An empirical comparison of Transformer-based models in Vulnerability Prediction

Conference Paper · May 2023

CITATIONS
0

READS
227

5 authors, including:

Ilias Kalouptsoglou
The Centre for Research and Technology, Hellas
12 PUBLICATIONS **36** CITATIONS
SEE PROFILE

Miltiadis Siavvas
Imperial College London
46 PUBLICATIONS **406** CITATIONS
SEE PROFILE

Apostolos Ampatzoglou
University of Macedonia
160 PUBLICATIONS **2,286** CITATIONS
SEE PROFILE

Dionysios Kehagias
The Centre for Research and Technology, Hellas
131 PUBLICATIONS **1,010** CITATIONS
SEE PROFILE

An empirical comparison of Transformer-based models in Vulnerability Prediction

Ilias Kalouptsoglou[1,2][0000−0002−5118−2508], Miltiadis Siavvas[1][0000−0002−3251−8723], Apostolos Ampatzoglou[2][0000−0002−5764−7302], Dionysios Kehagias[1][0000−0002−6912−3493], and Alexander Chatzigeorgiou[2][0000−0002−5381−8418]

[1] Centre for Research and Technology Hellas/Information Technologies Institute, Thessaloniki, Greece {iliaskaloup,siavvasm,diok}@iti.gr
[2] University of Macedonia/Department of Applied Informatics, Thessaloniki, Greece {iliaskaloup,a.ampatzoglou,achat}@uom.edu.gr

Abstract. Nowadays, the increasing flourishing of the language models provides a new direction not only for the task of text generation, which is their actual goal, but also for dealing with downstream tasks, such as text classification. Vulnerability prediction is a task that has been heavily connected with text mining techniques, since many studies in the related literature utilize source code as text, and identify vulnerable patterns through deep learning algorithms. Recently, there have appeared studies which employ transfer learning techniques in order to benefit from the large prior knowledge of the pre-trained models in order to perform accurate vulnerability prediction. In this study, several large language models, which are based on the Transformer architecture, are compared in their ability to predict vulnerable software components by receiving as input source code in textual format. More specifically, we fine-tune BERT, the several BERT variants, GPT-2, and BART models in the task of vulnerability prediction, we evaluate their performance, and we conduct a comparative study between them in order to identify which of these models are the most suitable and accurate ones on vulnerability prediction.

Keywords: Vulnerability prediction · Text mining · Large language models · Transformer-based models · BERT

1 Introduction

Nowadays, software security is considered a significant characteristic of the software development life-cycle, 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 and the 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]. Their identification is considered crucial of the deployment of any software product. Vulnerability Prediction (VP) is a technique capable of predicting hot-spots

(e.g., files, classes, methods, etc.) of a software product that contain vulnerabilities. Software enterprises can benefit from such a mechanism by allocating their limited time and testing effort to potentially vulnerable segments.

Vulnerability prediction is commonly performed through machine learning algorithms utilizing software attributes as input. The two largest categories of Vulnerability Prediction Models (VPMs) are based either on code metrics or text mining. Software metrics-based techniques utilize metrics like complexity and coupling that can be retrieved using static code analyzers and can be used as features to train a machine learning model [3–6]. On the other hand, text mining approaches receive the text of the source code as input and through machine learning models, they identify vulnerable patterns into the source code [7–11]. There are studies that have compared the performance of software metrics-based and text mining-based models. They demonstrated that the advanced text mining approaches provide more accurate predictions than software metrics-based studies [12, 13].

Initially, text mining-based studies in VP were employing the Bag of Words (BoW) technique, which represents the source code as a set of words. In BoW, each word is accompanied with the number or the frequency of its appearances in the analyzed software component and it is considered as a feature for a machine learning classifier [7, 12]. Then researchers started to represent the source code as tokens sequences. In this approach, deep learning models are utilized in order to recognize vulnerable patterns in the sequences of source code tokens [14–16]. To enhance the predictability of the models, the tokens are transformed to numerical vectors by using word embedding methods such as word2vec [17] and fastText [18]. Such techniques assist the models to capture semantic and syntactic relations between the tokens. Later, researchers represented the source code with text-rich graphs (e.g., Abstract Syntax Trees, Code Properties Graphs, etc.) [19, 20]. More specifically, they extracted graphical representations of the code, they employed Graphical Neural Networks (GNNs) to generate embeddings of these representations and then, through deep learning models, they classified the components as vulnerable or not.

Recent text mining-based studies adopted Natural Language Processing (NLP) techniques to perform vulnerability prediction, taking advantage of the innovative Transformer architecture [21]. In particular, they applied transfer learning utilizing Transformer-based [21] large language models (LLMs) [22, 23], which are pre-trained on large generic datasets for their primary tasks, such as next word prediction or prediction of masked tokens [21, 24]. These models have managed to learn syntax and semantics of thousands of tokens in several different contexts. Hence, in the VP field, researchers can fine-tune them to adjust their deep knowledge of natural language to the downstream task of classifying software components as vulnerable or not.

The objective of our study is to identify the optimal choice among a variety of pre-trained NLP models in the downstream task of vulnerability prediction showcasing any differences in their performance. In this way, our study aims at assisting researchers in their future endeavors to use transfer learning for

vulnerability prediction, providing them indications of which of all the models found in the related literature are the most appropriate for this specific task. For this purpose, we fine-tune several large pre-trained NLP models on a labeled vulnerability-related dataset. More specifically, in this paper, we train, evaluate and compare the following models, which are all based on the Transformer architecture [21]:

- Bidirectional Encoder Representations from Transformers (BERT) [25]
- Robustly optimized BERT approach (RoBERTa) [26]
- A distilled version of BERT called DistilBERT [27]
- A Lite BERT (ALBERT) architecture [28]
- A bimodal (i.e., pre-trained on pairs of programming and natural language samples) version of BERT called CodeBERT [29]
- Bidirectional Auto-Regressive Transformers (BART) [30]
- Generative Pre-trained Transformer (GPT) [24, 31]

The rest of the paper consists of the following parts: In Section 2 we provide a summary of the state-of-the-art approaches in the VP field using text mining. In Section 3, we provide details for the examined models and we describe thoroughly the methodology that we follow to conduct the current study, while in Section 4, we present the results of our evaluation scheme. Finally, Section 5 concludes the study and provides future research directions.

2 Related Work

In the related literature, vulnerability prediction using text mining has shown encouraging results [7, 12, 19, 32]. Early research efforts paid attention on the BoW technique [7, 12]. Current efforts focus on identifying vulnerabilities by extracting more meaningful patterns from the source code rather than the frequency of the tokens. Commonly, these studies train Deep Learning (DL) models, such as the Recurrent Neural Networks (RNNs), which are suitable for receiving large sequences. In this approach, they feed the DL models with the analyzed software components, each of which constitutes a sequence of tokens [14, 19]. The challenge of this task is to encode the syntactic and semantic patterns that reside in the source code. The technique of representing the tokens as real-valued vectors, which encode the meaning of the tokens, can contribute to this direction significantly.

Towards this direction, the authors in [14, 33] employed the word2vec [34] tool to create embedding vectors for the source code tokens, whereas Zhou et al. [19] used the pre-trained word2vec vectors. Fang et al. [35] introduced the fastEmbed model, which is based on the fastText embeddings. They recognized essential textual characteristics that are pertaining to vulnerabilities and they created a model for assessing the exploitability of the vulnerabilities on unbalanced datasets. In [36] the authors compared the BoW with other complex code representations, which were automatically learned by the embedding layer of DL models. Kalouptsoglou et al. compared the performance of word2vec and

fastText word embeddings in enhancing the predictability of DL models in VP [16]. They also examined the training of the Embedding layer, which contains the embedding vectors, during the training of the rest layers of the DL model. Their findings revealed that the use of a Convolutional Neural Network (CNN), along with embeddings that have been produced by word2vec, succeeded the highest accuracy.

Bagheri et al., conducted a comparison of different Python source code representation methods for VP [22]. They investigated the efficiency of word2vec, fastText, and BERT embedding vectors for code representation. At every case, they used a Long Short-Term Memory (LSTM) model to classify the embedded software components as vulnerable or not. Their findings suggested that all these three techniques are suitable for representing source code for the task of VP, but the BERT-based embedding method seemed to be the most promising one. Yuan et al. [37] compared a CodeBERT-based embedding method with word2vec, fastText, and GloVe [38] showing that the former outperforms the latter in the task of vulnerability prediction.

Kim et al. [39] proposed a model called VulDeBERT, which is the BERT pretrained model fine-tuned on the downstream task of predicting vulnerabilities. VulDeBERT succeeded a significantly better performance than the state-of-the-art study of VulDeePecker [14]. In [23], Ziems et al. examined how transferring knowledge from English language to raw computer code written in C/C++ can enhance the effort of performing vulnerability prediction. Their results indicated that their BERT-based model outperformed the LSTM model, which was traditionally the state-of-the-art method for learning sequences of text tokens. Hanif et al. proposed the VulBERTa model [40], by pre-training a RoBERTa model on real-world code data from open-source C/C++ projects and then fine-tuning it on vulnerability-related data. They compared VulBERTa with several state-of-the-art models showcasing its efficiency.

Fu et al. proposed LineVul in an effort to predict vulnerabilities in a low level of granularity and specifically the line-level [41]. LineVul is a Transformer-based model, which, based on the experimental results of the study, is more accurate than the existing line-level prediction approaches. Steenhoek et al. reproduced and compared 9 DL-based VP models, including in the comparison some Transformer-based ones, which were found to be promising and need further investigation [42]. Coimbra et al. in their study [43], presented an evaluation of the Code2vec [44] model in contrast with simple Transformer-based methods such as the RoBERTa. Through this study, they compared the graphical code representation in the form of Abstract Syntax Tree, which is included in the Code2vec model, with the pure textual representation of the source code. Their findings highlight that the Code2vec model applied for vulnerability prediction succeeded comparable results to simple Transformer-based models managing to maintain much lower computational requirements.

From the above analysis, we can argue that the Transformer-based LLMs, especially the BERT-based ones, have attracted recently the interest of the research community in the VP field. The transferring of natural language knowl-

edge to source code processing is considered as a promising solution for the purpose of vulnerability prediction. Moreover, it seems that LLMs are capable of learning complex patterns relative to vulnerabilities that reside in the source code. However, researchers have utilized many variants of the Transformer architecture for the downstream task of VP and, from their findings, it is not clear if some of the variants are more suitable for this task than others. In the vast majority of the existing research works, the predictive performance of the pre-trained Transformer-based models was compared to the performance of basic state-of-the-art text mining-based VPMs.

In the present study, in contrast with the studies in the existing literature, we do not aim at proposing a model different or better than state-of-the-art models, but we proceed with an empirical comparison of several pre-trained models in the downstream task of VP, in an attempt to identify the optimal, if any, model. For this purpose, we fine-tune BERT, GPT, BART, and several BERT variants using a vulnerability-related dataset from real world software components written in Python programming language. This way we examine the efficiency and the accuracy of those techniques in VP and hence, we provide implications to researchers for which models to focus on during their future efforts. The strength of our study lies in the fact that we examine and fine-tune many Transformer variants, including in our analysis both models from the same architecture (i.e., BERT variants) and different ones (i.e., BERT, GPT, and BART), and we also follow the same procedure for all of them in order to extract fair conclusions. In particular, the architecture as well as the pre-trained weights of all the examined models are retrieved from the same provider (i.e., Hugging Face). We also follow the same evaluation scheme using the same evaluation metrics for all the cases to ensure that we perform a fair and unbiased comparison. We employ the same pre-processing procedure, as well. Therefore, our study can provide useful implications to researchers on which models they need to pay more attention in the future. Moreover, we provide replication material in order to enhance the reproducibility of our study [45].

3 Study Design

This section describes the overall methodology that we followed in order to fine-tune several pre-trained models to perform vulnerability prediction (VP). We trained GPT-2, BART, and different variants of BERT using a real-world dataset retrieved by commits on GitHub projects.

3.1 Dataset

For fine-tuning and evaluating the examined models, we used a dataset which consists of source code files written in Python programming language. This dataset is an extension of the dataset presented by Bagheri et al. [22], who used a version control system as a data source for collecting source code components. Specifically, they used GitHub since it has a high number of software

projects. To create a labeled dataset, i.e., a dataset of files signed with a label that declares if they are vulnerable or not, they scanned the commit messages in Python GitHub projects. 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 patch, whereas the version of the file in the vulnerability-fixing commit is considered as non-vulnerable. However, in their study, Bagheri et al. [22] utilized only the fragment of the diff file, which contains the difference between the vulnerable and the fixed version, and they proposed models to separate the "bad" and the "good" parts of a file. In the current study, we extend their dataset by collecting all the fixed versions from GitHub, which are actually the vulnerability-fixing commits of the files reported in their dataset. Hence, we can construct models to perform vulnerability prediction in file-level of granularity. Overall, the extended dataset contains 4,184 Python files, 3,186 of which are considered as vulnerable and 998 are considered as neutral (i.e., non-vulnerable).

Before proceeding with the process of training the vulnerability prediction models (VPMs), we applied a series of pre-processing steps so as to transform the dataset in the form of sequences of tokens. Initially, we removed all the comments and the commands which import external libraries or other files. Subsequently, 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 sequences of tokens more generic and free from application specific constants, which could affect the performance of the produced models. We also dropped all the empty lines, and finally, we converted the source code of each file into a list of tokens retaining the order in which the tokens appear in the file. The tokenized form of the dataset is provided online for replication purposes [45].

3.2 Strategy

This section describes the large pre-trained models that we have included in our study as well as the methodology that we have followed in order to fine-tune these models to perform vulnerability prediction (VP). All the examined models, which are based on the Transformer architecture, have been pre-trained on a large corpus of textual data for a specific task in an unsupervised manner.

Models Characteristics To begin with, the BERT model presented by Google[3] is pre-trained on the Masked Language Model (MLM) objective. Before being fed into the neural network, 15 % of each sequence of tokens is replaced with a masked token. Then the model aims at predicting the original value of the masked tokens, based on the rest non-masked tokens. This way, BERT learns a bidirectional representation of the sentences [25]. In a replication study of BERT,

[3] https://about.google/

Liu et al. [26] observed that BERT was significantly undertrained, and therefore, they proposed a robustly optimized pre-training approach, called RoBERTa. Later, Sanh et al. presented the DistilBERT [27] in an attempt to reduce the size of the BERT model significantly and making it faster, while retaining the 97 % of its language understanding. Furthermore, Lan et al. proposed a BERT variant called ALBERT in order to improve BERT performance by reducing its memory consumption and increasing its training speed [28].

A RoBERTa-based model called CodeBERT, which was presented by Microsoft[4], is the only one of the examined BERT variants that is not pre-trained solely on the English natural language, but on natural language and programming language pairs from 6 programming languages (i.e., Python, Java, JavaScript, PHP, Ruby, Go). In particular, it has been trained on bimodal data that include function-level natural language documentations and source code. During the pre-training phase, CodeBERT learns general-purpose representations, which can support applications that need both natural and programming languages, such as natural language code search and code documentation generation [29].

A similar but different to BERT Transformer-based architecture is the Generative pre-trained transformer (GPT) [24] that was developed by OpenAI[5]. GPT is not pre-trained on the MLM objective but on predicting the next word in a sentence based on the context provided by the preceding words. It is more suitable for text generation tasks such as questioning-answering and content creation, but it can be fine-tuned for several NLP purposes including text classification, as well. In this study, we utilize the GPT-2 version of the GPT, which is the latest open-source version. It has no major architectural differences in contrast with the first GPT version, but it is much larger in terms of trainable parameters, and also it has been trained on a much larger dataset. Hence, it is considered to have a better language understanding.

Facebook AI[6] has released its own Transformer pre-trained language model named as BART. BART is a sequence-to-sequence pre-trained model, which is particularly effective in natural language generation tasks, but it can be fine-tuned in a large variety of NLP tasks including also text classification. To pre-train BART, some text is first corrupted using a noise function, and then a model learns to recreate the original text [30].

For all the aforementioned models, we utilize the implementations that are provided by Hugging Face[7] (HF). More specifically, for BERT we use the *bert-base* model, which has 12 layers, 768 hidden size, 12 attention heads [21], and 110 millions (M) parameters. For RoBERTa we use the *roberta-base* model that has the BERT architecture but 125M parameters, while for DistilBERT, which is a lighter version of BERT, we use the *distilbert-base* model that contains 6 layers, 768 hidden nodes, 12 attention heads, and 66M parameters. Hugging Face provides also the *albert-base-v2* model for ALBERT, containing 11M parameters.

[4] https://github.com/microsoft/CodeBERT
[5] https://openai.com/
[6] https://ai.facebook.com/
[7] https://huggingface.co/

For CodeBERT, we choose the *codebert-base-mlm* version, which has the same architecture with *roberta-base*. Regarding GPT-2, we use the *gpt2* model with 12 layers, 768 hidden size, 12 attention heads, and 117M parameters. Finally, for BART that employs both the encoder and the decoder parts of the Transformer architecture, we utilize the *bart-base* implementation, which has nearly 140M trainable weights. It also contains 12 layers for both the encoder and the decoder, 768 hidden size, 12 encoder attention heads, and 12 decoder attention heads. All the aforementioned statistics are summarized in Table 1.

Table 1. The characteristics of the pre-trained models considered in the study

Model	Version	Layers	Hidden Size	Attention heads	Parameters
BERT	*bert-base*	12	768	12	110M
RoBERTa	*roberta-base*	12	768	12	125M
DistilBERT	*distilbert-base*	6	768	12	66M
ALBERT	*albert-base-v2*	12	768	12	11M
CodeBERT	*codebert-base-mlm*	12	768	12	125M
GPT-2	*gpt2*	12	768	12	117M
BART	*bart-base*	12	768	24	140M

Approach For the construction of the above-mentioned models, we followed an approached that is illustrated in Figure 1. As can be seen in Figure 1, the process includes three-phases: (1) pre-training, (2) fine-tuning, and (3) execution. The pre-training has been implemented by each model provider. During pre-training, a large textual dataset is tokenized in a format suitable for each Transformer-based model. Then it is fed to the model, which learns a specific objective (e.g., masked word prediction, next word predicton, etc.) in an unsupervised manner. This way, the trainable weights of the model have been trained to understand natural language and there have been produced context aware word embedding vectors, which encode the context of the words. In this study, we receive pre-trained models and we implement the fine-tuning and the execution steps.

During fine-tuning, the training dataset described in Section 3.1 is utilized. Each Python file with source code is pre-processed (see Section 3.1) and is tokenized. For the tokenization of the data, we use the tokenizer that corresponds to each model and is provided by Hugging Face. After applying such a tokenizer, the dataset is being transformed in a form that contains only words included in each model's vocabulary as well as some special tokens. For instance, when using BERT for sequence classification, we need a special classification token (i.e., [CLS]) to be placed in the beginning of each sequence, and another special token (i.e., [SEP]) to separate the sequences. Subsequently, the tokenized sequences along with their corresponding labels are going to be fed to our classifier that consists of the already trained Transformer-based model and a newly added classification layer. This way, we train, in a supervised manner, the classification layer to separate the vulnerable from the non-vulnerable sequences.

Simultaneously, the weights of the Transformer are also updated in order to be adjusted to the needs of the specific classification task. During fine-tuning, the hyperparameters that need to be determined are (1) the number of epochs, (2) the learning rate, (3) the optimizer, (4) the loss function, and (5) the max length of the sequences. For the selection of the first four, we applied several values to each one in order to succeed optimal efficiency in the validation data (see Section 3.3), while for the max length we used the maximum length that can be processed by these Transformer-based models (i.e., length equal to 512). Table 2 presents the selected values of the hyperparameters. For all the examined models we ended up with the same hyperparameters' values except for GPT-2.

Table 2. Hyperparameters of the Transformer-based models

Model	Epochs	Learning rate	Optimizer	Loss function
BART and BERT variants	6	0.00005	Adam	Cross-Entropy
GPT-2	4	0.00002	Adam	Cross-Entropy

Finally, in the execution phase, after fine-tuning the optimal Transformer-based classifier for the objective of VP, we can use it as our vulnerability predictor in order to assess new Python files and classify them as vulnerable or not.

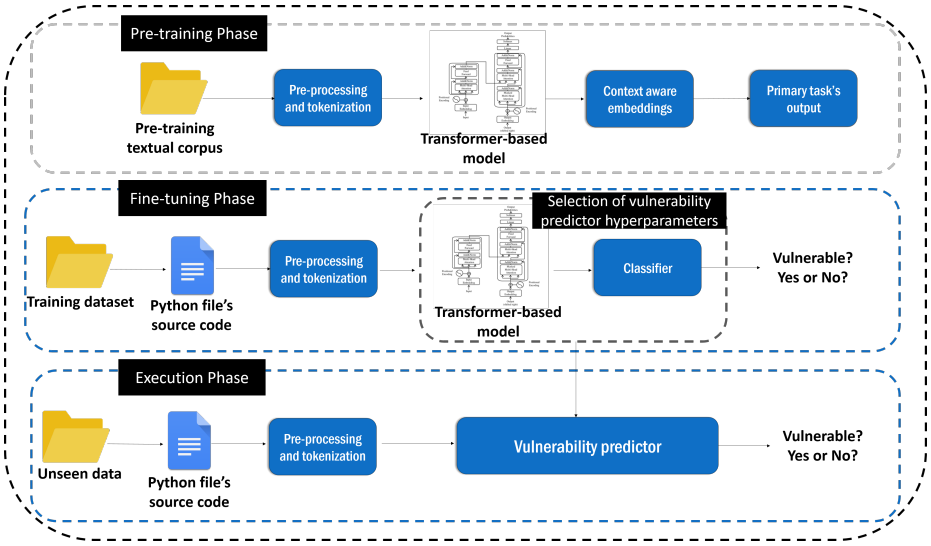

Fig. 1. An overview of the approach followed for fine-tuning Transformer-based pre-trained models for vulnerability prediction.

3.3 Evaluation Scheme

For the evaluation of the models that we fine-tuned for the task of VP, we separated the dataset on three sets: (1) training, (2) validation, and (3) testing sets. In this way, we can use the validation set for the selection of the optimal models' hyperparameters. Then, after the validation step, we train the selected model on the whole training set that consists of both training and validation sets, and subsequently, we proceed with evaluating the model's predictive power using the testing set (i.e., unseen data). This way, the process of identifying the optimal hyperparameters, and therefore, the best models, is not affected by the testing set. Hence, we avoid putting data bias on the developed models.

Regarding the quantitative measurement of the predictive performance of deep learning classification models, a number of evaluation metrics are frequently utilized in the literature. The number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) that are produced by the models is commonly used to determine these performance indicators. Regarding the evaluation of the VPMs, we put a particular emphasis on the recall of the created models since the higher the recall, the more actual vulnerabilities the model predicts. However, precision is also crucial since it shows how many FP were generated by the models and is associated to the time and effort needed to review the outcome of the models. As a result, a score which considers both precision and recall, such as F_1-score and F_2-score, is most suitable for evaluating these models. Among these two F-scores, we choose as the main evaluation metric the F_2-score, since it gives a little more weight to recall than to precision, whereas the F_1-score gives equal weight to them. The mathematical formula of F_2-score is given below:

$$F_2 = \frac{5 \times precision \times recall}{4 \times precision + recall} \tag{1}$$

4 Results

This section presents the results of our comparative study among the Transformer-based models, after fine-tuning them on the downstream task of VP. All the reported results are the outcome of applying the fine-tuned models on the testing set of the dataset (see Section 3.3). The experiments took place on a parallel computing platform called CUDA[8] that we have installed on a GeForce RTX 3060 Nvidia GPU. To enhance the reproducibility of the experiments, we provide our scripts online [45].

In Table 3, we present the outcome of our evaluation scheme. More specifically, Table 3 provides the values of accuracy, precision, recall, F_1-score, and F_2-score, with the latter to be considered as the most critical metric, as explained in Section 3.3.

[8] https://developer.nvidia.com/cuda-toolkit

Table 3. Evaluation results of the utilization of different BERT variants, GPT-2, and BART models in vulnerability prediction

Model	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F_1-score (%)**	**F_2-score (%)**
BERT	90	81	79	80	79.4
RoBERTa	87	72	73	72	72.8
DistilBERT	91	81	81	81	81
ALBERT	88	80	67	73	69.25
CodeBERT	93	87	83	85	**83.78**
GPT-2	90	84	71	77	73.14
BART	87	70	80	75	77.78

As can be seen in Table 3, the highest F_2-score, which is in bold, is achieved by CodeBERT. Moreover, CodeBERT seems to be the best model in every evaluation metric. Of particular interest is the observation that CodeBERT achieves an F_2-score 11 % higher than RoBERTa, which has the same architecture but is pre-trained on a different dataset. This observation highlights the importance of the data utilized in the pre-training process. The fact that CodeBERT proved to be the superior model in our analysis can be attributed to the nature of its pre-training knowledge. As explained in Section 3.2, CodeBERT is the only one of the examined models, which is not pre-trained only on natural language but in natural language and programming language pairs. Based on this observation, we can argue that in downstream tasks related to source code, it is beneficial to use models that have prior knowledge of source code, as well. However, we have to notice also that BERT achieves F_2-score equal to 79.4 %, which is close to CodeBERT's 83.78 %, indicating that although not pre-trained in programming languages, its good understanding of natural language is a sufficient basis for being fine-tuned on the task of VP using a labeled dataset consisting of source code.

Furthermore, regarding the comparison between the BERT variants, we can see in Table 3 that the initial BERT model succeeds a significantly higher F_2-score than RoBERTa and ALBERT, but slightly lower scores than DistilBERT. Hence, after CodeBERT which has prior knowledge of programming language, the best BERT variant that is solely pre-trained on natural language, proved to be the DistilBERT in our analysis. DistilBERT succeeds F_2-score 81 %, which is higher than BERT, RoBERTa, ALBERT by 1.6 %, 8.2 %, and 11.75 % respectively. Regarding the comparison among the three different kinds of Transformer-based models that we examined, BERT seems to be the most accurate one in VP, at least on the utilized dataset. It succeeds an F_2-score 1.6 % and 6.26 % higher than BART and GPT-2 respectively.

We also investigated if there is any correlation between the performance of the models and their size. Specifically, we examined if the F_2-scores (see Table 3) of our 7 models is correlated with the number of trainable parameters of the models (see Table 1). For this purpose, we employed the Spearman's rank coefficient [46]. Spearman's correlation is a non-parametric statistical measure used to determine whether there is a monotonic relationship between two ordinal

variables, including its strength and direction. By using the "stats" module of the "SciPy" library[9], we computed the Spearman correlation coefficient equal to 0.198. We judged the strength of the correlation utilizing the guidelines provided by Cohen et al. [47]. Cohen et al. state that a correlation of less than 0.3 is weak, between 0.3 and 0.5 is moderate, and greater than 0.5 is strong. A positive correlation that is close to one generally indicates that the studied rankings are nearly identical. Therefore, a Spearman cofficient equal to 0.198 indicates a weak positive correlation between the F_2-scores and the number of trainable parameters of the models. Hence, we can argue that the accuracy of the examined models does not depend on the models size.

A remark on the limitations and the threats to the present work's validity is considered necessary. First, we have to note that the conclusions of our study contain a threat of data validity, since they are based on our experiments on a specific training and evaluation dataset that consists of open-source code written in Python programming language. They cannot be considered as ground truth. More studies applied on different datasets and for different programming languages would contribute to the generalization of the findings. Furthermore, all the employed models are retrieved from a library provided by Hugging Face. These models is possible to have slight differences from the ones presented in the respective publications. They are based on the same architectures but they have been pre-trained with a different corpus, a different number of samples, and probable a different set of hyperparameters. However, they are considered the closest open-source implementations of the Transformer-based pre-trained models.

5 Conclusions

In this paper, our aim was to compare several pre-trained models from the field of natural language processing, which are based on the emerging deep learning architecture called Transformer, in their capacity to be fine-tuned on the downstream task of vulnerability prediction. To achieve this, we utilize BERT, DistilBERT, RoBERTa, ALBERT, CodeBERT, GPT-2, and BART models as a basis for creating vulnerability prediction models. More specifically, we added a classification layer on top of the Transformer-based models and then, we fine-tuned them using a labeled vulnerability-related dataset. The findings of our work indicate that CodeBERT is the best among the considered models in the vulnerability prediction objective, which can be attributed to the fact that it has not solely natural language-related prior knowledge, but it has also knowledge of 6 programming languages. In particular, it is pre-trained on natural language and programming language pairs.

Based on the aforementioned observation, future work includes the investigation of whether prior knowledge of natural language is useful for a downstream task associated with source code, such as in the case of vulnerability prediction,

[9] https://scipy.org/

or pre-training models exclusively on source code will be proved a better approach for downstream tasks. For this purpose, we could pre-train from scratch models such as GPT-2, BERT, and BART on their primary objective using source code data, and then fine-tune them to perform vulnerability prediction.

Acknowledgements This work was carried out as part of the project VM4SEC (Project code: KMP6-0083702) under the framework of the Action Investment Plans of Innovation of the Operational Program Central Macedonia 2014 2020, that is co-funded by the European Regional Development Fund and Greece.

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. ISO/IEC: ISO/IEC 27000:2018. https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en (Accessed: 2023-04-03)
3. Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings of the 4th ACM workshop on Quality of protection. (2008) 47–50
4. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. (2008) 315–317
5. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture **57**(3) (2011) 294–313
6. Kalouptsoglou, I., Siavvas, M., Tsoukalas, D., Kehagias, D.: Cross-project vulnerability prediction based on software metrics and deep learning. In: International Conference on Computational Science and Its Applications, Springer (2020) 877–893
7. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering **40**(10) (2014) 993–1006
8. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM conference on Computer and communications security. (2007) 529–540
9. Pang, Y., Xue, X., Wang, H.: Predicting vulnerable software components through deep neural network. In: Proceedings of the 2017 International Conference on Deep Learning Technologies. (2017) 6–10
10. Filus, K., Siavvas, M., Domańska, J., Gelenbe, E.: The random neural network as a bonding model for software vulnerability prediction. In: Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, Springer (2020) 102–116
11. Dam, H.K., Tran, T., Pham, T.: A deep language model for software code. arXiv preprint arXiv:1608.02715 (2016)
12. Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th international symposium on software reliability engineering, IEEE (2014) 23–33

13. Kalouptsoglou, I., Siavvas, M., Kehagias, D., Chatzigeorgiou, A., Ampatzoglou, A.: Examining the capacity of text mining and software metrics in vulnerability prediction. Entropy **24**(5) (2022)
14. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)
15. Dam, H.K., Tran, T., Pham, T.T.M., Ng, S.W., Grundy, J., Ghose, A.: Automatic feature learning for predicting vulnerable software components. IEEE Transactions on Software Engineering (2018)
16. Kalouptsoglou, I., Siavvas, M., Kehagias, D., Chatzigeorgiou, A., Ampatzoglou, A.: 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)
17. GENSIM: Word2vec. http://radimrehurek.com/gensim/models/word2vec.html. (Accessed: 2023-04-03)
18. Fast Text: fastText. https://fasttext.cc/ (Accessed: 2023-04-03)
19. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:1909.03496 (2019)
20. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet. IEEE Transactions on Software Engineering (2021)
21. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in neural information processing systems. (2017) 5998–6008
22. Bagheri, A., Hegedűs, P.: A comparison of different source code representation methods for vulnerability prediction in python. In Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R., eds.: Quality of Information and Communications Technology, Cham, Springer International Publishing (2021) 267–281
23. Ziems, N., Wu, S.: Security vulnerability detection using deep learning natural language processing. In: IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE (2021) 1–6
24. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. (2018)
25. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
26. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)
27. Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108 (2019)
28. Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., Soricut, R.: Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942 (2019)
29. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
30. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L.: Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461 (2019)

31. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8) (2019) 9
32. Hovsepyan, A., Scandariato, R., Joosen, W., Walden, J.: Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th international workshop on Security measurements and metrics. (2012) 7–10
33. Cao, S., Sun, X., Bo, L., Wei, Y., Li, B.: Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. Information and Software Technology **136** (2021) 106576
34. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
35. Fang, Y., Liu, Y., Huang, C., Liu, L.: Fastembed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. Plos one **15**(2) (2020) e0228439
36. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE (2018) 757–762
37. Yuan, X., Lin, G., Tai, Y., Zhang, J.: Deep neural embedding for software vulnerability discovery: Comparison and optimization. Secur. Commun. Networks **2022** (2022) 5203217:1–5203217:12
38. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). (2014) 1532–1543
39. Kim, S., Choi, J., Ahmed, M.E., Nepal, S., Kim, H.: Vuldebert: A vulnerability detection system using bert. In: 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). (2022) 69–74
40. Hanif, H., Maffeis, S.: Vulberta: Simplified source code pre-training for vulnerability detection. In: 2022 International Joint Conference on Neural Networks (IJCNN), IEEE (2022) 1–8
41. Fu, M., Tantithamthavorn, C.: Linevul: A transformer-based line-level vulnerability prediction. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR). (2022) 608–620
42. Steenhoek, B., Rahman, M.M., Jiles, R., Le, W.: An empirical study of deep learning models for vulnerability detection. arXiv preprint arXiv:2212.08109 (2022)
43. Coimbra, D., Reis, S., Abreu, R., Păsăreanu, C., Erdogmus, H.: On using distributed representations of source code for the detection of c security vulnerabilities. arXiv preprint arXiv:2106.01367 (2021)
44. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (jan 2019)
45. Kalouptsoglou, Ilias and Siavvas, Miltiadis and Ampatzoglou, Apostolos and Kehagias, Dionysios and Chatzigeorgiou, Alexander: Transformer-based models in Vulnerability Prediction. https://sites.google.com/view/transformer-models-vulpred/ (Accessed: 2023-04-10)
46. Spearman, C.: The proof and measurement of association between two things. (1961)
47. Cohen, J.: Statistical power analysis for the behavioral sciences. Academic press (2013)