# Transfer Learning for Software Vulnerability Prediction using Transformer Models

Ilias Kalouptsoglou[a,b,*], Miltiadis Siavvas[a], Apostolos Ampatzoglou[b], Dionysios Kehagias[a], Alexander Chatzigeorgiou[b]

[a] *Centre for Research and Technology Hellas/Information Technologies Institute, 6th km Charilaou-Thermi Rd, Thermi, 57001, Thessaloniki, Greece*
[b] *University of Macedonia/Department of Applied Informatics, Egnatia 156, Thessaloniki, 54636, Thessaloniki, Greece*

**Abstract**

Recently software security community has exploited text mining and deep learning methods to identify vulnerabilities. To this end, the progress in the field of Natural Language Processing (NLP) has opened a new direction in constructing Vulnerability Prediction (VP) models by employing Transformer-based pre-trained models. This study investigates the capacity of Generative Pre-trained Transformer (GPT), and Bidirectional Encoder Representations from Transformers (BERT) to enhance the VP process by capturing semantic and syntactic information in the source code. Specifically, we examine different ways of using CodeGPT and CodeBERT to build VP models to maximize the benefit of their use for the downstream task of VP. To enhance the performance of the models we explore fine-tuning, word embedding, and sentence embedding extraction methods. We also compare VP models based on Transformers trained on code from scratch or after natural language pre-training. Furthermore, we compare these architectures to state-of-the-art text mining and graph-based approaches. The results showcase that training a separate deep learning predictor with pre-trained word embeddings is a more efficient approach in VP than either fine-tuning or extracting sentence-level features. The findings also highlight the importance of context-aware embeddings in the models' attempt to identify vulnerable patterns in the source code.

*Keywords:* Software security, Deep learning, Transfer learning, Transformer, Vulnerability prediction

## 1. Introduction

The security level of software systems is a major concern for software development enterprises, which want to produce high-quality and secure software free of vulnerabilities.

*Corresponding Author.
*Email addresses:* iliaskaloup@iti.gr, iliaskaloup@uom.edu.gr (Ilias Kalouptsoglou), siavvasm@iti.gr (Miltiadis Siavvas), a.ampatzoglou@uom.edu.gr (Apostolos Ampatzoglou), diok@iti.gr (Dionysios Kehagias), achat@uom.edu.gr (Alexander Chatzigeorgiou)

Security vulnerabilities are weaknesses in the software, which can be exploited by external threats [1]. The number of new Common Vulnerabilities and Exposures (CVEs) that are discovered annually has increased significantly since 2017 and continues its upward trend[1]. Therefore, there is a need for techniques capable of identifying vulnerabilities in software and hence, to prevent their exploitation. Vulnerability Prediction (VP) refers to the set of techniques that can assist software developers to prioritize their inspection efforts and time by identifying the vulnerable components of a software system.

The number of research publications in the field of VP is steadily growing [2]. Those studies mainly propose Vulnerability Prediction Models (VPMs), which aim at classifying the examined software components as vulnerable or not. VPMs commonly comprise Machine Learning (ML) models, which are fed with software characteristics encoded mainly in the form of software metrics or in textual form (i.e., text mining). Text mining-based models seem to be the most promising ones [3],[4] and have attracted the most research interest, as well [2].

Initially, text mining methods utilized the Bag of Words (BoW) technique for representing the source code as a set of words, each accompanied by the number of occurrences or frequency of occurrence in the code [4],[5]. The advances in Deep Learning (DL) led to the creation of more complex models, which are trained to learn sequential data that consist of large sequences of tokens (i.e., words) of the source code [3],[6],[7],[8],[9]. In this approach, the tokens of the source code are encoded with the so-called word embedding vectors using algorithms such as word2vec [10] and then they are given as input into DL models, usually into Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs).

Later studies appeared, which proposed the representation of the source code in text-rich graphs (e.g., Abstract Syntax Trees, Code Property Graphs, etc.) [11],[12] in order to capture more meaningful syntactic and semantic relationships between the tokens than the traditional word embedding techniques (e.g., word2vec, fastText [13], etc.). Those text-rich graphs are fed into Graph Neural Networks (GNNs) that generate the graph embeddings of the analyzed software components and then the produced embeddings are given as input to a ML classifier.

Recently, studies using transfer learning to construct accurate VPMs based on text mining have begun to emerge [14],[15],[16]. Since the introduction of the Transformer architecture [17], several pre-trained Large Language Models (LLMs) have been proposed by the leading companies in Natural Language Processing (NLP) and Artificial Intelligence (AI) fields, such as the Bidirectional Encoder Representations from Transformers (BERT) [18] and the Generative Pre-trained from Transformer (GPT) [19]. Those models have acquired a deep understanding of Natural Language (NL) through being trained in a primary task, such as the Masked Language Modeling (MLM), and they can be further trained (i.e., fine-tuned) for several downstream tasks (e.g., text classification).

Such a downstream task which could benefit from transfer learning is text mining-based VP. Early attempts of using Transformer-based models for VP, examined whether extracting pre-trained Transformer embeddings (most commonly BERT embeddings), instead of using embeddings generated by word2vec or fastText, is beneficial in predicting vulnerabilities [14],[20]. Some other studies proceeded with fine-tuning Transformer-based models on the classification task of VP using a labeled vulnerability-related dataset

[1] `https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures`

[15],[16]. Most studies that employed pre-trained models for VP compared their approach with previous state-of-the-art methods, and their findings indicated that transfer learning is a promising solution in the field.

However, there is a variety of techniques and different directions (i.e., implementation choices) considered when applying transfer learning for tasks related to code analysis in general and VP in particular. First, as mentioned above, the studies in the VP-related literature use different implementation choices to construct Transformer-based VPMs. For instance, some studies use pre-trained models to extract their embeddings, feed them to separate ML models, and train the models on the classification task (i.e., word embeddings extraction approach) [20], whereas some others train the entire pre-trained model on VP (i.e., fine-tuning approach) [16],[21]. One can also freeze the pre-trained layers, extract the sentence embeddings, and train only the classification head (i.e., sentence-level embeddings extraction approach).

Moreover, there are questions as to whether bimodal LLMs that are pre-trained in both NL and Programming Language (PL), or LLMs solely specialized in coding tasks are more suitable for the task of VP. Such bimodal models may offer an advantage by leveraging both semantic information from NL (e.g., relations between function names and targeted functionalities) and contextual information from comments in the code, which can potentially improve understanding of the code and, thus, improve the accuracy of VP. In other words, it is a challenge to investigate whether the choice of including prior knowledge of NL can assist code-oriented LLMs to capture those semantic relations in source code that are similar to NL semantics or whether it is redundant. At this point, we have to specify that by the term bimodal we refer to the two examined data formats (i.e., PL and NL modalities). Therefore, when we refer to bimodal models in this study, we are referring to models that have been pre-trained on these two kinds of data, whereas mentioning unimodal models refers to models pre-trained exclusively on code.

Overall, it can be argued that there are various implementation choices that can be made during the implementation of transfer learning-based VPMs. However, previous research attempts just presented the model that was derived from their implementation choices, without providing insight on how these choices affected the model performance and which actually played the most important role in the final accuracy of the model. To the best of our knowledge, no research endeavor exists that specifically evaluates the impact of different implementation choices on the predictive performance of Transformer-based VPMs. These questions are of high interest both for practitioners who would like to understand how their implementation choices could affect the VPM performance, and for facilitating further research in the field of VP.

To this end, this study aims at answering the aforementioned questions and dilemmas. More specifically, our objective is to explore transfer learning for VP, with the intention to not only check whether there is any benefit to applying transfer learning to VP, but mainly to examine how we can benefit the most by leveraging the utilization of pre-trained Transformer models for this purpose. In other words, the main objective of our work is to examine which of the implementation choices that are fundamental part of the Transformer-based model construction, seem to play a more important role in the model's accuracy in the downstream task of VP. It is important to shed some light on the field with an empirical evaluation scheme and, therefore, to show which transfer learning methods are most beneficial to VP, thereby assisting future research endeavors in selecting the optimal ones. For instance, our study could guide researchers and practitioners on which of all the pre-trained models that continuously arise to choose for their analyses.

To conduct our analysis, we employ the CodeGPT [22] model, which is the pre-trained on code variant of the GPT model, specifically the GPT-2 model, which is the latest open source version of GPT. We also use the CodeBERT [23] model (i.e., pre-trained on code variant of BERT). We proceed with the selection of these two models, which are variants of two of the most widely used Transformer-based models (i.e., GPT and BERT) in software engineering and in VP [24], as representative examples of two distinct architectures. Specifically, CodeGPT-2 represents a decoder-only Transformer-based model, whereas CodeBERT represents an encoder-only one. Briefly, our contributions are summarized as follows:

- We investigated the two common ways of applying transfer learning in vulnerability prediction: (i) feature-based approach (i.e., embedding extraction), and (ii) fine-tuning approach.
- We compared the usage of pre-trained on code unimodal models to bimodal models with both programming and natural language understanding.
- We identified the optimal way of utilizing CodeGPT-2 and CodeBERT in vulnerability prediction.
- We compared with several state-of-the-art approaches, including BoW and word2vec embeddings, which, in contrast with Transformer's embeddings, are not contextual [25], as well as with graph-based models.

The rest of the paper is organized as follows: Section 2 provides a summary of the existing work in the related literature. Section 3 describes thoroughly our approach, the utilized dataset, our experimental setup, and the evaluation scheme that we follow. In Section 4, we present the results of the conducted experiments, and in Section 5, we discuss our findings and the lessons learned. Section 6 discusses threats to validity, and finally, Section 7 concludes the paper and provides directions for future work.

## 2. Related work

In this section, we discuss related work on software VP using ML techniques. We also present the background of the Transformer models, and subsequently, we provide an overview of the previous studies that utilized variants of the Transformer architecture to perform VP.

### *2.1. Vulnerability prediction*

Vulnerability prediction models are commonly created using ML techniques that utilize software attributes as features. The two most widespread input kinds of VPMs are (i) software metrics, and (ii) text features. Shin and Williams [26],[27] examined the capacity of complexity metrics to predict software vulnerabilities. Chowdhury and Zulkernine [28] observed that metrics such as complexity, coupling, and cohesion can be used efficiently to predict vulnerabilities using ML algorithms (e.g., Decision Trees, Naive Bayes, etc.). Kalouptsoglou et al. [29] presented a Multilayer Perceptron (MLP)-based approach using several software metrics extracted by static code analysis as features in order to perform cross-project VP with sufficient results.

Text mining-based attempts initially represented the source code as BoW, i.e., a set of words along with their frequencies of appearance [5],[30]. Later on, various researchers represented the source code as sequences of tokens (i.e., words), encoded the sequences to numerical vectors, and provided them to DL models suitable for learning sequential data, such as RNNs and CNNs. In particular, Dam et al. [6] trained a Long Short-Term Memory (LSTM) model to learn sequences of tokens represented with numerical vectors. Li et al. [8] proposed a Bidirectional LSTM model, which received as input slices of code tokens after transforming them to word2vec embeddings. In [9], an empirical evaluation of different techniques for word embedding software components took place, showcasing the efficiency of word2vec. Moreover, Li et al. [31] proposed the SySeVR framework that uses DL to detect vulnerabilities, focusing on obtaining code representations capable of accommodating semantic and syntax information related to vulnerabilities.

In their study [11], Zhou et al. proposed Devign, a model based on GNNs, for identifying vulnerable source code functions. They extracted graphical representations of the source code such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graphs (DFG) etc., and through GNNs, they generated graph embeddings of the source code functions. Those embeddings were then fed to a classifier in order to classify the functions as vulnerable or not. Chakraborty et al. [12] presented a DL-based method named ReVeal, where they extracted graphical representations of the source code, specifically Code Property Graphs (CPGs) [32]. They fed the CPGs to GNNs to learn the graph embeddings, and then, they used an MLP to classify the functions as vulnerable or not. Through their analysis, they replicated several state-of-the-art methods. Their findings highlighted the failure of the existing approaches to perform accurate VP on real-world data and emerged the need for techniques of greater precision.

### *2.2. Transformer models*

Bahdanau et al. [33] proposed the Attention mechanism in neural machine translation, addressing the issue of using a fixed-length context vector for input sentences. Specifically, the Attention mechanism assigns Attention weights to each element of the input sequence, and thus, it enables sequence-to-sequence models to generate new words by searching specific positions, which contain the most relevant information [33]. Based on the Attention mechanism, Vaswani et al. [17] introduced the Transformer architecture for sequence-to-sequence tasks managing to outperform state-of-the-art models (e.g., CNN and LSTM) in accuracy and training cost.

Radford et al. [19] developed OpenAI GPT leveraging the decoder part of the Transformer architecture. GPT was pre-trained in next word prediction to gain a deep understanding of NL and then fine-tuned on objectives such as questioning-answering, sentence similarity, etc. Next, Devlin et al. [18] introduced Google AI's pre-trained model called BERT, which utilizes the encoder part of the Transformer architecture. Having been pre-trained on the MLM objective, where certain tokens of the input sentences have been masked and then the model is trained to predict the masked tokens, BERT can be fine-tuned in various downstream tasks. BERT has attracted much interest in the NLP field, forming the basis for the development of many other models [34],[35],[36].

Furthermore, Facebook AI presented Bidirectional Auto-Regressive Transformers (BART) [37], proposing a pre-trained autoencoder for sequence-to-sequence learning. By using both the encoder and decoder parts of the Transformer architecture, they pre-trained a model that demonstrated high accuracy when fine-tuned in text generation tasks. Later on, more pre-trained Transformer-based models appeared in the NLP-related literature, such as Google's Text-to-Text Transfer Transformer (T5) [38] and Language Model for Dialog Application (LaMDA) [39].

### *2.3. Transformer models in vulnerability prediction*

In an early attempt to employ Transformer models in VP, Bagheri et al. [20] presented a comparison of various Python source code encoding techniques for VP. Specifically, they examined the effectiveness of word2vec, fastText, and BERT embeddings along with an LSTM model. Yuan et al. compared traditional word2vec, fastText, and glove [40] embedding techniques with the embeddings gained from BERT's code variant called CodeBERT [41]. Their results highlighted the effectiveness of CodeBERT embeddings.

VulDeBERT was a study that applied fine-tuning on BERT pre-trained model to the downstream task of VP, succeeding promising results [42]. VulBERTa also managed to surpass several state-of-the-art approaches having pre-trained knowledge of source code [43]. Fu et al. proposed LineVul that was a promising attempt to employ a Transformer model in VP [21]. In particular, they used the attention mechanism of the BERT architecture in order to detect vulnerabilities at line level. In a more recent endeavor, Purba et al. utilized some of the most powerful LLMs to perform vulnerability detection showing promising results in comparison with traditional static code analysis tools [44].

### *2.4. Beyond state-of-the-art*

As opposed to the aforementioned studies, in this paper, we do not intend to propose a new model, but we are interested in how to get the most out of transfer learning in VP. More specifically, we empirically examine how implementation choices that are fundamental in the Transformer-based model construction process affect the performance of the produced VPMs. Existing research works were limited to introducing a novel Transformer-based VPM, without examining (at least reporting) how their implementation choices affected the model performance. In particular, Bagheri et al. [20] and Yuan et al. [41], were limited to the utilization of pre-trained word embeddings without considering other transfer learning approaches. Furthermore, VulDeBERT [42], VulBERTa [43] and LineVul [21] as well as the study of Purba et al. [44] applied solely fine-tuning without examining whether the utilization of pre-trained embeddings with a common ML algorithm could provide similar or even better results, or without trying to freeze some pre-trained layers and train the rest ones.

Concisely, most studies in the VP field that use pre-trained models present an approach without neither explaining nor justifying their choice to use a specific model architecture, a PL, NL or mixed PL and NL - pre-trained model, and a word embedding, sentence embedding or fine-tuning approach. Usually, they do not even specify which pre-trained model variant they use (e.g., unimodal or bimodal CodeBERT). To this end, contrary to previous studies that focused solely on proposing a Transformer-based VPM without examining the implementation choices that lead to improved accuracy, the current study examines different approaches that can be followed during the implementation of transfer learning-based VPMs and attempts to shed some light regarding the suitability and the accuracy of the different variations in the employment of pre-trained

Transformer-based solutions for VP. Through this process, the best possible approach for applying transfer learning in the field of VP emerges. Table 1 summarizes all the aforementioned related studies that present VPMs in terms of dataset, code representation format, model, evaluation metrics and, in case of Transformer-based models, the implementation choices made to build VPMs using transfer learning. We label the studies by the name of the proposed model or, if there is no name, by the names of the authors.

Table 1: Summary of the various related studies presenting vulnerability prediction models.

| **Name** | **Dataset** | **Representation** | **Model** | **Eval. Metric** | **Implem. Choice** |
|---|---|---|---|---|---|
| Shin and Williams [26],[27] | Firefox JS Engine | Software metrics | Statistical correlation | Accuracy, Recall, FNR | N/A |
| Chowdhury and Zulkernine [28] | Mozilla Firefox | Software metrics | Decision Tree | $F_1$-score | N/A |
| Kalouptsoglou et al. [29] | PHP Drupal, PHPMyAdmin, Moodle [4] | Software metrics | MLP | Recall | N/A |
| Scandariato et al. [5] | Android OS Platform | BoW | Random Forest | $F_2$-score | N/A |
| Hovsepyan et al. [30] | K9 mail client | BoW | Support Vector Machine | Accuracy Recall, Precision | N/A |
| Dam et al. [6] | Android OS Platform, Firefox | Sequence of tokens | LSTM | $F_1$-score | N/A |
| VulDeePecker [8] | NVD [45] + SARD [46] | Sequence of tokens | word2vec+ BiLSTM | $F_1$-score | N/A |
| Kalouptsoglou et al. [9] | NVD [45] + SARD [46] | Sequence of tokens | word2vec+ CNN/LSTM | $F_2$-score | N/A |
| SySeVR [31] | NVD [45] + SARD [46] | Sequence of tokens, AST, PDG | word2vec+ BiGRU | $F_1$-score | N/A |
| Devign [11] | Devign [11] | Graph | GNN | $F_1$-score | N/A |
| ReVeal [12] | ReVeal [12] | Graph | GNN | $F_1$-score | N/A |
| Bagheri et al. [20] | Python GitHub [20] | Sequence of tokens | CodeBERT+ LSTM | $F_1$-score | Word embedding extraction |
| Yuan et al. [41] | NVD [45] + SARD [46] | Sequence of tokens | CodeBERT | Precision, Recall | Word embedding extraction |

| Name | Dataset | Representation | Model | Eval. Metric | Implem. Choice |
|---|---|---|---|---|---|
| VulDeBERT [42] | NVD [45] + SARD [46] | Sequence of tokens | BERT | $F_1$-score | Fine-tuning |
| VulBERTa [43] | Draper [47], CodeXGLUE [22], D2A [48] | Sequence of tokens | RoBERTa | $F_1$-score | Fine-tuning, Sentence embedding extraction |
| Purba et al. [44] | CVEfixes [49] | Sequence of tokens | CodeGen, GPT-3.5, Davinci | $F_1$-score | Fine-tuning |
| LineVul [21] | Big-Vul [50] | Sequence of tokens | CodeBERT | $F_1$-score | Fine-tuning |
| Present study | Big-Vul [50] | Sequence of tokens | CodeBERT, CodeGPT-2 | $F_1$-score | Fine-tuning, Word embedding extraction, Sentence embedding extraction, Data modalities |

## 3. Study design

In this section, the entire methodology of the current study is described. Initially, we define the research questions that outline the analysis and explain the selection of the implementation choices examined. Then we present the overview of our methodology and subsequently we provide details for the dataset, the utilized models, the embeddings (i.e., features) - based approaches, the fine-tuning method as well as the training and evaluation procedures we followed.

### *3.1. Research questions definition*

The research goals of the current study can be expressed through the following Research Questions (RQs):

- **RQ$_1$**: What is the impact of different transfer learning strategies on the performance of Transformer-based models for vulnerability prediction?

  RQ$_1$ investigates whether it is better for vulnerability prediction to fine-tune the pre-trained models on this specific task (i.e., fine-tuning approach) or extract their features and learn to classify them (i.e., feature-based approach). In addition, it compares the two feature-based approaches by investigating whether it is better to extract the sentence embeddings to represent the input and train a classifier or to extract the pre-trained word embeddings and train a separate DL predictor.

- **RQ$_2$**: What are the computational trade-offs between different transfer learning strategies in constructing Transformer-based vulnerability prediction models?

  RQ$_2$ aims at providing insights into trade-offs between performance and computational factors such as training time, memory requirements, model complexity, and inference time, when selecting between fine-tuning and feature-based approaches for constructing Transformer-based VPMs.

- **RQ$_3$**: How does pre-training on both natural and programming languages impact vulnerability prediction performance compared to pre-training on code alone?

  RQ$_3$ is responsible for determining whether transfer learning from models pre-trained in both natural and programming languages (i.e., bimodal pre-training) enhances vulnerability prediction as opposed to models pre-trained exclusively in programming languages (i.e., unimodal pre-training).

- **RQ$_4$**: How do context-aware embeddings compare to traditional static embeddings in vulnerability prediction?

  RQ$_4$ examines the impact of the context-aware embedding vectors, which are extracted from Transformer-based models, on the implementation of vulnerability prediction solutions compared to traditional techniques that have been utilized for embedding the source code in static vectors (i.e., global vectors), thereby extracting insightful observations regarding the context-awareness of LLMs.

- **RQ$_5$**: How does the best-performing transfer learning approach compare against state-of-the-art vulnerability prediction approaches?

  RQ$_5$ compares the optimal model, as identified in the previous RQs, in contrast to some well established state-of-the-art approaches, which are based either on text mining or on graphical representations.

### *3.2. Selection of implementation choices*

As the primary motivation of this study is to shed some light in the fragmented literature on Transformer-based VP, we explore how specific implementation choices in constructing Transformer-based VPMs influence model accuracy. It is important to note that our study does not aim to analyze all possible transfer learning implementation choices and their combinations, but we focus on the key approaches that have already been employed in the VP-related literature.

To the best of our knowledge, no prior research has systematically evaluated the impact of different implementation choices on Transformer-based VPMs, and thus, in the current empirical study we focus on providing recommendations on the fundamental transfer learning strategies that have been previously utilized in VP research, as discussed in Section 2. In particular, we compare the fine-tuning [21],[42],[43],[44] and feature-based [20],[41],[43] (both sentence-level and word-level feature extraction) approaches, which represent fundamental transfer learning methodologies for adapting Transformer models to downstream tasks.

Concisely, we based the selection of implementation choices on whether they are fundamental in transfer learning, and whether they have already been used in the VP-related literature, taking also into account the specific characteristics of the field. Hence, we decided to proceed with the comparison of the fundamental fine-tuning and feature-based directions, which have shown promising performance in VP, as well as by evaluating a domain-specific option regarding the benefit of incorporating prior knowledge of NL compared to prior knowledge of source code data solely. This evaluation allows us to derive actionable insights for practitioners and researchers interested in building Transformer-based models for VP.

### *3.3. Methodology*

This section provides a thorough presentation of the current study's entire methodology. In Figure 1 there is the high-level overview of our implementation strategy. It includes three main phases, namely data preparation, training setup, and model testing and comparison. More details about the steps of these phases are provided in the following subsections.

#### *3.3.1. Data preparation*

Regarding the utilized dataset, we used the Big-Vul dataset [50] that consists of source code retrieved from public repositories found on GitHub, all written in the C/C++ programming languages. This dataset contains real world vulnerabilities that have been reported in the Common Vulnerabilities and Exposures (CVEs) database.

For the construction of Big-Vul, Fan et al. [50] developed a tool capable of crawling the public CVE database to gather all the available information of a CVE, including the references linking to the relevant software products, which enabled them to search the products with open-source git repositories [50]. This way, they found vulnerability-related commits and extracted the corresponding changes in the source code in order to get the changed parts between before and after repairing a vulnerability. The parent version of these commits (i.e., version before repairing) is the one that contains the reported vulnerability.

Through this process, they managed to create a dataset of 188,636 samples collected from 348 open-source software projects with several vulnerability types included. Specifically, the dataset has 10,900 vulnerable methods along with their fixes and 177,736 other neutral (i.e., non-vulnerable) methods that can be considered as clean since no vulnerability has been reported for them yet. Hence, the dataset has a ratio of vulnerable methods equal to 6.13 %.

After fetching Big-Vul dataset, we proceeded with data cleansing. For this purpose, we examined the Croft et al. [51] study, which conducts a quality assessment to remove noise in the data of some well established vulnerability datasets including Big-Vul. Based on their findings, the *Chromium* project had to be removed from the dataset as it was found to reduce the accuracy of the overall dataset by leading the vulnerability prediction models to infer incorrect patterns between classes. In particular, Croft et al. noticed that most of the vulnerability reports related to *Chromium* were improperly traced since its repository is not naturally hosted by GitHub. They validated their observation by showing a large drop in vulnerability prediction models accuracy when removing noisy data from the testing set, indicating to incorrect patterns learnt during the training phase. Therefore, we proceeded to the removal of *Chromium* from both the training and evaluation sets in order to avoid introducing noise when constructing our models.

Next, we separated the dataset randomly in three different parts i.e., (i) training, (ii) validation, and (iii) testing parts, following a common evaluation technique in ML

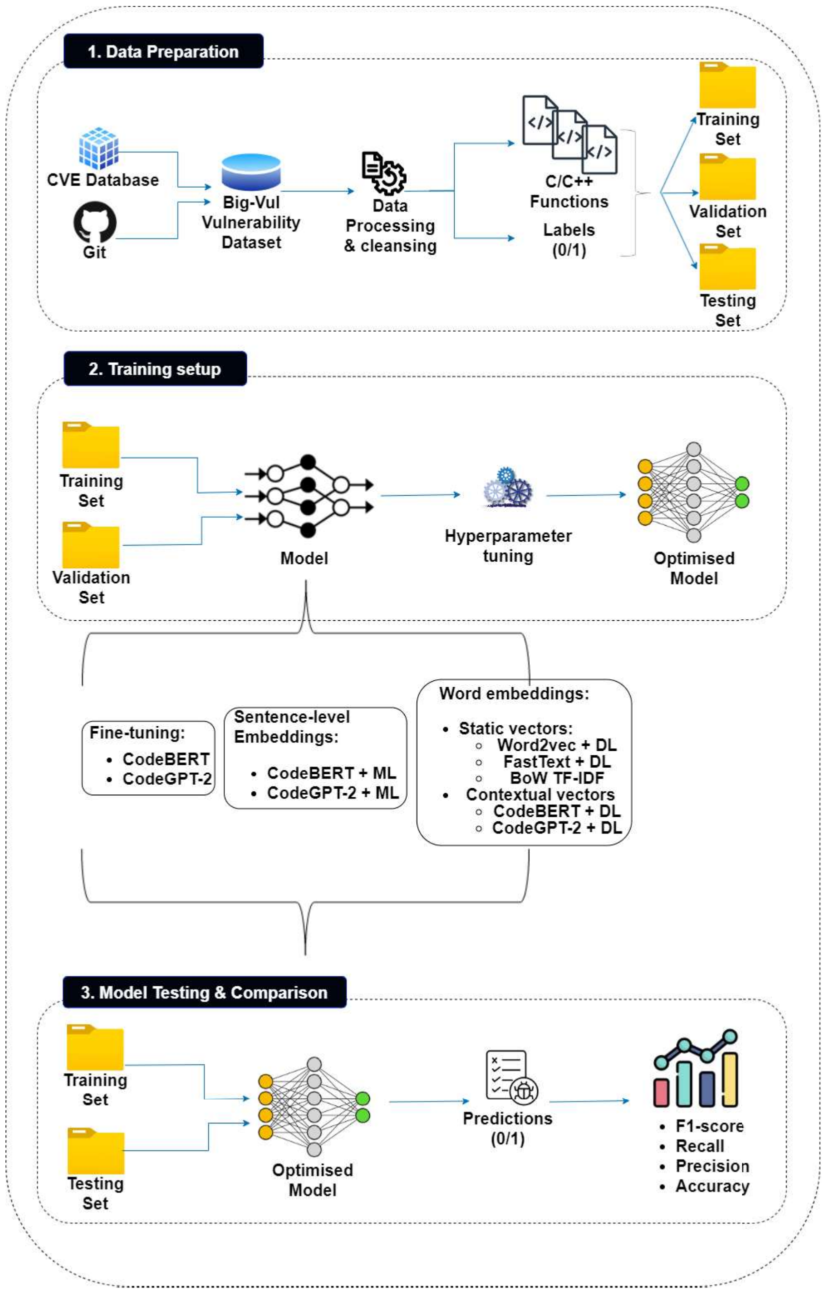


Figure 1: Overview of the overall approach.

and in VP specifically [2]. Specifically, we divided the entire dataset into two sets; one for training and validation and one as a completely unseen set for testing purposes in percentages of 90-10 %. We then divided the large set of 90 % further into training and validation sets, again in percentages of 90-10 %. The training dataset was utilized for the training of the examined models on the vulnerability-related data. The validation set was utilized for evaluating the produced models during the hyperparameter selection phase, whereas the testing set was used as a testbed for the final evaluation of the analyzed approaches, and therefore, for the models' comparison and the conclusions of our analysis.

#### *3.3.2. Training setup*

In the training phase of the methodology, as can be seen in Figure 1, we fed the input training data in the examined models in order to optimise them (i.e., find the optimal hyperparameters), based on their performance on the validation data, and therefore, produce the optimal model per each examined approach. In Figure 1, there are mentioned the three different examined approaches for leveraging pre-trained models in VP; (i) fine-tuning, (ii) sentence-level embeddings extraction, and (iii) word embeddings extraction.

For all the examined transfer learning approaches, we leveraged the pre-trained on code variants of BERT and GPT-2 models, namely as CodeBERT and CodeGPT-2, which have demonstrated promising results for code-related objectives [23],[21],[52]. CodeGPT-2 is based on the GPT-2, which is the latest open-source version of the GPT model that was developed by OpenAI[2]. GPT-2 employs the decoder part of the Transformer architecture and is pre-trained on the primary task of next word prediction. It is more suitable for text generation tasks. In *CodeXGLUE* study [22], Lu et al. presented CodeGPT-2, a variant of GPT-2, which was pre-trained on PL data retrieved from the Java and Python sets of the *CodeSearchNet*[3] dataset. This model has the same decoder-only Transformer architecture and pre-training objective (i.e., next word prediction) as the GPT-2 but has prior knowledge of PL. Specifically, there are two versions of it; one that is pre-trained from scratch on PL[4] and another which uses as a basis the GPT-2 weights and continues its training on the code corpus (i.e., domain adaptive model)[5]. To distinguish them, the latter is mentioned as *CodeGPT-2-Adapted*.

On the other hand, CodeBERT is an encoder-only architecture, which, as its name implies, belongs to the BERT variants. BERT was originally pre-trained on the MLM objective. In MLM, the 15% of the tokens sequences is masked and then the model learns to predict the actual values of the masked tokens. An improved variant of BERT is RoBERTa, which is trained on a much larger dataset with a more effective training approach. The architecture of the RoBERTa model is the basis for CodeBERT, which was developed by Microsoft AI. CodeBERT has two variants: (i) the *CodeBERT-base* model[6], which is pre-trained on natural and programming language pairs (i.e., bimodal data), and (ii) the *CodeBERT-base-MLM* model[7], which is pre-trained on source code

[2]https://openai.com/
[3]https://huggingface.co/datasets/code_search_net
[4]https://huggingface.co/microsoft/CodeGPT-small-py
[5]https://huggingface.co/microsoft/CodeGPT-small-py-adaptedGPT2
[6]https://huggingface.co/microsoft/codebert-base
[7]https://huggingface.co/microsoft/codebert-base-mlm

data in the task of predicting the masked tokens in code fractions. The former is pre-trained on documentation and code pairs of the *CodeSearchNet* dataset, whereas the latter is pre-trained on the code corpus of the *CodeSearchNet* dataset.

For the examined Transformer-based models, we used the *Transformers* library that is provided by Hugging Face (HF)[8]. This library provides implementations of several pre-trained NLP models, including CodeGPT-2, and CodeBERT, and also their pre-trained weights. We used the HF library to load the aforementioned models in order both to fine-tune the pre-trained models in the downstream task of VP and to extract their sentence-level or word-level embeddings.

### *3.3.3. Fine-tuning experiments*

To begin with, fine-tuning in VP is the process of training on a labeled dataset both the layers of a pre-trained model and an extra classification layer placed on the top of the existing model (classification head). In fine-tuning, the model adapts its prior language knowledge on a specific objective. In this study, we fine-tuned CodeBERT and CodeGPT-2 on the objective of VP, using the vulnerability dataset described in Section 3.3.1. During fine-tuning, the training dataset, in the format of sequences of tokens, is fed to the VPM that consists of the pre-trained Transformer-based model and the classification head, and then, both of them are trained and their weights are updated in order to minimize a loss function, which is the Cross-Entropy loss[9]. Finally, the models learn to classify software components as vulnerable or not.

CodeBERT and CodeGPT-2 were fine-tuned using their default Transformer architecture as provided by HF. Specifically, CodeBERT, similarly to RoBERTa, has 12 Transformer layers with 768 hidden size, 12 attention heads and a total of 125 millions parameters. CodeGPT-2, similarly to GPT-2, has also 12 layers, 768 hidden size, and 12 attention heads, but it has 117 millions parameters. We added a classification head, and, then we proceeded with manual hyperparameter tuning based on empirical observations, using as a starting point the default hyperparameter settings recommended in the original studies of CodeBERT [23] and CodeGPT-2 [22], in order to determine the values of the optimization hyperparameters, ending up with a learning rate (LR) set as 0.00002 (i.e., 2e-5) along with a linear scheduler where the LR decays linearly during the training procedure. For the optimization of the gradient descent, we used AdamW (Weighted Adam) optimizer [53]. The sequences of the input had a maximum length equal to 512, the longest length they are capable of supporting. We employed also the Early Stopping technique to determine the number of epochs. In addition, zero padding was used to ensure that each sequence had the same length throughout the encoding of the textual data using CodeBERT and CodeGPT-2 tokenizers, and the truncation approach was used to trim sequences that exceeded the maximum length. Table 2 summarizes the aforementioned characteristics of the fine-tuned CodeBERT and CodeGPT-2 models.

### *3.3.4. Sentence-level embedding extraction experiments*

Apart from fine-tuning, we leveraged those LLMs in vulnerability prediction by following feature-based approaches. Specifically, we fed to them the input data and we

[8]`https://huggingface.co/`

[9]`https://en.wikipedia.org/wiki/Cross-entropy`

Table 2: Characteristics of the models considered in the fine-tuning and sentence-level embedding extraction approaches.

| **Attribute** | **CodeBERT** | **CodeGPT-2** |
|---|---|---|
| Versions | Base, Base-MLM | Base, Adapted |
| Transformer Variant | RoBERTa | GPT-2 |
| Transformer Layers | 12 | 12 |
| Fully Connected Layers | 1 | 1 |
| Hidden Size | 768 | 768 |
| Attention Heads | 12 | 12 |
| Learning Rate (LR) | 0.00002 | 0.00002 |
| Optimizer | AdamW | AdamW |
| Loss Function | Cross-Entropy | Cross-Entropy |
| Max Length | 512 | 512 |

extracted the features of the last hidden layer (i.e., sentence-level embeddings). The extracted embeddings were given then as input to an ML classifier, which actually is a classification layer placed on top of the Transformer model and is called classification head. During training, the pre-trained layers froze and the classification head learnt to classify functions as vulnerable or not.

#### *3.3.5. Word-level embedding extraction experiments*

Furthermore, we employed LLMs by extracting their pre-trained word embedding vectors. Word embedding is a method that represents words as vectors of real numbers. These word vectors capture semantic and syntactic information about words, enhancing the capabilities of ML models to learn textual data. A popular word embedding algorithm is word2vec [10], which is the method most commonly used in the VP field [2]. However, LLMs have proposed a different embedding approach by producing context-aware embedding vectors. During this approach, the way words are used in sequences varies based on their context. This results in a word having different vector representations depending on the context. In our analysis, we transformed the sequences of source code tokens to sequences of contextual (CodeBERT or CodeGPT-2) embedding vectors. These sequences were the input to a DL model, which was trained in binary classification to predict whether a sequence has vulnerabilities.

For the selection of the DL classifier, we examined various DL algorithms. We focused mainly on the RNNs, since they are the most capable neural networks (except Transformers) for learning sequential data similar to language data. During training, we used the validation data in order to configure the optimal hyperparameters of the DL models. After an extensive hyperparameter tuning employing the Grid-search approach [54], we ended up with Adam optimizer, learning rate equal to 0.001, and batch size 64. We applied also the Early Stopping technique to determine the number of epochs before the models start to overfit. To avoid overfitting, we also utilized dropout layers in all layers (both hidden layers and dense output's dense layer). For the initialization of the weights, the Xavier Initialization was used. We put three recurrent layers (with 500-100-200 nodes respectively) along with the tanh activation function. The loss function utilized was the binary cross entropy with the Sigmoid activation function in the last

Table 3: Optimal configurations of the deep learning model used in the word-level embedding extraction approach.

| **Configuration** | **Value** |
|---|---|
| Embedding Model | CodeBERT & CodeGPT-2 |
| Embedding Type | Contextual |
| Embedding Size | 768 |
| DL Classifier | BiGRU |
| Recurrent Layers | 3 |
| Hidden Size | 500 - 100 - 200 |
| Initializer | Xavier |
| Optimizer | Adam |
| Learning Rate (LR) | 0.001 |
| Activation Function | tanh |
| Output Activation Function | Sigmoid |
| Loss Function | Cross-Entropy |
| Dropout per Layer | 0.2 - 0.1 - 0.1 |

layer. We also experimented with different kinds of recurrent layers. In particular, we compared several RNN variants including LSTM, Gated Recurrent Unit (GRU), Bidirectional LSTM (BiLSTM), and Bidirectional GRU (BiGRU). We also examined 1-D CNNs since they have demonstrated competent performance in VP [2],[3],[9]. It turned out that BiGRU is the best model for our case, at least for Big-Vul. A summary of the aforementioned configurations of the DL classification model is provide in Table 3.

*3.3.6. Traditional word embedding experiments*

Finally, we used also traditional embedding methods in order to facilitate direct comparison of context-aware embeddings versus static (i.e., global) embeddings, such as word2vec ones. Static vectors represent each word in a unique matter, regardless their particular context. To train static embeddings such as the word2vec vectors, we utilized the training set of our dataset, while the validation set was used as a basis for selecting the hyperparameters of the word embedding algorithms by employing the Grid-search technique [54]. We applied the word2vec algorithm on this set in order to learn source code-aware word2vec static embedding vectors. We then fed the sequences of word2vec vectors to the DL model and we trained it to classify the sequences to vulnerable and non-vulnerable. We repeated the whole process by employing also the fastText algorithm, which, although still produces static vectors, it manages to tokenize also out of vocabulary words, since it acts on sub-word and character level. To train the static word embeddings on C/C++ data, the vector dimension, after several experimentations, was chosen equal to 100 and the context window size equal to 20 words. We utilized Skip-gram and Continuous Bag-of-Words (CBOW) variants for word2vec and fastText respectively. Last but not least, for reasons of completeness, we proceeded also with the state-of-the-art BoW representation. For the BoW approach, we utilized the Random Forest classifier, which has been widely adopted in the literature of VP [2],[4],[21], often demonstrating high accuracy.

### *3.3.7. Model testing and comparison*

For the evaluation of all the examined techniques, we quantify the produced results using common classification metrics. In particular, we employed accuracy, recall, precision, $F_1$-score, and $F_2$-score. We consider as the most critical measurement for our analysis the $F_1$-score, which considers equally both recall and precision and therefore consists a measurement capable of reflecting the attempt to both increase the actual vulnerabilities identified and to decrease the false positives. It is also the most utilized metric in the VP-related literature [2] and thereby facilitates the comparison with other studies. The mathematical formula of $F_1$-score is provided below:

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

, where TP stands for True Positives, FP for False Positives, and FN for False Negatives.

At this point, we have to notice that the experiments (i.e., both training and evaluation processes) for each different examined approach were repeated ten times by using a different seed each time, and their average values were calculated and reported. This setting prevents our analysis from depending on the randomness that exists in various processes during data shuffling, model training, computational calculations, etc.

## 4. Results

In this section, the study's experimental results are presented with respect to the research questions defined in Section 3.1. All the experiments took place on the CUDA[10] platform of a GeForce RTX 4080 Super Nvidia Graphics Processing Unit (GPU). The results provided are derived from the testing part of the dataset. To facilitate the reproduction of the results, we also provide a replication package[11].

### *4.1. $RQ_1$ - Most effective strategy for leveraging pre-trained Transformer-based models in vulnerability prediction*

In $RQ_1$ we examined the transfer learning approaches that one can follow to leverage LLMs for the downstream task of VP. To this end, we employed the CodeBERT and the CodeGPT-2 models. First, we fine-tuned in VP both of the pre-trained on code *CodeBERT-base-MLM* and *CodeGPT-2* models by updating the weights of all their layers to adapt the models to this particular task. Next, we extracted their embeddings at either sentence or word level and trained DL classification models in VP.

In case of sentence embeddings, we fed the input data to pre-trained models, subsequently, we froze their pre-trained layers, and therefore the given sequences of tokens were encoded as sentence-level embeddings. Those embeddings were given to the classification head of the models, which was then trained on binary classification to discriminate vulnerable and non-vulnerable functions. In case of word embedding, we extracted the

[10] `https://developer.nvidia.com/cuda-toolkit`
[11] `https://sites.google.com/view/vulgpt/`

word embedding vectors from the pre-trained models and provided them to the embedding layer of a separate neural network. This way, we utilized the pre-trained models to gain their prior knowledge so as to represent the source code tokens. We then trained the neural network on the binary classification task of VP. Specifically, we trained a BiGRU model, which proved to be the best one among the examined RNNs and the CNN on the Big-Vul dataset, as described in Section 3.3.5.

Table 4 summarizes the evaluation results of the three approaches on the testing set. Particularly, it presents the average values of ten different repetitions of the evaluation per case. As can be seen, the word embedding extraction approach is the clear winner among the two feature-based methods. Specifically, word embedding extraction from CodeBERT and CodeGPT-2 leads to a VPM of $F_1$-score equal to 91.4% and 90.2% respectively, whereas the sentence embedding extraction from these models achieves 67% and 56.8% $F_1$-scores, which are much lower values. It seems that, by just training the classification head, the models do not manage to capture efficiently the vulnerability patterns.

Table 4: Evaluation results of fine-tuning and feature-based (sentence and word-level embedding extraction) approaches of transfer learning in vulnerability prediction.

| **Model / Metric (%)** | **Accuracy** | **Precision** | **Recall** | **$F_1$-score** | **$F_2$-score** |
|---|---|---|---|---|---|
| CodeBERT fine-tuning | 99.0 (±0.06) | 96.3 (±0.9) | 87.4 (±1.1) | 91.6 (±0.6) | 89.0 (±0.8) |
| CodeGPT-2 fine-tuning | 98.8 (±0.04) | 96.0 (±0.7) | 85.2 (±0.7) | 90.3 (±0.3) | 87.2 (±0.5) |
| CodeBERT sentence-level | 96.8 (±0.91) | 98.3 (±2.5) | 50.8 (±3.1) | 67.0 (±3.5) | 56.2 (±4.1) |
| CodeGPT-2 sentence-level | 96.0 (±0.12) | 89.8 (±3.0) | 41.5 (±3.8) | 56.8 (±4.0) | 46.6 (±4.5) |
| CodeBERT word-level | 98.9 (±0.04) | 96.0 (±1.0) | 87.4 (±1.3) | 91.4 (±0.4) | 88.9 (±0.9) |
| CodeGPT-2 word-level | 98.8 (±0.06) | 96.4 (±0.8) | 84.8 (±0.9) | 90.2 (±0.5) | 86.9 (±0.7) |

As can be seen in Table 4, the word embedding extraction approach and the fine-tuning approach, are really close to each other, both when using CodeBERT and CodeGPT-2. Specifically, CodeBERT fine-tuning achieves an average $F_1$-score equal to 91.6% with a standard deviation of 0.6%, in contrast to CodeBERT word embedding that achieves $F_1$-score 91.4% with standard deviation 0.4%. Similarly, CodeGPT-2 fine-tuning manages to achieve an $F_1$-score equal to 90.3% on average, with standard deviation equal to 0.3%, as opposed to CodeGPT-2 word embedding approach, which succeeds $F_1$-score 90.2% with a standard deviation equal to 0.5%. We can discern a very slight lead of the fine-tuned models but we can not single out just one as the best.

To facilitate the comparison, we proceeded with conducting a statistical test so as to identify whether there is a statistical significance in the superiority of the fine-tuning approach. Specifically, we performed the Wilcoxon-Signed Rank Test [55], which can judge whether there is a statistically significant difference between two pairs (i.e., $F_1$-scores achieved by fine-tuning and by word embeddings extraction). For this purpose, we used all the ten $F_1$-scores computed by each model and each approach, and we utilized them as pairs based on the seed's value. Based on the Wilcoxon analysis, we found that the p-values were 0.49 and 0.84 for CodeBERT and CodeGPT-2 respectively, which are higher than the 0.05 threshold, and therefore we cannot state that there is a statistically significant difference between the two approaches. Concisely, the results of the analysis suggest that:

**Fine-tuning and word embedding extraction strategies achieve comparable predictive performance in vulnerability prediction, while word embedding extraction is the most accurate feature-based approach, outperforming sentence embedding extraction.**

### *4.2. $RQ_2$ - Computational trade-offs between different transfer learning strategies in vulnerability prediction*

To provide a comprehensive evaluation of Transformer-based VPMs, we analyze the computational trade-offs associated with different transfer learning strategies. While $RQ_1$ focused on prediction accuracy, this section examines key computational factors, such as training time, memory requirements, implementation complexity, and execution speed. Specifically, Table 5 presents the values of training time in seconds (s), GPU memory consumption in gigabytes (GB), model size on disk in GB, number of trainable parameters (i.e., indicator of model complexity), and inference time in seconds.

Table 5: Trade-offs between fine-tuning and feature-based (sentence and word-level embedding extraction) approaches in vulnerability prediction in terms of training time, memory requirements, model complexity, and inference time.

| **Model/Metric** | **Train. time** | **GPU memory** | **Disk space** | **Train. params** | **Inf. time** | **$F_1$-score** |
|---|---|---|---|---|---|---|
| CodeBERT fine-tuning | 7,091 | 8.20 | 1.46 | 124,647,170 | 0.00572 | 91.6 |
| CodeGPT-2 fine-tuning | 11,336 | 10.90 | 1.45 | 124,245,504 | 0.00770 | 90.3 |
| CodeBERT sentence-level | 3,894 | 1.94 | 0.50 | 592,130 | 0.00574 | 67.0 |
| CodeGPT-2 sentence-level | 7,550 | 2.30 | 0.50 | 1,536 | 0.00775 | 56.8 |
| CodeBERT word-level | 1,515 | 4.68 | 0.66 | 4,954,801 | 0.00147 | 91.4 |
| CodeGPT-2 word-level | 4,883 | 4.68 | 0.66 | 4,954,801 | 0.00146 | 90.2 |

As can be seen in Table 5, fine-tuned models require the most training time, GPU memory, disk space, and number of trainable parameters, as they modify all Transformer layers during training. On the other hand, the sentence embedding extraction approach has the smallest GPU memory and disk space as well as the fewest trainable parameters, since it only trains the classification head, which is a feed-forward layer added on top of the Transformer. Word embedding extraction, while requiring more parameters to be trained and more GPU memory than sentence embeddings, still has substantially fewer trainable parameters than fine-tuning and requires much less training time than both fine-tuning and sentence-level embeddings.

In particular, the CodeBERT word embeddings method needed 1,515 seconds (less than half an hour of training) approximately to be completed, and CodeBERT sentence embedding extraction needed 3,894 seconds (about 1 hour of training), whereas fine-tuning CodeBERT required 7,091 seconds (almost 2 hours of training). Significant differences exist also in the CodeGPT-2 case. The CodeGPT-2 word embeddings-based training of the RNN lasted 4,883 seconds (about 81 minutes), the sentence embedding extraction needed 7,550 seconds (about 2 hours), while CodeGPT-2 fine-tuning needed 11,336 seconds (more than 3 hours). CodeGPT-2 training completed in more time than CodeBERT, since it usually needed more epochs until reaching the optimal $F_1$-score. In any case, both models' fine-tuning proved to be a much more time-consuming process than embedding extraction (i.e., training of classification head) and word embedding extraction (i.e., DL model's training).

Furthermore, as regards the memory requirements, CodeBERT sentence embedding extraction requires 1.94 GB of GPU memory to feed the vectors and train the classification head, and 0.5 GB for hosting the trained model, while CodeBERT word embedding extraction needs 4.68 GB in GPU for training the DL classifier and 0.66 GB disk space for hosting the model. In contrast, CodeBERT fine-tuning requires 8.2 GB of GPU memory and 1.46 GB on disk. Similarly, in case of CodeGPT-2, sentence embedding extraction requires 2.3 GB in GPU and 0.5 GB on the disk, word embedding extraction approach requires 4.68 GB of GPU memory and 0.66 GB for hosting the trained model, while fine-tuning demands 10.9 GB of GPU memory and 1.45 GB disk space.

Moreover, the CodeBERT sentence embedding extraction needs training of 592,130 parameters and CodeBERT word embeddings method includes 4,954,801 trainable parameters that need to be optimized during training, whereas the CodeBERT fine-tuning approach requires the training of 124,647,170 parameters, which is a significantly higher number. In case of CodeGPT-2, the sentence embedding extraction approach includes 1,536 trainable weights and the word embeddings extraction needs to update 4,954,801 trainable parameters (i.e., same number as in CodeBERT word embeddings, since the trainable parameters regard the DL model, which is common) in contrast to the more complex fine-tuning approach, which includes 124,245,504 trainable weights.

As regards the inference time, which is a crucial factor for real-world deployment, all the three techniques exhibit quite short times. However, a substantially lower inference time is observed for the word embedding extraction approach, making it particularly suitable for real-time security applications. Such a low inference time demonstrates also the advantage of word embedding extraction approach in terms of scalability in relation with the size of the analyzed projects. Considering that it needs on average 1.47 ms to analyze a function under test, it will be able to analyze an entire project of 100 functions in 147 ms, while for a project of 1000 functions it will need 1,470 ms. It is also capable of analyzing larger projects that consist, for instance, of 100,000 functions in only 147,000 ms, being resilient to the scale of the project's size. On the contrary, fine-tuning, which can analyze one function in 5.72 ms, needs 5,720 ms to analyze a project of 1000 functions, and 572,000 ms for a larger project of 100,000 functions, which are substantially higher values than those of word embedding extraction. This observation also highlights the advantage of transfer learning approaches, and in particular word embedding extraction, over static code analysis techniques, since static code analyzers, which are traditionally used to scan software projects for vulnerabilities, often require a considerable amount of more time to analyze large codebases [56],[57] and, therefore, often run in nightly builds (i.e., no actual working time).

Overall, feature-based approaches achieve lower computational costs compared to fine-tuning. Although fine-tuning provides a good predictive performance, it is the most computationally expensive approach of the three considered. On the other hand, sentence embedding extraction is the most lightweight choice, but it suffers from a significant drop in accuracy metrics compared to fine-tuning and word embedding extraction approaches. Given its balance between efficiency and accuracy, word embedding extraction emerges as the most practical choice. Concisely, regarding the question of what are the computational trade-offs, the findings of the analysis of $\mathrm{RQ}_2$ suggest that:

**Word embedding extraction presents the optimal trade-offs between predictive performance and computational footprint compared to fine-tuning and sentence embedding extraction approaches. Additionally, sentence embedding extraction is the most lightweight approach in terms of memory requirements and complexity, while word embedding extraction is the fastest in both the training and inference phases.**

Finally, considering both the results of $RQ_1$ and $RQ_2$, we can notice that word embedding extraction and fine-tuning approaches achieve a substantially higher accuracy than sentence embedding extraction. In addition, we notice that word embedding extraction achieves almost equal predictive performance with fine-tuning, but by requiring less training time, resources, and parameters to be trained. Moreover, it demonstrates faster inference. Therefore, concisely, we argue that:

**The most effective strategy to leverage transfer learning techniques in the field of vulnerability prediction is the feature-based approach of extracting the pre-trained word embedding vectors from code-oriented LLMs such as CodeBERT, use them to represent the sequences of source code tokens, and train a separate DL model specifically on the task of classifying functions as vulnerable or not.**

### *4.3. $RQ_3$ - Contribution of prior knowledge of natural language in vulnerability prediction*

After identifying in $RQ_1$ and $RQ_2$ the optimal way of using transfer learning from LLMs to VP, in the context of $RQ_3$, we examined whether it is preferable to use models pre-trained on unimodal data (i.e., exclusively on programming language) or pre-trained on bimodal data (i.e., both programming and natural language). For this purpose, we employed the fine-tuning and word embeddings extraction approaches presented in Table 4, where we used the unimodal versions of CodeBERT and CodeGPT-2, but we also repeated the experiments by retaining the NL knowledge of the models (i.e., bimodal models). Specifically, in the case of CodeBERT, we included in the analysis the bimodal *CodeBERT-base* variant, which is pre-trained on pairs of documents and code. In addition, in the context of $RQ_3$, for CodeGPT-2, we used the *CodeGPT-2-Adapted* version, which retains the pre-trained weights from the initial pre-training of GPT-2 on NL data and is further pre-trained using code data. Table 6, shows the values of the evaluation metrics for all these experiments.

The results shown in Table 6, which correspond to the average values of ten different repetitions of the evaluation per case, do not clearly demonstrate an optimal approach. In particular, unimodal CodeBERT achieves $F_1$-score 91.6% (±0.6%) and 91.4% (±0.4%) in fine-tuning and word embedding cases respectively, compared to $F_1$-score of 91.5% (±0.5%) and 91.3% (±0.3%) of the bimodal CodeBERT fine-tuning and word embedding respectively. Similarly, unimodal CodeGPT-2 achieves $F_1$-score 90.3% (±0.3%) and 90.2% (±0.5%) in fine-tuning and word embedding approaches, as opposed to $F_1$-score of 91.2% (±0.4%) and 90.5% (±0.2%) of the bimodal CodeGPT-2 fine-tuning and word embedding approaches. We can see that the $F_1$-scores (and the other metrics as well) are very close in unimodal and bimodal scenarios, in both CodeBERT and CodeGPT-2 models and in both fine-tuning and word embeddings approaches.

Therefore, we conducted the Wilcoxon-Signed Rank Test [55] to decide if there is a statistically significant difference. The p-values among unimodal and bimodal models

Table 6: Evaluation results of Transformer models pre-trained solely on source code versus ones pre-trained on bimodal data, when utilized for vulnerability prediction.

| Model / Metric (%) | Accuracy | Precision | Recall | $F_1$-score | $F_2$-score |
|---|---|---|---|---|---|
| unimodal CodeBERT fine-tuning | 99.0 (±0.06) | 96.3 (±0.9) | 87.4 (±1.1) | 91.6 (±0.6) | 89.0 (±0.8) |
| bimodal CodeBERT fine-tuning | 98.9 (±0.05) | 96.6 (±0.8) | 87.0 (±1.0) | 91.5 (±0.5) | 88.7 (±0.7) |
| unimodal CodeGPT-2 fine-tuning | 98.8 (±0.04) | 96.0 (±0.7) | 85.2 (±0.7) | 90.3 (±0.3) | 87.2 (±0.5) |
| bimodal CodeGPT-2 fine-tuning | 98.9 (±0.04) | 95.2 (±0.8) | 87.6 (±0.8) | 91.2 (±0.4) | 89.0 (±0.6) |
| unimodal CodeBERT embeddings | 98.9 (±0.04) | 96.0 (±1.0) | 87.4 (±1.3) | 91.4 (±0.4) | 88.9 (±0.9) |
| bimodal CodeBERT embeddings | 98.9 (±0.04) | 96.1 (±1.3) | 86.9 (±1.0) | 91.3 (±0.3) | 88.6 (±0.7) |
| unimodal CodeGPT-2 embeddings | 98.8 (±0.06) | 96.4 (±0.8) | 84.8 (±0.9) | 90.2 (±0.5) | 86.9 (±0.7) |
| bimodal CodeGPT-2 embeddings | 98.8 (±0.03) | 96.9 (±0.9) | 84.9 (±0.6) | 90.5 (±0.2) | 87.1 (±0.4) |

were 0.49 for the CodeBERT fine-tuning case, 0.002 for the CodeGPT-2 fine-tuning, 0.69 for CodeBERT word embeddings, and 0.81 for CodeGPT-2 word embeddings. In three of the four cases, the p-values were greater than 0.05, and therefore, we cannot state that there is a statistically significant difference. Only the case of fine-tuning CodeGPT-2 presents a statistically significant difference with a Wilcoxon p-value lower than 0.05. However, it happens only in one case and the absolute difference in $F_1$-score is quite low (i.e., just 0.9%). Hence, our study cannot yield any clear answer as to which of the bimodal and unimodal pre-trained models is better in VP. It can be argued that:

**The prior knowledge of natural language does not offer a clear benefit in vulnerability prediction but, neither does it act as noise. It can be considered neutral.**

### *4.4. RQ$_4$ - Benefit of context-aware embeddings compared to traditional static embeddings in vulnerability prediction*

In the context of RQ$_4$, we conducted a comparison of the best transfer learning VP approach that we identified in the previous RQs as opposed to other text mining-based approaches that leverage traditional word embedding techniques, which produce static vectors (i.e., a single global vector per word). As the best transfer learning method we qualified the unimodal CodeBERT word embedding extraction, which emerged as the optimal (i.e., most accurate and lightweight) one in the previous RQs, considering both the $F_1$-score and the computational trade-offs. As regards the traditional techniques, we chose word2vec and fastText as baselines due to their widespread use in VP research, with word2vec being the most commonly employed [2],[8],[9],[11],[20], while fastText, which addresses the out-of-vocabulary issue of word2vec, is also widely used in VP [2],[9],[20]. In addition, we employed the BoW text representation technique, which is also a widely used method for representing source code and is often used as a baseline for text mining-based VPMs [2],[3],[4],[21]. Our focus is to compare Transformer-based context-aware embeddings against traditional static embeddings, and, therefore, we included embedding techniques that are both static and word-level to ensure a fair comparison.

Table 7: Comparison of contextual and static word embeddings in vulnerability prediction.

| Model / Metric (%) | Accuracy | Precision | Recall | $F_1$-score | $F_2$-score |
|---|---|---|---|---|---|
| word2vec | 93.2 | 44.0 | 29.9 | 35.6 | 31.9 |
| fastText | 94.3 | 63.8 | 21.0 | 31.6 | 24.3 |
| BoW | 93.0 | 40.7 | 15.1 | 22.0 | 17.2 |
| CodeBERT embeddings | 98.9 | 96.0 | 87.4 | 91.4 | 88.9 |

Subsequently, we conducted an experiment to compare the same DL architecture and training paradigm (i.e., word embedding extraction and DL classifier) when (i) using word embeddings extracted from LLMs, and when (ii) employing word embeddings learnt by traditional text mining algorithms. This way, we can demonstrate the advantage of the prior knowledge of the LLMs, which is expressed through the context-aware embedding vectors. Specifically, we trained the word2vec and fastText models in the training part of the dataset, we encoded the input's sequences with word2vec and fastText vectors, we fed them to a DL model, and we trained it on VP. Table 7 presents the evaluation scores of the examined embedding approaches.

Based on Table 7, CodeBERT word embeddings demonstrated a much higher F1-score by 55.8%, 59.8%, and 69.4% as opposed to word2vec, fastText, and BoW respectively. Actually, they surpassed the static embeddings-based approach in all the 5 evaluation metrics. This observation indicates that there is an important benefit from the use of context-aware vectors instead of the global (i.e., static) ones. In other words, the ability of the Transformer-based models to learn, during the pre-training phase, the words syntax and semantics based on their context and to give words different vectors corresponding to their context, is an important factor that can enhance the performance of the text mining-based VPMs. Hence, the results suggest that:

**Transformer-based embeddings outperform traditional static embeddings, demonstrating the advantage of context-awareness in vulnerability prediction.**

### *4.5. $RQ_5$ - Comparison with other text mining-based and graph-based vulnerability prediction approaches*

The purpose of $RQ_5$ is to demonstrate whether there is an advantage of the best model (as identified in the previous RQs) as opposed to some of the most well-accepted, established, and referenced in the literature VPMs, which are based either on text mining (e.g, sequences of tokens) or on graphical representations of the source code (e.g., CPGs, ASTs, CFGs, etc.). To this end, we compared the CodeBERT word embedding extraction approach against 5 state-of-the-art DL-based VP approaches, namely VulDeePecker [8], SySeVR [31], Devign [11], ReVeal [12], and Linevul [21], which are often used as baselines in the current literature [12],[21],[58].

The aforementioned 5 models provide a comprehensive comparison between traditional text mining, Transformer-based text mining, and text-rich graph-based approaches. These methods represent key advances in the field, from LSTM-based text processing in VulDeePecker and leveraging program dependencies in SySeVR to graph-based learning in ReVeal and Devign as well as the use of LLMs in LineVul, covering diverse approaches to VP. Table 8 presents the evaluation results of the CodeBERT

Table 8: Comparison of CodeBERT word embedding extraction approach versus text mining-based and graph-based state-of-the-art models on Big-Vul dataset.

| **Approach** | **Precision (%)** | **Recall (%)** | **$F_1$-score (%)** |
|---|---|---|---|
| VulDeePecker | 12 | 49 | 19 |
| SySeVR | 15 | 74 | 27 |
| Devign | 18 | 52 | 26 |
| ReVeal | 19 | 74 | 30 |
| LineVul | 97 | 86 | 91 |
| CodeBERT embeddings | 96 | 87 | 91 |

word embedding extraction approach in contrast to the 5 baseline approaches. The experimental results of the baseline methods are based on the experiments conducted by Fu et al. [21].

As can be seen in Table 8, the word embeddings from the pre-trained CodeBERT model, fed in a DL classifier (i.e., transfer learning approach) managed to clearly surpass all the popular VulDeePecker, SySeVR, Devign, and ReVeal models. Although these models had presented important improvement over previous (non-transfer learning - based) VP approaches, CodeBERT word embedding extraction approach achieved an improvement of 72%, 64%, 65%, and 61%, in terms of $F_1$-score, over VulDeePecker, SySeVR, Devign, and ReVeal, respectively. One can observe the particular difficulty of these models to achieve high precision, which suggests their limited capacity to eliminate false positives. On the other hand, word embedding extraction achieved almost identical results with LineVul, which is the other Transformer-based method. This result is actually expected, since LineVul methodology is based on the CodeBERT model. In addition, it verifies the superiority of the Transformer-based solutions, also indicating that the benefits gained from the large prior knowledge and the contextual awareness might be a more valuable solution than graphical representations in VP.

Overall, the results presented in Table 8 not only enhance the argument that transfer learning provides a great benefit in VP, with transfer learning solutions outperforming state-of-the-art text mining-based ones, but also showcase that by using transfer learning, even solely text mining-based models manage to perform well in VP, achieving better results than even sophisticated graph-based models. Hence, we can argue that:

**Transformer-based transfer learning surpasses state-of-the-art vulnerability prediction models, including both text-mining and graph-based approaches.**

### *4.6. Results on other datasets*

To further validate the generalizability of our findings, we repeated our analysis using two additional open-source datasets. In particular, we employed FFmpeg+QEMU [11] and ReVeal [12] datasets, which have been both widely used in VP [12],[43],[51],[58],[59]. Table 9 presents the evaluation metrics of all the examined approaches. Specifically, it provides precision, recall, and $F_1$-score for both CodeBERT and CodeGPT-2 fine-tuning, sentence embedding extraction, word embedding extraction, and the bimodal alternatives (following the flow of the analysis on Big-Vul), as well as the text mining and graph-based state-of-the-art VPMs, including the traditional embedding algorithms.

Table 9: Evaluation results on the FFmpeg+QEMU and ReVeal datasets.

| **Dataset** | **Approach** | **Precision** | **Recall** | **$F_1$-score** |
|---|---|---|---|---|
| FFmpeg+QEMU | CodeBERT fine-tuning | 56 | 79 | 66 |
| | CodeGPT-2 fine-tuning | 58 | 66 | 62 |
| | CodeBERT sentence embeddings | 52 | 59 | 55 |
| | CodeGPT-2 sentence embeddings | 50 | 59 | 54 |
| | CodeBERT word embeddings | 55 | 79 | 65 |
| | CodeGPT-2 word embeddings | 59 | 71 | 64 |
| | Bimodal CodeBERT fine-tuning | 57 | 74 | 64 |
| | Bimodal CodeGPT-2 fine-tuning | 55 | 76 | 64 |
| | Bimodal CodeBERT word embeddings | 58 | 73 | 65 |
| | Bimodal CodeGPT-2 word embeddings | 58 | 68 | 63 |
| | Word2Vec | 53 | 48 | 51 |
| | FastText | 50 | 78 | 60 |
| | BoW | 51 | 56 | 54 |
| | VulDeePecker | 47 | 29 | 35 |
| | SySeVR | 48 | 66 | 56 |
| | LineVul | 57 | 74 | 64 |
| | Devign | 54 | 63 | 57 |
| | ReVeal | 55 | 73 | 62 |
| ReVeal | CodeBERT fine-tuning | 38 | 58 | 46 |
| | CodeGPT-2 fine-tuning | 32 | 66 | 43 |
| | CodeBERT sentence embeddings | 29 | 29 | 29 |
| | CodeGPT-2 sentence embeddings | 21 | 33 | 26 |
| | CodeBERT word embeddings | 37 | 65 | 47 |
| | CodeGPT-2 word embeddings | 35 | 59 | 44 |
| | Bimodal CodeBERT fine-tuning | 36 | 62 | 46 |
| | Bimodal CodeGPT-2 fine-tuning | 33 | 63 | 44 |
| | Bimodal CodeBERT word embeddings | 36 | 66 | 47 |
| | Bimodal CodeGPT-2 word embeddings | 34 | 63 | 44 |
| | Word2Vec | 30 | 57 | 39 |
| | FastText | 32 | 53 | 40 |
| | BoW | 33 | 48 | 39 |
| | VulDeePecker | 18 | 14 | 16 |
| | SySeVR | 24 | 40 | 30 |
| | LineVul | 39 | 57 | 46 |
| | Devign | 35 | 27 | 30 |
| | ReVeal | 31 | 61 | 41 |

To build the models based on word embedding extraction, we performed the DL model selection process again and ended up choosing the CNN model as the DL classifier for these two datasets, as opposed to Big-Vul, where we had chosen the BiGRU model. Furthermore, to compare against state-of-the-art models, we retrieved the ReVeal scores as they are provided by the ReVeal study [12]. In the case of Devign, we also reported its results as provided in ReVeal study, since both ReVeal and our analysis utilized only the FFmpeg and QEMU projects of the Devign dataset (i.e., half of the Devign projects), which were those provided as open-source by its authors[12]. Moreover, to facilitate a fair comparison, the scores of the VulDeePecker and SySeVR were also retrieved by the ReVeal study, which uses them as baseline methods. For LineVul, we proceeded with replicating it, since it is a later study than ReVeal.

By inspecting Table 9, we can see that, in both FFmpeg+QEMU and ReVeal datasets,

[12] `https://sites.google.com/view/devign`

and for both CodeBERT and CodeGPT-2 cases, the fine-tuning and word embedding extraction approaches produced quite similar results, clearly outperforming the approach of sentence embedding extraction. These findings align with our previous observations on Big-Vul dataset, strengthening the effectiveness of Transformer-based word embedding extraction, which has lower computational cost than fine-tuning. In addition, the bimodal models (pre-trained in both source code and natural language) did not show noticeable performance improvement over unimodal models (pre-trained only in source code), confirming our previous findings.

Moreover, Transformer-based word embeddings outperformed static word embedding techniques. In addition, although the predictive performance on FFmpeg+QEMU and (especially) ReVeal datasets is quite lower than in Big-Vul, Transformer-based transfer learning techniques still proved to be clearly more accurate than the VulDeePecker, SySeVR, and Devign baseline methods, and slightly more accurate than ReVeal. LineVul, achieved comparable results with the examined transfer learning approaches, as it is also based on the CodeBERT model, but with a higher computational cost than word embedding extraction, since it employs a fine-tuning strategy, which updates the parameters of all the Transformer layers.

Finally, the observation that the ReVeal model manages to be lower but close to the examined Transformer-based approaches indicates that a potential combination of textual and graphical representations of the source code will provide a real advantage in the VPMs. This could be achieved either as an unified Transformer-based model, which will have been pre-trained in a large amount of such multi-modal data, or by using ensemble learning of Transformers and graph-based models.

In brief, our findings suggest that word embedding extraction consistently outperforms sentence embedding extraction across all datasets, being the most accurate feature-based approach. Moreover, fine-tuning and word embedding extraction achieve comparable performance, with the latter requiring a much lower computational cost and, therefore, being identified as the optimal implementation choice. In addition, bimodal models do not offer a clear advantage over unimodal models. Furthermore, Transformer-based word embeddings consistently outperform static embeddings, demonstrating the importance of the context-aware embeddings, while Transformer-based transfer learning approaches show higher accuracy than both text mining-based and graph-based state-of-the-art VPMs, thus advancing the field of VP.

Hence, from the above analysis it is eminent that the same observations and conclusions can be reached with those of the Big-Vul dataset, strengthening in that way the generalizability of our findings and enhancing our confidence on the lack of potential bias imposed by the selected dataset.

## 5. Discussion and implications

In this paper, we set specific RQs about the capacity of the emerging LLMs on VP, examining the BERT and GPT-2 architectures, specifically their pre-trained on code variants namely CodeBERT and CodeGPT-2, paying particular emphasis on identifying the optimal implementation choices for leveraging transfer learning in VP tasks. The results of our experiments demonstrated several insights about the best practices for achieving high accuracy while maintaining computational efficiency. Therefore, in this section, we discuss several implications for both practitioners and researchers.

Table 10: Summary of the scenarios where each implementation choice is more suitable.

| **Implementation Choice** | **Best-Suited Scenarios** |
|---|---|
| Fine-tuning | Appropriate for high-accuracy environments where computational resources are not a constraint. Suitable for large-scale security applications and software organizations with high-end GPU infrastructure. |
| Sentence Embedding Extraction | Most computationally efficient approach, which is an efficient choice for resource-constrained environments where efficiency is prioritized over accuracy, such as in cases of rapid development of prototypes, and small-scale security projects. |
| Word Embedding Extraction | Approach that achieves both high accuracy in predicting vulnerabilities and high computational efficiency. Constitutes the best choice for resource-limited environments, where achieving high accuracy is still a priority. It is also a very effective method for large-scale security applications. |

Our findings suggest that extracting word embeddings from LLMs and feeding them to a separate classifier achieves better results than extracting sentence-level embeddings. Furthermore, this approach achieves equal results with the fine-tuning method, but requires a much shorter training time, substantially fewer trainable parameters, and much less GPU memory compared to fine-tuning. More specifically, fine-tuning achieves high scores in terms of accuracy metrics, but demands the most GPU memory, the largest disk space, and the longest training times. On the contrary, the feature-based approach of extracting sentence-level embeddings is the most computationally efficient approach, but lacks in accuracy significantly. On the other hand, the word embedding extraction-based approach manages to achieve high accuracy results by demanding low resources. Therefore, it provides a balance between computational efficiency and predictive performance, constituting a suitable choice for resource limited environments where achieving high accuracy is still a priority. To summarize the scenarios in which each approach is optimal, we provide Table 10.

In addition, given the low inference time per function of all the approaches, but especially of the word embedding extraction one, we suggest practitioners to use the examined solutions as copilots within their Integrated Development Environments (IDEs). They could integrate these models to identify potential vulnerabilities without disrupting their development workflow. In this way, they could also compare transfer learning-based solutions with existing vulnerability detection tools, making them a valuable alternative to traditional static code analyzers. Moreover, for practitioners who develop their own AI-based models, our findings can provide guidance regarding which implementation choices to use. For instance, they could use as a starting point the word embedding extraction approach to build LLM-based VPMs.

As regards the implications to researchers, this study found that different implementation choices significantly impact the performance and computational efficiency of Transformer-based VPMs. To this end, researchers can extend our work by exploring additional implementation choices. In particular, they can examine complex and combinatorial options such as hybrid Transformer architectures that employ both fine-tuning and embedding extraction, hierarchical embeddings that represent multi-level structures in data (e.g., modules, classes, functions, and statements), and Ensembles of LLMs so that one model can help the others. Furthermore, we recommend that researchers enrich the VP literature with new techniques that are constantly emerging as AI-driven code analysis continues to evolve rapidly. For instance, future research endeavors could focus on examining techniques such as Retrieval-Augmented Generation (RAG), Mixture of Experts (MoE), and Reinforcement Learning from Human Feedback (RLHF) during building VPMs.

Moreover, experimentation with various LLMs of the same or larger scale is proposed as a future research direction. Researchers can repeat this analysis using models such as Mistral, Llama, and GPT-4 to construct VPMs and assess the generalizability of our findings regarding the implementation choices in transfer learning. Furthermore, researchers could explore the effectiveness in VP of additional embedding methods used in NLP (e.g., SBERT [60], ELMo [61], CoVE [62], etc.). In addition, we suggest researchers consider the importance of computational trade-offs in selecting transfer learning strategies for VP. Given the already analyzed advantages and disadvantages of the examined implementation choices regarding the computational footprint, future research could also explore quantization techniques for LLM-based VP models, which could further reduce memory requirements and training times.

Finally, we suggest researchers who build VPMs to explore further the utilization of LLMs for constructing VPMs, focusing on the use of a code-oriented Transformer-based model for contextual representation of source code tokens. We encourage them to examine the word embeddings extraction approach first in their research and to attempt to leverage both the power of pre-trained text-mining models and of models capable of learning graphical representations to further improve the performance of the VPMs.

## 6. Threats to validity

This section discusses potential threats to the construct, internal, and external validity of our study. By critically evaluating methodological challenges and limitations, we aim to enhance the transparency and reliability of our findings.

### *6.1. Construct validity*

The validity of the study's findings could have been affected by the accuracy of the vulnerability-related data used for training and evaluating the developed models. The identification of vulnerability-fixing commits is challenging, as it is possible that some commits may not fully repair the underlying vulnerability or may only address a subset of vulnerabilities within a given software component. Furthermore, there is a small possibility that the data samples that were utilized as non-vulnerable may contain an undetected vulnerability. Thereby, we considered them as neutral.

### *6.2. Internal validity*

Regarding the internal validity, there is a potential limitation by the specific Transformer models chosen for the analysis. The selection of CodeGPT-2 and CodeBERT may introduce a selection bias, as it cannot be excluded that other Transformer models or different pre-training techniques could lead to different results. Additionally, we may not have tested all the possible combinations of hyperparameter values. The correlation between the models' hyperparameters can make it difficult to isolate the effects of individual hyperparameters, which could lead to sub-optimal model performance. To mitigate this risk, the hyperparameter tuning process that we followed was really exhaustive.

### *6.3. External validity*

External validity of this study concerns the use of datasets restricted to C/C++ source code. This narrow focus may restrict the generalizability of the findings to other programming languages. Another external validity threat is the exclusive reliance on open-source code. Open-source projects may differ in terms of development practices, coding styles, and vulnerability patterns compared to proprietary software, potentially reducing the applicability of the findings to commercial software development environments. To mitigate this threat, it would be useful to incorporate data from both open-source and proprietary software projects in the future.

## 7. Conclusions and future work

In this work, our purpose was to examine the different implementation choices when using transfer learning in the task of vulnerability prediction, and therefore, to highlight the optimal approach. We compared the fine-tuning and sentence-level embedding extraction approaches, and we investigated the possible benefits of extracting word embeddings from pre-trained LLMs so as to feed and train separate DL models in vulnerability prediction. We also examined whether it is better to have LLMs pre-trained on both source code and NL or only on source code for a code analysis task such as vulnerability prediction. Moreover, we compared our best models with state-of-the-art vulnerability prediction models.

The analysis demonstrated that, for the downstream task of vulnerability prediction, there is no benefit of proceeding with the time-consuming approach of fine-tuning instead of the LLM word embeddings extraction, and therefore, it suggests the latter, which achieves the same accuracy but with by far smaller training cost. Furthermore, regarding the question about the possible advantage of pre-training LLMs on bimodal data (i.e., both on source code and natural language), the study concludes that it is not necessarily beneficial but neither damaging for vulnerability prediction. Finally, this study highlighted the importance of context aware embeddings for representing the source code, which can lead to vulnerability predictors of much higher accuracy than static embedding vectors and indicated that a combination of textual and graphical source code representations through a multi-modal model could provide even better vulnerability predictors.

Future work includes the interpretability of LLMs in vulnerability prediction as well as the examination of their capabilities in the cross-project evaluation scenario. Regarding the former, we aim at applying explainable AI techniques to identify the reasoning behind the vulnerability predictions of the LLMs. For the latter, we are interested in comparing the performance of LLMs to traditional techniques for predicting vulnerabilities in software projects that are completely different from the projects that constitute the training dataset. We plan also to explore additional LLMs and implementation choices, focusing on hybrid and complex architectures, to enhance predictive performance in vulnerability prediction.

### Declaration of competing interest

The authors declare no conflict of interest.


### Acknowledgments

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


### Appendix A.

Supplementary material related to this article (e.g., code and data) can be found online [63]: `https://sites.google.com/view/vulgpt`

### CRediT authors' contribution statement

**Ilias Kalouptsoglou:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Visualization. **Miltiadis Siavvas:** Methodology, Validation, Formal analysis, Investigation, Writing - Review & Editing, Supervision. **Apostolos Ampatzoglou:** Methodology, Writing - Review & Editing, Supervision, Project administration. **Dionysios Kehagias:** Writing - Review & Editing, Resources, Supervision, Project administration, Funding acquisition. **Alexander Chatzigeorgiou:** Methodology, Writing - Review & Editing, Supervision, Project administration.

### References


[1] ISO/IEC 27000:2018, `https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en` (Accessed: 2023-11-21).

[2] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, A. Chatzigeorgiou, Software vulnerability prediction: A systematic mapping study, Information and Software Technology 164 (2023) 107303. doi:https://doi.org/10.1016/j.infsof.2023.107303.
URL `https://www.sciencedirect.com/science/article/pii/S095058492300157X`

[3] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, A. Ampatzoglou, Examining the capacity of text mining and software metrics in vulnerability prediction, Entropy 24 (5) (2022). doi:10.3390/e24050651.
URL `https://www.mdpi.com/1099-4300/24/5/651`

[4] J. Walden, J. Stuckman, 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.

[5] R. Scandariato, J. Walden, A. Hovsepyan, W. Joosen, Predicting vulnerable software components via text mining, IEEE Transactions on Software Engineering 40 (10) (2014) 993–1006.

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

[7] Y. Pang, X. Xue, 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.

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, arXiv preprint arXiv:1801.01681 (2018).

[9] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, A. Ampatzoglou, An empirical evaluation of the usefulness of word embedding techniques in deep learning-based vulnerability prediction, Security in Computer and Information Sciences (2021) 23.

[10] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781 (2013).
URL `https://doi.org/10.48550/arXiv.1301.3781`

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

[12] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we there yet?, IEEE Transactions on Software Engineering 48 (9) (2022) 3280–3296. doi:10.1109/TSE.2021.3087402.

[13] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov, Bag of tricks for efficient text classification, arXiv preprint arXiv:1607.01759 (2016).
URL `https://doi.org/10.48550/arXiv.1607.01759`

[14] N. Ziems, 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.

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

[16] B. Steenhoek, M. M. Rahman, R. Jiles, W. Le, An empirical study of deep learning models for vulnerability detection, arXiv preprint arXiv:2212.08109 (2022).

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

[18] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).
URL `https://doi.org/10.48550/arXiv.1810.04805`

[19] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al., Improving language understanding by generative pre-training, arXiv preprint arXiv:2106.01367 (2018).

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

[21] M. Fu, 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. doi:10.1145/3524842.3528452.

[22] 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).

[23] 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).
URL `https://doi.org/10.48550/arXiv.2002.08155`

[24] M. Siavvas, I. Kalouptsoglou, E. Gelenbe, D. Kehagias, D. Tzovaras, Transforming the field of vulnerability prediction: Are large language models the key?, in: 2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2024, pp. 1–6.

[25] Q. Liu, M. J. Kusner, P. Blunsom, A survey on contextual embeddings, arXiv preprint arXiv:2003.07278 (2020).

[26] Y. Shin, 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.

[27] Y. Shin, L. Williams, 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, pp. 315–317. doi:10.1145/1414004.1414065.
URL `https://doi.org/10.1145/1414004.1414065`

[28] I. Chowdhury, M. Zulkernine, Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities, Journal of Systems Architecture 57 (3) (2011) 294–313.

[29] I. Kalouptsoglou, M. Siavvas, D. Tsoukalas, 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.

[30] A. Hovsepyan, R. Scandariato, W. Joosen, 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.

[31] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A framework for using deep learning to detect software vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19 (4) (2021) 2244–2258.

[32] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

[33] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv preprint arXiv:1409.0473 (2014).

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

[35] V. Sanh, L. Debut, J. Chaumond, T. Wolf, Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, arXiv preprint arXiv:1910.01108 (2019).

[36] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut, Albert: A lite bert for self-supervised learning of language representations, arXiv preprint arXiv:1909.11942 (2019).

[37] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, L. Zettlemoyer, Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, arXiv preprint arXiv:1910.13461 (2019).

[38] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, The Journal of Machine Learning Research 21 (1) (2020) 5485–5551.

[39] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, et al., Lamda: Language models for dialog applications, arXiv preprint arXiv:2201.08239 (2022).

[40] J. Pennington, R. Socher, 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.

[41] X. Yuan, G. Lin, Y. Tai, J. Zhang, Deep neural embedding for software vulnerability discovery: Comparison and optimization, Secur. Commun. Networks 2022 (2022) 5203217:1–5203217:12.

[42] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, H. Kim, Vuldebert: A vulnerability detection system using bert, in: 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2022, pp. 69–74. doi:10.1109/ISSREW55968.2022.00042.

[43] H. Hanif, 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.

[44] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software vulnerability detection using large language models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), 2023, pp. 112–119. doi:10.1109/ISSREW60843.2023.00058.

[45] H. Booth, D. Rike, G. A. Witte, The national vulnerability database (nvd): Overview (2013).

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

[47] C. Seas, G. Fitzpatrick, J. A. Hamilton, M. C. Carlisle, Automated vulnerability detection in source code using deep representation learning, in: 2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, 2024, pp. 0484–0490.

[48] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, Z. Su, D2a: A dataset built for ai-based vulnerability detection methods using differential analysis, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2021, pp. 111–120.

[49] G. Bhandari, A. Naseer, L. Moonen, Cvefixes: automated collection of vulnerabilities and their fixes from open-source software, in: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30–39.

[50] J. Fan, Y. Li, S. Wang, T. N. Nguyen, Ac/c++ code vulnerability dataset with code changes and cve

summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.
[51] R. Croft, M. A. Babar, M. M. Kholoosi, Data quality for software vulnerability datasets, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 121–133.
[52] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, A. Chatzigeorgiou, Vulnerability classification on source code using text mining and deep learning techniques, in: 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C), IEEE, 2024, pp. 47–56.
[53] I. Loshchilov, F. Hutter, Decoupled weight decay regularization, arXiv:1711.05101 (2017).
[54] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, Advances in neural information processing systems 24 (2011).
[55] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in statistics: Methodology and distribution, Springer, 1992, pp. 196–202.
[56] M. Siavvas, E. Gelenbe, D. Kehagias, D. Tzovaras, Static analysis-based approaches for secure software development, in: Security in Computer and Information Sciences, Springer International Publishing, 2018, pp. 142–157.
[57] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 672–681.
[58] Y. Li, S. Wang, T. N. Nguyen, Vulnerability detection with fine-grained interpretations, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 292–303.
[59] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th international conference on mining software repositories, 2022, pp. 596–607.
[60] N. Reimers, Sentence-bert: Sentence embeddings using siamese bert-networks, arXiv preprint arXiv:1908.10084 (2019).
[61] J. Sarzynska-Wawer, A. Wawer, A. Pawlak, J. Szymanowska, I. Stefaniak, M. Jarkiewicz, L. Okruszek, Detecting formal thought disorder by deep contextualized word representations, Psychiatry Research 304 (2021) 114135.
[62] B. McCann, J. Bradbury, C. Xiong, R. Socher, Learned in translation: Contextualized word vectors, Advances in neural information processing systems 30 (2017).
[63] Transfer Learning for Software Vulnerability Prediction using Transformer Models, `https://sites.google.com/view/vulgpt` (Accessed: 2024-07-03).