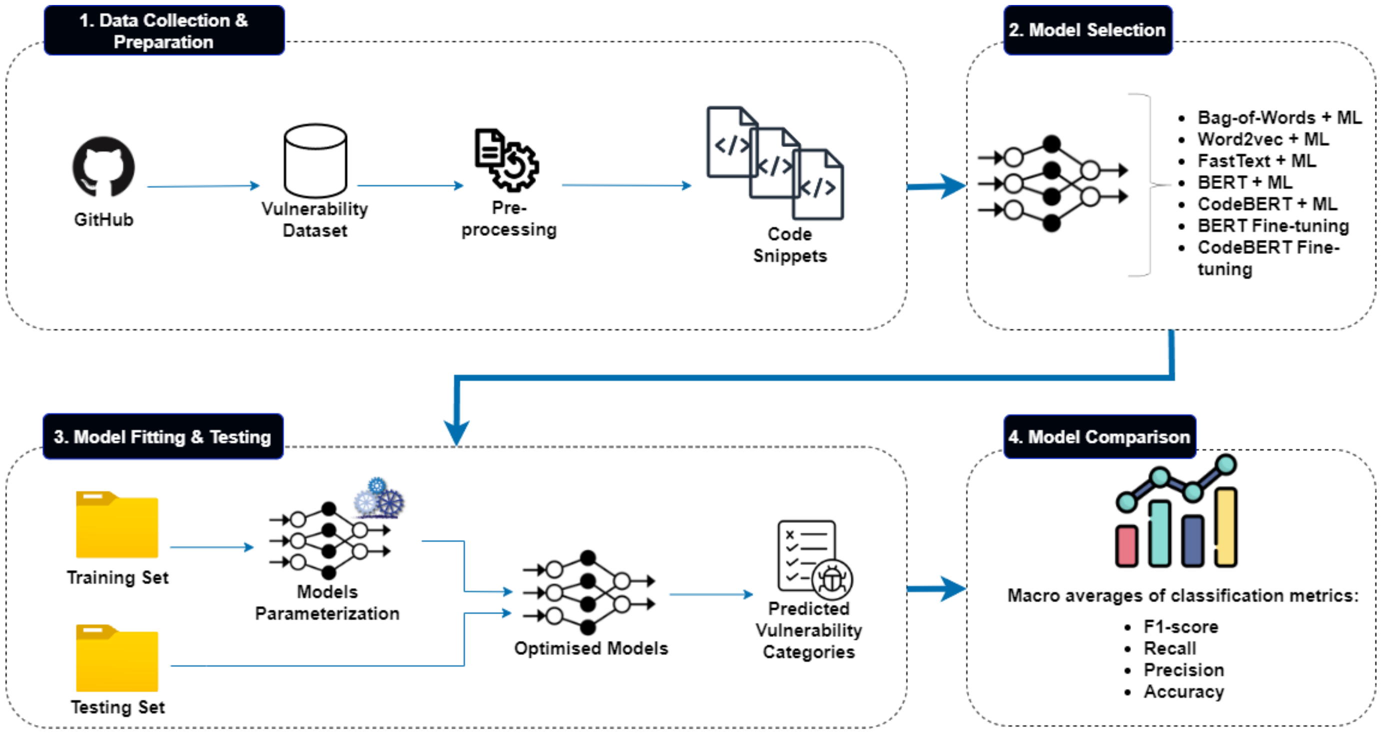

Fig. 1. Overview of the overall approach.

the Transformer. This way we gained a contextual embedding vector for each sequence of tokens given as input.

Furthermore, either the BoW or the embedding vectors of the sequences were fed to a ML model in order to perform multi-class classification to vulnerability categories. For the selection of the classifier, we examined several ML models including Decision Trees (DT), K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Random Forest (RF) classifiers. We also examined the approach of fine-tuning the pre-trained BERT and CodeBERT models to the downstream task of vulnerability category prediction.

As opposed to the embeddings extraction approach, during fine-tuning the whole model participates in the training to the downstream task. Regarding the Transformer-based models utilized (i.e., BERT and CodeBERT), we leveraged the pre-trained models that are provided by HF. More specifically, for BERT we used the *bert-base* model, which has 12 layers, 768 hidden size, 12 attention heads [38], and 110 millions (M) parameters, whereas for CodeBERT we leveraged the *codebert-base-mlm* version, which, similarly with *roberta-base*, has the BERT architecture but 125M parameters.

In step 3 of Figure 1, we trained the aforementioned models using the training set of the dataset. We conducted several experiments until ending up with the optimal hyperparameters. Then we utilized the optimized models to predict vulnerability categories for the code snippets of the testing set. Finally, in step 4, we evaluated the optimized models of all the different approaches using common classification metrics in order to identify the method which achieves the highest accuracy scores.

4.3 Evaluation Scheme

For the evaluation of the models that we trained for the task of vulnerability classification, we applied the well established technique called k-fold cross-validation [27]. During this process, the dataset is separated in k different parts, specifically in ten folds. Then, the nine folds are utilized as training set and the rest one as testing set. The training and testing is repeated ten times, each time selecting a different fold as the test set. This way, we avoid putting data bias on the developed models, assuring that the models can perform well on various parts of the dataset and not on one random split.

As regards the measurement of the performance of the DL models, we utilized common classification metrics, such as accuracy, precision, recall, and F_1-score, which are frequently utilized in the literature [8][27]. The value of these performance indicators is determined from the number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) that are produced by the models. For facilitating the comparison among the examined techniques, we pay more attention on the F_1-score, which considers both precision and recall giving equal weight to them, and therefore, is most suitable for evaluating these models. On the contrary, we do not pay so much attention on accuracy, since our dataset is imbalanced and therefore accuracy could be misleading.

We have to clarify also that we present the macro average values of the aforementioned metrics, instead of micro or weighted average. Macro averaging is probably the most straightforward averaging method as it is equal to the arithmetic mean of all the per-class scores (e.g., F_1-score), without using any class weights for the aggregation. To the contrary, weighted averaging, for instance, of F_1-score, calculates all the per-class F_1-scores but when adding them, it uses a weight depending on the number of true labels of every class, whereas micro averaging computes TP, FP, TN, and FN separately for every class and then computes the global F_1-score [43]. Since we consider all classes equally important, and therefore, we do not have to take into account the number of samples per class, we qualify the macro averaging method. Equations (1) and (2) present the mathematical formulas of the F_1-score and the macro average of F_1-score for multi-class classification respectively.

$$F_1 = \frac{2 \times precision \times recall}{precision + recall} = \frac{2 \times TP}{2 \times TP + FP + FN} \quad (1)$$

$$MacroF_1 = \frac{1}{N} \sum_{i=1}^{N} F_{1i} \quad (2)$$

, where i is the index of each class.

5. Results and Discussion

This section presents the results of our experimental analysis. The experiments took place on a parallel computing platform called CUDA that we have installed on a GeForce RTX 3060 Nvidia GPU. To enhance the reproducibility of the experiments, we will provide our scripts online [44].

Initially, an empirical comparison of several ML classifiers (i.e., Decision Trees, Support Vector Machines, KNN, and Random Forest) was conducted so as to identify the best performing one for each utilized NLP technique (i.e., BoW, Word2vec, fastText, BERT, and CodeBERT). Figure 2 provides a bar chart that shows the F_1-scores of all the examined models.

As can be seen, Decision Tree (DT) is the less accurate model regardless of the NLP method, whereas the Random Forest (RF) achieves the highest scores. The K-Nearest Neighbors (KNN) and Support Vector Machines (SVM) models are the second and third best models in all cases, with one outperforming the other in some cases and vice versa. Therefore, we can argue that RF, which is an Ensemble model which combines the output of several decision trees [45], is superior than the other examined classifiers. Hence, we qualify it as our predictor in the subsequent experiments.

Regarding the comparison among the NLP method utilized for text representation, in Table II the macro average values of the classification metrics achieved by the best performing ML model (i.e., Random Forest) are presented. We can see that all of the examined methods achieve adequate prediction performance with F_1-scores above 75% except for Word2vec, which suffers from the OOV problem and also does not consider the context of each token when representing in with a numerical vector.

To the contrary, fastText operates on character and sub-word level, and therefore, it can represent efficiently OOV tokens. However, it still overlooks the context of tokens in different sequences. This is where BERT excels, since it provides context-aware embeddings, managing to outperform Word2vec by almost 10%, but it lacks knowledge of the programming language. The BERT variant called CodeBERT is a method that encompasses all the aforementioned concepts as it is context-aware (like BERT), handles OOV problem using sub-word tokenization (similarly to fastText), and has prior knowledge of programming languages.

The findings presented in Table II show that the OOV problem is a significant one, resulting in low performance by Word2vec. Furthermore, fastText managed to surpass the context-aware BERT highlighting the importance of having domain-specific prior knowledge. Moreover, the fact that fastText and CodeBERT are very close not only to each other but also to BoW, with BoW actually to be a bit higher, suggests that the classification process may pay attention to the occurrence in the code snippets of specific tokens that are indicative of a vulnerability category and instead have difficulty in capturing syntactic patterns that reside in the source code.

Subsequently, in order to depict clearer the role of the prior knowledge in transfer learning-based vulnerability classification, we present Table III. On the one hand, Table III contains the classification scores of our approach when using pre-trained Word2vec, fastText, and BERT embeddings. These pre-trained embeddings have been trained on large corpuses of natural language and have learnt syntactics and semantics of words. Word2vec was originally pre-trained on a dataset of google news, while fastText was pre-trained on Wikipedia data and BERT on a dataset comprising English Widipedia and BookCorpus[3]. On the other hand, Table III presents the classification scores for Word2vec and fastText embeddings that we trained using a Python corpus, and also for the CodeBERT which is already pre-trained on programming languages-related data.

Based on the findings presented in Table III, it is clear that in all cases the prior knowledge of domain-specific language (i.e., programming language) is very beneficial in the task of vulnerability classification. In particular, we can see that when representing code snippets with code-aware embeddings, we succeed higher scores in all of accuracy, precision, recall, and F_1-score metrics.

Finally, we proceeded with training the whole Transformer-based models. More specifically, instead of extracting their embeddings and feeding them to the ML classifiers, we fine-tuned BERT and CodeBERT models on the downstream task of vulnerability classification. Table IV compares the fine-tuning and embeddings extraction approaches of employing Large Language Models (LLMs).

By inspecting Table IV, the fine-tuning approach seems to the optimal one, at least for this specific objective. The

[3]https://en.wikipedia.org/wiki/BookCorpus

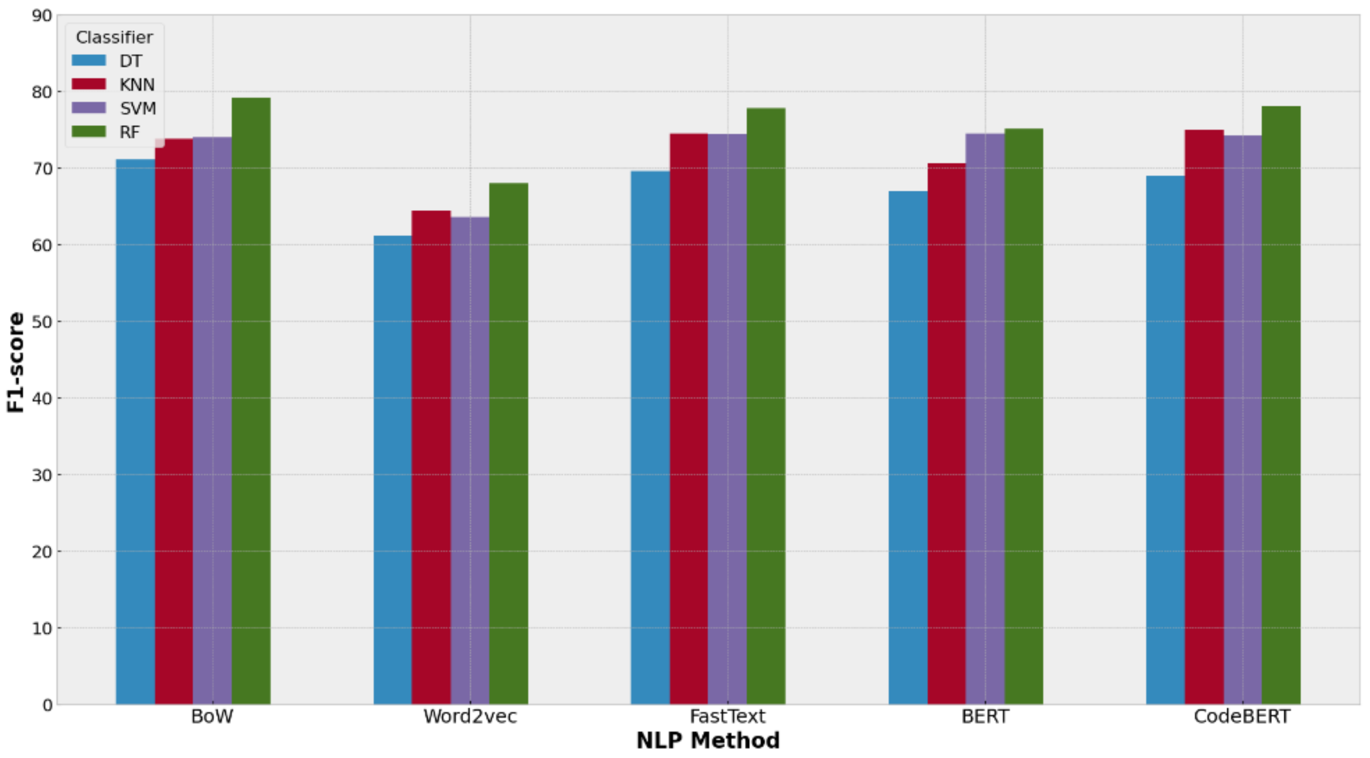

Fig. 2. Comparison of different ML classification models per NLP approach.

TABLE II
EVALUATION RESULTS OF THE RANDOM FOREST CLASSIFIER PER TEXT VECTORIZING METHOD

Vectorizing Method	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F_1-score (%)**
Bag-of-Words	81.9	82.3	77.2	79.1
Word2vec	71.6	76.2	64.3	68.0
fastText	80.2	84.0	73.9	77.7
BERT	76.9	86.6	69.4	75.1
CodeBERT	80.7	87.6	72.9	78.0

TABLE III
CLASSIFICATION PERFORMANCE OF NLP MODELS WITH PRIOR KNOWLEDGE OF NATURAL LANGUAGE VERSUS PROGRAMMING LANGUAGE

Vectorizing Method	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F_1-score (%)**
pre-trained Word2vec	68.1	73.2	59.9	63.8
re-trained Word2vec	71.6	76.2	64.3	68.0
pre-trained fastText	74.9	78.0	68.0	71.5
re-trained fastText	80.2	84.0	73.9	77.7
pre-trained BERT	76.9	86.6	69.4	75.1
pre-trained CodeBERT	80.7	87.6	72.9	78.0

results showcase (i) the benefit of performing fine-tuning over feature extraction in both BERT and CodeBERT cases, and (ii) the superiority of the fine-tuned CodeBERT over the fine-tuned BERT. More specifically, when utilizing BERT embeddings with a RF classifier, the accuracy achieved is 76.9%, with corresponding precision, recall, and F_1-score of 86.6%, 69.4%, and 75.1%, respectively. However, fine-tuning BERT significantly improves results, with a significant boost in F_1-score from 75.1% to 82.5%. This improvement suggests that fine-tuning enables BERT to adapt better to the specific nuances of the downstream task, leading to enhanced predictive performance.

In addition, fine-tuned CodeBERT demonstrates higher scores than fine-tuned BERT, confirming the findings of Table III regarding the effectiveness of CodeBERT in capturing code-related features. Moreover, fine-tuning CodeBERT results in a substantial performance gain compared to the features (i.e., embeddings) extraction approach, achieving an F_1-score of 85.5%, which is by far the highest F_1-score reported in this study. This improvement highlights the importance of

TABLE IV
COMPARISON OF EMBEDDINGS EXTRACTION AND FINE-TUNING OF TRANSFORMER MODELS APPROACHES

Vectorizing Method	Accuracy (%)	Precision (%)	Recall (%)	F_1-score (%)
BERT + RF	76.9	86.6	69.4	75.1
BERT fine-tuning	84.5	82.4	82.7	82.5
CodeBERT + RF	80.7	87.6	72.9	78.0
CodeBERT fine-tuning	87.4	86.3	85.2	85.5

fine-tuning pre-trained models, pointing the capability of the Transformer architecture to capture long-term dependencies that RF cannot. Interestingly, although fine-tuning CodeBERT enhances accuracy and recall, it presents a slight decrease in precision when compared to its feature extraction alternative, indicating a likely trade-off among recall and precision. All things considered, these results highlight the importance of fine-tuning Transformer-based LLMs in order to maximize their performance in downstream tasks such as vulnerability classification, especially when working with domain-specific data (i.e., source code).

For reasons of clarity and completeness, at this point we provide the detailed results of the best performing vulnerability classification approach (i.e., fine-tuning CodeBERT). Table V showcases the precision, recall, and F_1-score of CodeBERT for each of the seven vulnerability categories.

TABLE V
DETAILED RESULTS OF THE BEST PERFORMING CODEBERT MODEL

Category	Precision (%)	Recall (%)	F_1-score (%)
SQL injection	90	90	90
XSRF	87	92	90
Open redirect	79	70	75
XSS	92	80	86
Remote code execution	78	85	81
Command injection	95	86	91
Path disclosure	82	94	87

6. CONCLUSION AND FUTURE WORK

In this paper, our purpose was to propose a different approach of classifying security vulnerabilities to vulnerability categories such as path disclosure, command injection, etc. Primarily, we focused on the categorization of detected vulnerabilities from the testing phase of the SDLC as opposed to traditional techniques, which classify reported vulnerabilities based on descriptions provided by NVD. To this end, we leveraged several NLP text representation and text classification techniques adopting a multi-class classification procedure.

During the conducted investigation, BoW, Word2vec, fastText, BERT, and CodeBERT models were trained to represent the vulnerable code snippets. Several ML models were also compared in order to choose the best classification model that is fed with the numerical (e.g., embedding) representation of the code snippets. We also proceeded with fine-tuning BERT and CodeBERT to leverage the Transformer architecture. The findings demonstrate the superiority of CodeBERT model, which is the one that handles the OOV issue using subtokenization, has domain-specific prior knowledge, and also has contextual understanding. The results show important benefit from fine-tuning over extracting embeddings approach, as well.

Several directions for future work can be followed. For instance, an interesting analysis could include the application of explainable AI techniques to reveal which parts of the code snippets were the most influential in the models' category predictions. Additionally, we aim at implementing a complete working prototype that will be able to parse the source code of software projects, identify vulnerable components, detect the specific code snippets that contain vulnerabilities, and then classify them to vulnerability categories.

ACKNOWLEDGMENT

Work reported in this paper has received funding from the European Union's Horizon Europe Research and Innovation Program through the DOSS project, Grant Number 101120270.

REFERENCES

[1] ISO/IEC, *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. ISO/IEC, 2011.

[2] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garcerán, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2019.

[3] ISO/IEC, *ISO/IEC 27000:2018 - Information technology — Security techniques — Information security management systems*. ISO/IEC, 2005.

[4] "Cyber Crime & Security," https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures, Accessed: 2024-03-15.

[5] "Common Weakness Enumeration," https://cwe.mitre.org/, Accessed: 2024-03-15.

[6] T. Wen, Y. Zhang, Q. Wu, and G. Yang, "Asvc: An automatic security vulnerability categorization framework based on novel features of vulnerability data." *J. Commun.*, vol. 10, no. 2, pp. 107–116, 2015.

[7] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems and Software*, vol. 146, pp. 152–166, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S01641 21218302061

[8] G. Aivatoglou, M. Anastasiadis, G. Spanos, A. Voulgaridis, K. Votis, and D. Tzovaras, "A tree-based machine learning methodology to automatically classify software vulnerabilities," in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021, pp. 312–317.

[9] C. Liu, J. Li, and X. Chen, "Network vulnerability analysis using text mining," in *Intelligent Information and Database Systems: 4th Asian Conference, ACIIDS 2012, Kaohsiung, Taiwan, March 19-21, 2012, Proceedings, Part II 4*. Springer, 2012, pp. 274–283.

[10] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[11] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 2014, pp. 23–33.

[12] I. Kalouptsoglou, M. Siavvas, D. Tsoukalas, and D. Kehagias, "Cross-project vulnerability prediction based on software metrics and deep learning," in *International Conference on Computational Science and Its Applications*. Springer, 2020, pp. 877–893.

[13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[14] M. Siavvas, I. Kalouptsoglou, D. Tsoukalas, and D. Kehagias, "A self-adaptive approach for assessing the criticality of security-related static analysis alerts," in *Computational Science and Its Applications–ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part VII 21*. Springer, 2021, pp. 289–305.

[15] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[16] F. Yang, F. Zhong, G. Zeng, P. Xiao, and W. Zheng, "Lineflowdp: A deep learning-based two-phase approach for line-level defect prediction," *Empirical Software Engineering*, vol. 29, no. 2, pp. 1–49, 2024.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[18] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[20] 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.

[21] Y. Yamamoto, D. Miyamoto, and M. Nakayama, "Text-mining approach for estimating vulnerability score," in *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2015, pp. 67–73.

[22] E. Aghaei, W. Shadid, and E. Al-Shaer, "Threatzoom: Hierarchical neural network for cves to cwes classification," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2020, pp. 23–41.

[23] V. Yosifova, A. Tasheva, and R. Trifonov, "Predicting vulnerability type in common vulnerabilities and exposures (cve) database with machine learning classifiers," in *2021 12th National Conference with International Participation (ELECTRONICA)*, 2021, pp. 1–6.

[24] Y. Pang, X. Xue, and H. Wang, "Predicting vulnerable software components through deep neural network," in *Proceedings of the 2017 International Conference on Deep Learning Technologies*, 2017, pp. 6–10.

[25] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, "Challenging machine learning algorithms in predicting vulnerable javascript functions," in *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 2019, pp. 8–14.

[26] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and A. Ampatzoglou, "Examining the capacity of text mining and software metrics in vulnerability prediction," *Entropy*, vol. 24, no. 5, 2022. [Online]. Available: https://www.mdpi.com/1099-4300/24/5/651

[27] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "Software vulnerability prediction: A systematic mapping study," *Information and Software Technology*, p. 107303, 2023.

[28] 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*, p. 106809, 2022.

[29] L. Kong, S. Luo, L. Pan, Z. Wu, and X. Li, "A multi-type vulnerability detection framework with parallel perspective fusion and hierarchical feature enhancement," *Computers & Security*, p. 103787, 2024.

[30] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[31] C. Mamede, E. Pinconschi, R. Abreu, and J. Campos, "Exploring transformers for multi-label classification of java vulnerabilities," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 43–52.

[32] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-Vásquez, and G. Bavota, "Shallow or deep? an empirical study on detecting vulnerabilities using deep learning," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 276–287.

[33] "Top programming languages that will rule in 2022," https://fireart.studio/blog/top-programming-languages-that-will-rule-in-2021/, accessed: 2024-03-20.

[34] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and A. Ampatzoglou, "An empirical evaluation of the usefulness of word embedding techniques in deep learning-based vulnerability prediction," in *EuroCybersec2021,Lecture Notes in Communications in Computer and Information Science*, 10 2021.

[35] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[36] 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," *arXiv preprint arXiv:1801.01681*, 2018.

[37] L. Wolf, Y. Hanani, K. Bar, and N. Dershowitz, "Joint word2vec networks for bilingual semantic representations." *Int. J. Comput. Linguistics Appl.*, vol. 5, no. 1, pp. 27–42, 2014.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[39] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[40] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[41] "Explore the world of cyber security," https://owasp.org/, accessed: 2024-03-20.

[42] L. Wartschinski, "Vudenc - python corpus for word2vec," Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3559480

[43] "Understanding Micro, Macro, and Weighted Averages," https://iamirmasoud.com/2022/06/19/understanding-micro-macro-and-weighted-averages-for-scikit-learn-metrics-in-multi-class-classification-with-example/, accessed: 2024-02-10.

[44] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "Software Vulnerability Classification using Text Mining and Deep Learning Techniques," https://sites.google.com/view/vulnerabilityclassification/, Accessed: 2024-04-01.

[45] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.