Vulnerability prediction using pre-trained models: An empirical evaluation

Ilias Kalouptsoglou*†, Miltiadis Siavvas*, Apostolos Ampatzoglou†,
Dionysios Kehagias*, and Alexander Chatzigeorgiou†
* Centre for Research and Technology Hellas/Information Technologies Institute, Thessaloniki, Greece
†University of Macedonia/Department of Applied Informatics, Thessaloniki, Greece
iliaskaloup@iti.gr, siavvasm@iti.gr, a.ampatzoglou@uom.edu.gr, diok@iti.gr, achat@uom.edu.gr

***Abstract*—The rise of Large Language Models (LLMs) has provided new directions not just for natural language understanding and text generation but also for addressing downstream tasks, such as text classification. A downstream text classification task is vulnerability prediction, where segments of the source code are classified as vulnerable or not. Several recent studies have employed transfer learning in order to enhance vulnerability prediction taking advantage from the prior knowledge of the pre-trained LLMs. In the current study, different Transformer-based pre-trained LLMs are examined and evaluated in their capacity to identify vulnerability patterns in the source code and therefore predict vulnerable software components. In particular, we fine-tune in vulnerability prediction the BERT, GPT-2, and T5 models, as well as their code-oriented variants namely CodeBERT, CodeGPT, and CodeT5 respectively. Subsequently, we assess their performance and we conduct an empirical comparison between them to identify the models that are the most accurate ones in vulnerability prediction.**

***Index Terms*—Software security, Vulnerability prediction, Transfer learning, Large language models, Transformer**

I. 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 for the deployment of any software product. A technique capable of predicting hot-spots (e.g., files, classes, methods, etc.) of a software product that contain vulnerabilities is Vulnerability Prediction (VP). Software enterprises can benefit from such a mechanism by allocating their limited time and testing effort to potentially vulnerable components.

Vulnerability prediction is commonly performed through Machine Learning (ML) 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 ML model [3]–[6]. On the other hand, text mining approaches receive the text of the source code as input and through ML models, they identify textual 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 ML 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 language knowledge to the downstream task of classifying software components as vulnerable or not.

However, the wide variety of LLMs that have already

been proposed in the literature on NLP and the different characteristics and specialties of each raise questions about which ones are the most suitable and competent for code analysis and in particular for the downstream task of VP. For instance, there are LLMs focused on natural language (NL) understanding whereas other have been designed for source code analysis. Moreover, some LLMs utilize both the encoder and decoder parts of the Transformer architecture whereas some other use only the encoder or the decoder part.

The objective of this study is to identify the optimal choice among a variety of pre-trained Transformer-based models in the downstream task of VP showcasing any differences in their performance. To this end, our study aims at assisting researchers in their future endeavors to use transfer learning for VP, providing them indications on which of the models found in the related literature are the most appropriate for this specific objective. For this purpose, we fine-tune several large pre-trained models on a labeled vulnerability-related dataset of Python source code. More specifically, in this paper, we train (i.e., fine-tune), evaluate, and compare three NLP models and their code-oriented variants, which are all based on the Transformer architecture [21]. These are the following models:

- Bidirectional Encoder Representations from Transformers (BERT) [25]
- Generative Pre-trained Transformer (GPT) [24], [26]
- Text-To-Text Transfer Transformer (T5) [27]
- CodeBERT [28]
- CodeGPT [29]
- CodeT5 [30]

The rest of the paper consists of the following parts: In Section II we provide a summary of the state-of-the-art approaches in the VP field focusing on the text mining-based ones. In Section III, we provide details for the examined models and we describe thoroughly the methodology that we follow to conduct the current study, while in Section IV, we present the results of our evaluation scheme. Section V discusses the threats to validity, and finally, Section VI concludes the study and provides future research directions.

II. Related Work

The main VPMs in the literature utilize ML-based approaches that use software metrics [3]–[5], [31], [32] or text mining [7]–[9] to predict vulnerabilities, as stated in [33], [34]. Early efforts [3]–[5] proposed using statically extracted metrics as indicators of vulnerabilities in software components. More specifically, a method using software metrics such as complexity, coupling, and cohesion was presented by Chowdhury and Zulkernine [32], [35]. They analyzed Mozilla Firefox and achieved an average recall of almost 75%. To develop a VPM, Zimmerman et al. [36] conducted an empirical study on Windows Vista, evaluating the effectiveness of code churn, code complexity, dependencies, and organizational variables. They obtained low recall but high precision. Moreover, Nguyen et al. [37] presented a model training method based on a code metric extracted from the dependency graph and tested it on the Mozilla JavaScript Engine.

In the related literature, VP using text mining has shown encouraging results [7], [12], [19], [38]. Early research efforts that employed text mining paid attention on the BoW technique [7], [12]. Next efforts focused 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], [39] employed the word2vec [17] tool to create embedding vectors for the source code tokens, whereas Zhou et al. [19] used the pre-trained word2vec vectors. Fang et al. [40] 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 [41] 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.

In [13] the authors compared models built based on software metrics with models based on text mining. The results showed significant benefit from text mining. Moreover, a systematic mapping study in the software VP domain [34] observed the superiority of the text mining-based approaches and the increased preference of the research community to these methods. It also highlighted the need to explore the use of Transformer-based pre-trained models that are already commonly utilized in NLP through transfer learning.

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. [42] compared a CodeBERT-based embedding method with word2vec, fastText, and GloVe [43] showing that the former outperforms the latter in the task of vulnerability prediction.

Coimbra et al. in their study [44], presented an evaluation of the Code2vec [45] 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 that is encapsulated by simple Transformer-based models. Their findings highlight that both approaches succeeded comparable results.

Kim et al. [46] proposed a model called VulDeBERT, which is the BERT pre-trained 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 [47], 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 [48]. 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 [49].

Additionally, Zhang et al. [50] explored the capability of LLMs to discover vulnerabilities by evaluating both proprietary models, such as ChatGPT, and open-source models, such as CodeBERT. They employed prompt engineering techniques for proprietary models and fine-tuning for open-source models. Finally, Carletti et al. [51] investigated how modern DL techniques perform in finding vulnerabilities in C/C++ source code from real software projects. They compared common text-mining DL methods with advanced approaches such as Transformer and GNNs, aiming to establish a benchmark for evaluating vulnerability detection methods.

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 knowledge 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 and accurate for this task than others. In the majority of the existing research works, there is not any justification for the selection of a specific LLM as a basis for a VP model.

In the present study, 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 leverage popular open-source pre-trained LLMs and we fine-tune them using a vulnerability-related dataset from real world software components written in Python programming language. Specifically, we fine-tune BERT, GPT-2, T5, and their pre-trained on code variants namely as CodeBERT, CodeGPT-2, and CodeT5. First, we examine the efficiency and the accuracy of those techniques in VP by comparing them to state-of-the-art text mining techniques, and then we compare them with each other. Therefore, we can 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, considering encoder-only (e.g., BERT), decoder-only (e.g. GPT), and encoder-decoder (e.g., T5) model architectures. To this end, we present our observations regarding the role of the encoder and decoder parts as well as the role of the size of the models. Moreover, we follow the same procedure for all of the LLM-based VPMs 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. Hence, 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 [52].

III. 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 BERT, GPT-2, T5, as well as CodeBERT, CodeGPT-2, and CodeT5 using a real-world dataset retrieved by commits on GitHub projects.

A. 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 clean (i.e., non-vulnerable) versions from GitHub. For this purpose, we retrieved files from the latest version of the dataset's GitHub repositories, since the latest versions are the safest versions that can be considered as non-vulnerable, because no vulnerabilities have yet been reported for them. 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 preprocessing 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 [52].

B. Strategy

This section describes all the Large Language Models (LLMs) 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.

1) Models Characteristics: To begin with, the BERT model presented by Google AI [1] is pre-trained on the Masked Language Modeling (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, Liu et al. [53] observed that BERT was significantly undertrained, and therefore, they proposed a robustly optimized pre-training approach, called RoBERTa.

A RoBERTa-based model called CodeBERT, which was presented by Microsoft[2], is the the examined BERT variant 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 [28].

A similar but different to BERT Transformer-based architecture is the Generative pre-trained transformer (GPT) [24] that was developed by OpenAI[3]. 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.

Lu et al. introduced CodeGPT-2 [29], a variant of GPT-2. It was pre-trained using programming language data obtained from the Java and Python subsets of the CodeSearchNet[4] dataset. It was specifically designed using prior knowledge of programming languages, although it retains the same Transformer decoder-only architecture and the next-token prediction objective as GPT-2. The utility of CodeGPT-2 in software development and code understanding activities can be increased by its fine-tuning for a variety of coding tasks, including code completion, code summarization, and error detection among others [29].

Google AI has released another Transformer-based pre-trained language model named T5. T5, introduced by Raffel et al. [27], is an encoder-decoder model pre-trained on a combination of unsupervised and supervised tasks, where each task is converted into a text-to-text format. Using a large dataset, spans of text are masked during pre-training and the model learns to predict these missing spans. This versatile approach allows T5 to be fine-tuned for various specific NLP tasks, achieving strong performance across numerous benchmarks [27].

Finally, Wang et al. provided CodeT5 [30] by training the T5 model on the semantics of code using identifiers provided by developers. CodeT5 uses the Text-To-Text Transfer Transformer framework, which has been pre-trained in a variety of programming languages from the CodeSearchNet

[1]https://en.wikipedia.org/wiki/Google_AI

[2]https://github.com/microsoft/CodeBERT

[3]https://openai.com/

[4]https://huggingface.co/datasets/code_search_net

dataset. By leveraging the encoder and decoder parts of the T5 architecture, this model performs well on a variety of code interpretation and generation tasks, including code summarization, code generation, code translation, and code completion [30].

For all the aforementioned models, we utilize the implementations that are provided by Hugging Face[5] (HF). More specifically, for BERT we use the *bert-base* model, which has 12 layers, 768 hidden size, 12 attention heads in each attention block [21], and 110 millions (M) parameters. For CodeBERT, we choose the *codebert-base-mlm* version, which has the same architecture with *roberta-base*. Specifically, it is a slight differentiation of the BERT architecture with 125M parameters. Regarding GPT-2, we use the *gpt2* model with 12 layers, 768 hidden size, 12 attention heads, and 124M parameters. Similarly, CodeGPT-2 consists of 12 layers, 768 hidden size, 12 attention heads, and 124M trainable parameters. Finally, for T5 and CodeT5 models that employ both the encoder and the decoder parts of the Transformer architecture, we utilize the *t5-base* and *codet5-base* implementations respectively, which have nearly 220M trainable weights. They also contain 12 layers for both the encoder and the decoder (i.e., 24 in total), 768 hidden size, and 12 attention heads in each attention block. All the aforementioned statistics are summarized in Table I.

2) Approach: For the construction of the aforementioned models, we followed an approached that is illustrated in Fig. 1. As can be seen in Fig. 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 for VP.

During fine-tuning, the training dataset described in Section III-A is utilized. Each Python file with source code is pre-processed (see Section III-A) 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 (i.e., the classification head).

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 III-C), while for the max length we used the maximum length of the sequences of tokens in the dataset that can be processed by these Transformer-based models.

In particular, for each examined model, we used its pre-trained Transformer-based architecture along with an additional classification head and we train it using a learning rate equal to 0.00002. For GPT-based models a linear scheduler was also applied. For the gradient descent's optimization, we leveraged the Weighted Adam (AdamW) [54] for the BERT and T5-based models, and simple Adam [55] in case of GPT models. For the selection of the epochs number, we applied the Early Stopping technique[6]. In addition, the maximum length was configured equal to 512, which is the maximum input size that those models can receive. While the text data were encoded using the corresponding tokenizers, zero padding was used to ensure constant sequence length and the truncation technique was used to truncate sequences that were longer than the allowed length. Finally, the loss function was the Cross-Entropy loss [56] in every case.

In the execution phase, after fine-tuning the optimised Transformer-based classifiers for the objective of VP, we can use them as our vulnerability predictors in order to assess new Python files and classify them as vulnerable or not.

C. 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 training set, 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

[5]https://huggingface.co/

[6]https://en.wikipedia.org/wiki/Early_stopping

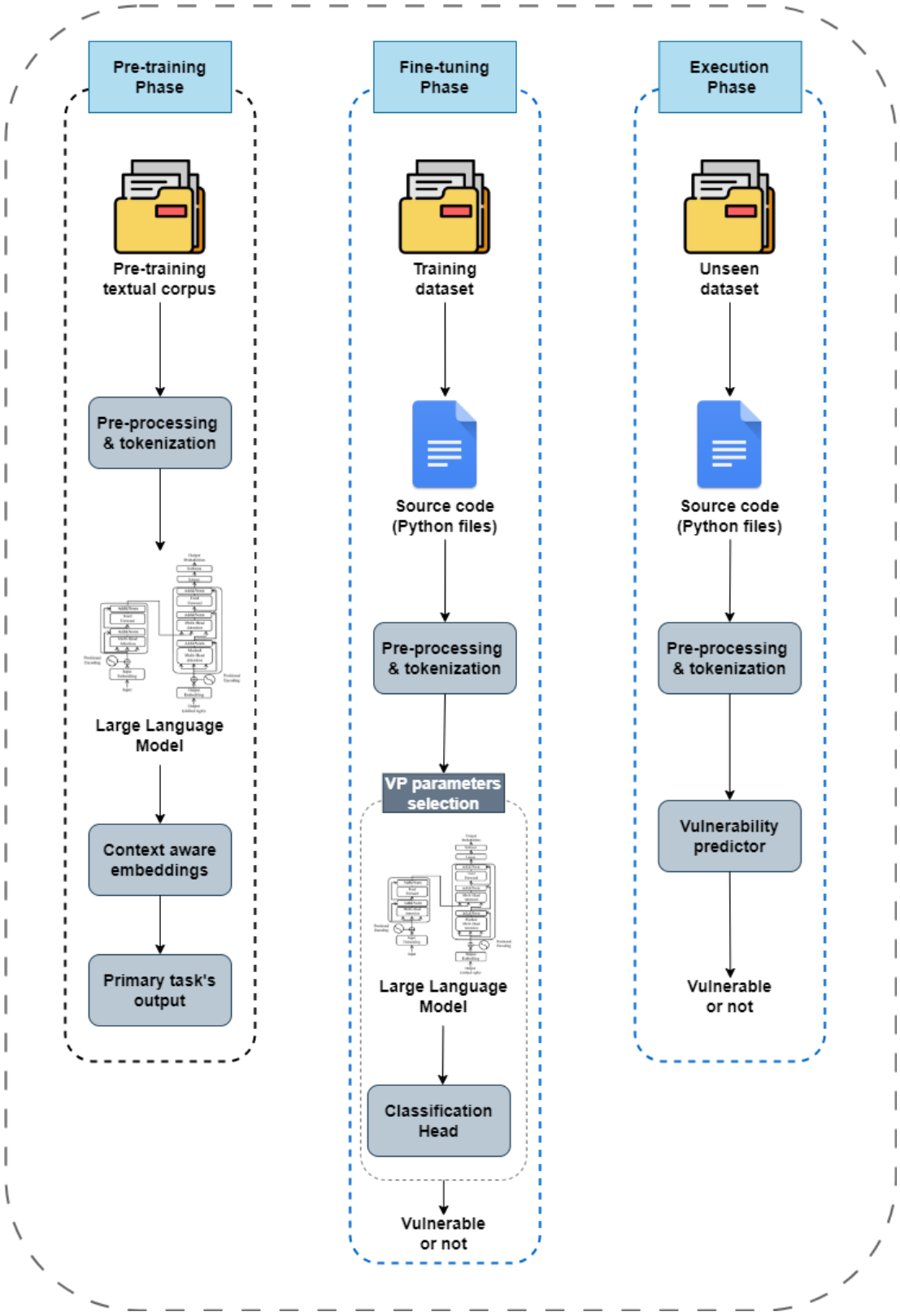

Fig. 1. An overview of the approach followed for fine-tuning Transformer-based pre-trained models for vulnerability prediction.

TABLE I
THE CHARACTERISTICS OF THE PRE-TRAINED MODELS CONSIDERED IN THE STUDY

Model	Version	Layers	Hidden Size	Attention heads	Parameters
BERT	*bert-base-uncased*	12	768	12	110M
CodeBERT	*codebert-base-mlm*	12	768	12	125M
GPT-2	*gpt2*	12	768	12	124M
CodeGPT-2	*CodeGPT-small-py*	12	768	12	124M
T5	*t5-base*	24	768	12	220M
codeT5	*codet5-base*	24	768	12	220M

indicators. For 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 important 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 (and therefore to the identification of real vulnerabilities) than to precision, whereas the F_1-score gives equal weight to them. The mathematical formula of F_2-score is given below:

$$F_2 = \frac{5 \times precision \times recall}{4 \times precision + recall} \quad (1)$$

IV. RESULTS AND DISCUSSION

This section presents the results of our comparative study among the Transformer-based models, after fine-tuning them on the downstream task of VP. The experiments took place on a parallel computing platform called CUDA[7] that we have installed on a GeForce RTX 4080 Nvidia GPU. For the implementation of the Transformer-based methods, we utilized the PyTorch framework, whereas for the state-of-the-are text mining-based approaches, we used the Tensorflow framework. To enhance the reproducibility of the experiments, we provide our scripts online [52].

All the reported results are the outcome of applying the fine-tuned LLMs on the testing set of the dataset (see Section III-C) along with some state-of-the-art text mining-based approaches. In Table II, we present the outcome of our evaluation scheme. More specifically, Table II 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 III-C.

First, Table II presents the superiority of the LLM-based VPMs over three state-of-the-art text mining-based approaches. In particular, we compare the fine-tuned LLMs to (i) BoW, (ii) Term Frequency–Inverse Document Frequency (TF-IDF)[8], which is a BoW variant, and (iii) word2vec word embeddings along with a DL classifier. The two BoW-based methods used a ML classifier to classify files as vulnerable or non-vulnerable. The best classifier proved to be the Random Forest model. On the other hand, the word2vec embeddings were fed to an LSTM model, which has demonstrated the best results among non-Transformer models in the VP literature [34]. The results in Table II show that:

All examined LLMs manage to achieve superior results compared to BoW, TF-IDF, and word2vec, highlighting the important benefit gained by leveraging transfer learning in VP.

Furthermore, we can notice that all the three pre-trained on code LLMs manage to outperform their NLP variants. In particular, CodeBERT surpasses BERT by 3.2% in terms of F_2-score, while both CodeGPT-2 and CodeT5 outperform GPT-2 and T5 by 2.3%. This observation highlights the importance of the data utilized in the pre-training process. The fact that CodeBERT, CodeGPT-2, and CodeT5 proved to be the superior models can be attributed to the nature of their pre-training knowledge. Hence, we can argue that:

In the VP downstream task, which is a source code related task, it is beneficial to use models that have prior knowledge of programming languages.

In addition, as can be seen in Table II, the highest F_2-score (i.e., 84.3%), which is in bold, is achieved by CodeBERT but the second highest is achieved by BERT instead of CodeT5 or CodeGPT-2. In other words, BERT surpasses clearly not only the other two NLP models but also their code-aware variants. This finding suggests that Transformer-based models that leverage the encoder part of the Transformer architecture (e.g., BERT and CodeBERT) may be more capable in predicting vulnerable software components as opposed to models that include only the decoder (e.g., GPT-2 and CodeGPT-2) or the whole Transformer architecture (e.g., T5 and CodeT5), and therefore, a more detailed analysis can be conducted in this direction. Concisely, we noticed that:

The encoder-only BERT and CodeBERT outperformed the decoder-only GPT-2 and CodeGPT-2 as well as the encoder-decoder T5 and CodeT5.

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 II) of our six fine-tuned models is correlated with the number of trainable parameters of these models, which is presented in Table III. For this purpose, we employed the Spearman's rank coefficient [57]. Spearman's correlation is a non-parametric statistical measure used to determine whether there is a monotonic relationship

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

[8]https://en.wikipedia.org/wiki/Tf%E2%80%93idf

TABLE II
EVALUATION RESULTS OF THE OVERALL ANALYSIS

Model	Accuracy (%)	Precision (%)	Recall (%)	F_1-score (%)	F_2-score (%)
BoW	89.7	96.7	59.0	73.2	63.9
TF-IDF	90.2	98.3	60.0	74.5	65.0
Word2vec	85.2	69.3	68.0	68.7	68.2
BERT	93.0	90.8	79.0	84.5	81.1
GPT-2	87.3	77.0	67.0	71.6	68.8
T5	91.1	88.9	72.0	79.5	74.8
CodeBERT	90.9	78.1	86.1	81.9	**84.3**
CodeGPT-2	88.7	81.1	68.9	74.5	71.1
CodeT5	90.9	86.0	74.0	79.5	76.1

TABLE III
NUMBER OF TRAINABLE PARAMETERS OF THE FINE-TUNED MODELS.

Model	Number of Trainable Parameters
BERT	109,485,314
GPT-2	124,443,648
T5	223,475,714
CodeBERT	124,648,706
CodeGPT-2	124,248,576
CodeT5	223,475,714

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 0.058. We judged the strength of the correlation utilizing the guidelines provided by Cohen et al. [58]. 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 0.058 indicates a very weak positive correlation between the F_2-scores and the number of trainable parameters of the models. Hence, we can argue that:

The efficiency (expressed by the F_2-score) of the examined LLMs fine-tuned in vulnerability prediction does not depend significantly on the number of trainable parameters and, therefore, on the size of the models.

V. THREATS TO VALIDITY

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. More studies applied on different datasets and for different programming languages would contribute to the generalization of the findings.

Furthermore, a threat to internal validity is that not all possible combinations of hyperparameter values have been explored. The interdependence of model hyperparameters can make it challenging to identify the impact of individual hyperparameters, possibly resulting in sub-optimal model performance. To address this, we undertook a comprehensive hyperparameter tuning process.

Finally, an external validity threat concerns the fact that all the employed models are retrieved from a Transformers 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, although they are considered the closest open-source implementations of the Transformer-based pre-trained models, they have been pre-trained within a slightly different setting.

[9]https://scipy.org/

VI. 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, CodeBERT, GPT-2, CodeGPT-2, T5, and CodeT5 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 showed that the three pre-trained on code LLMs achieved superior results as opposed to their pre-trained on natural language variants. We also observed an important benefit from the encoder part of the Transformer since encoder-only models (i.e., BERT and CodeBERT) were the two best performers. Moreover, we did not notice a statistically significant relationship between the performance of the fine-tuned LLMs and their size (i.e., number of trainable weights). Finally, all LLM-based VP solutions outperformed traditional text mining-based methods.

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, or pre-training models exclusively on source code will be proved a better approach for downstream tasks. In addition, future research endeavors may examine explainability techniques to reduce the level of code granularity to function-level or even to line-level.

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 27000:2018," https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en, Accessed: 2023-04-03.

[3] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008, pp. 47–50.

[4] ——, "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*, ser. ESEM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 315–317. [Online]. Available: https://doi.org/10.1145/1414004.1414065

[5] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[6] 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.

[7] 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.

[8] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 529–540.

[9] 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.

[10] K. Filus, M. Siavvas, J. Domańska, and E. Gelenbe, "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, pp. 102–116.

[11] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv preprint arXiv:1608.02715*, 2016.

[12] 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.

[13] 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

[14] 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.

[15] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, 2018.

[16] 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.

[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] Fast Text, "fastText," https://fasttext.cc/, Accessed: 2023-04-03.

[19] 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.

[20] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[21] 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.

[22] A. Bagheri and P. Hegedűs, "A comparison of different source code representation methods for vulnerability prediction in python," in *Quality of Information and Communications Technology*, A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, and R. Pérez-Castillo, Eds. Cham: Springer International Publishing, 2021, pp. 267–281.

[23] N. Ziems and S. Wu, "Security vulnerability detection using deep learning natural language processing," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2021, pp. 1–6.

[24] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[25] 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.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[27] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[28] 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.

[29] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[30] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[31] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[32] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proceedings of the fourth international workshop on Software engineering for secure systems*, 2008, pp. 57–64.

[33] M. Jimenez, M. Papadakis, and Y. Le Traon, "Vulnerability prediction models: A case study on the linux kernel," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 1–10.

[34] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "Software vulnerability prediction: A systematic mapping study," *Information and Software Technology*, vol. 164, p. 107303, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058492300157X

[35] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 1963–1969.

[36] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *2010 Third international conference on software testing, verification and validation*. IEEE, 2010, pp. 421–428.

[37] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th international workshop on security measurements and metrics*, 2010, pp. 1–8.

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

[39] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[40] Y. Fang, Y. Liu, C. Huang, and L. Liu, "Fastembed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm," *Plos one*, vol. 15, no. 2, p. e0228439, 2020.

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

[42] X. Yuan, G. Lin, Y. Tai, and J. Zhang, "Deep neural embedding for software vulnerability discovery: Comparison and optimization," *Secur. Commun. Networks*, vol. 2022, pp. 5 203 217:1–5 203 217:12, 2022.

[43] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[44] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, and H. Erdogmus, "On using distributed representations of source code for the detection of c security vulnerabilities," *arXiv preprint arXiv:2106.01367*, 2021.

[45] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290353

[46] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim, "Vuldebert: A vulnerability detection system using bert," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2022, pp. 69–74.

[47] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[48] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 608–620.

[49] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," *arXiv preprint arXiv:2212.08109*, 2022.

[50] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, and Y. Liu, "An empirical study of automated vulnerability localization with large language models," *arXiv preprint arXiv:2404.00287*, 2024.

[51] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Predicting source code vulnerabilities using deep learning: A fair comparison on real data," 2024.

[52] Kalouptsoglou, Ilias and Siavvas, Miltiadis and Ampatzoglou, Apostolos and Kehagias, Dionysios and Chatzigeorgiou, Alexander, "Vulnerability prediction using pre-trained models," https://sites.google.com/view/vpllm/, Accessed: 2024-08-01.

[53] 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.

[54] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv:1711.05101*, 2017.

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

[56] A. Mao, M. Mohri, and Y. Zhong, "Cross-entropy loss functions: Theoretical analysis and applications," in *International conference on Machine learning*. PMLR, 2023, pp. 23 803–23 828.

[57] C. Spearman, "The proof and measurement of association between two things." 1961.

[58] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.