# VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection

Hazim Hanif
*Department of Computing*
*Imperial College London, UK;*
*Faculty of Computer Science and Information Technology,*
*University of Malaya, Malaysia*
m.md-hanif19@imperial.ac.uk

Sergio Maffeis
*Department of Computing*
*Imperial College London, UK*
sergio.maffeis@imperial.ac.uk

2022 International Joint Conference on Neural Networks (IJCNN) | 978-1-7281-8671-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IJCNN55064.2022.9892280

***Abstract*—This paper presents VulBERTa, a deep learning approach to detect security vulnerabilities in source code. Our approach pre-trains a RoBERTa model with a custom tokenisation pipeline on real-world code from open-source C/C++ projects. The model learns a deep knowledge representation of the code syntax and semantics, which we leverage to train vulnerability detection classifiers. We evaluate our approach on binary and multi-class vulnerability detection tasks across several datasets (Vuldeepecker, Draper, REVEAL and muVuldeepecker) and benchmarks (CodeXGLUE and D2A). The evaluation results show that VulBERTa achieves state-of-the-art performance and outperforms existing approaches across different datasets, despite its conceptual simplicity, and limited cost in terms of size of training data and number of model parameters.**

***Index Terms*—Vulnerability detection, Software vulnerabilites, Pre-training, Deep learning, Representation learning.**

## I. Introduction

MITRE [1] reports an increase in the number of software CVEs submitted yearly since 2016, reflecting the increased threat to the overall security of the software ecosystem. Accordingly, there has been a steady growth of research in software vulnerability detection over the years [2], across the spectrum of vulnerability detection approaches such as static analysis, dynamic analysis and machine learning-based detection models.

Deep learning has obtained encouraging results for software vulnerability, in particular using sequence- and graph-based techniques such as Bidirectional-LSTM [3]–[5] and Graph Neural Networks [6], [7]. These techniques attempt to embed syntactic and semantic information from the code explicitly, for example by using various dependency and data flow analyses to preprocess source code and extract various artefact such as code gadgets, control flow graphs and dependency graphs which are eventually fed to the respective neural network.

In this paper, we follow a different approach inspired by the recent successes of Transformer-based neural architectures [8]–[12] able to learn deep representation knowledge of natural language textual data. Our aim is to build a representation model of C/C++ that embeds syntactic and semantic information about the language without our direct intervention, and then use that model as a basis to build vulnerability detection models using standard neural architectures.

We present VulBERTa, which aims to learn a deep representation of C/C++ source code from large codebases that consist of different software projects. To facilitate learning the internal code representation, we need to create a language-aware reliable tokenisation pipeline, to parse and tokenise source code while ensuring that basic, yet key syntactic and semantic information is made available to the model. In our tokenisation pipeline, we implemented several novel features to enhance the tokenisation ability of the existing Byte-pair Encoding (BPE) [13] tokeniser. We introduce a custom tokeniser that combines BPE with custom pre-defined code tokens to build the vocabulary of our tokeniser. These pre-defined tokens are based on the AST node type of Clang [14] (standard keywords and punctuation) and lists of common C/C++ API function names. Our custom tokenisation pipeline creates a better code representation while maintaining its original syntactic structure even after being encoded.

VulBERTa implements a Transformer-based deep learning architecture called RoBERTa [15]. VulBERTa builds its code representation knowledge via Masked Language Modelling (MLM) [9]. We pre-train VulBERTa on fewer samples (2.28 million) and model parameters (125 million) than existing approaches [16], [17]. Then, we connect the pre-trained VulBERTa model with alternatively a MultiLayer Perceptron (VulBERTa-MLP) and a Convolutional Neural Network (VulBERTa-CNN) in order to fine-tune vulnerability detection models.

We evaluate VulBERTa across different datasets to test the performance and transferability of our pre-trained model. The evaluation results show that we achieve state-of-the-art detection performance across these datasets. In particular, we achieved an F1 score of 57.92% on the Draper [18] dataset, outperforming existing approaches, which is encouraging as this dataset is considered challenging due to being imbalanced and having a mix of real-world and synthetic samples. VulBERTa also performs well on multi-class classification task on the easier muVuldeepecker [5] dataset, with a weighted F1 score of 99.59%. We also tested VulBERTa on two software vulnerability detection benchmarks (CodeXGLUE and D2A), where it outperforms most existing approaches, despite using less training data and fewer model parameters. These results


978-1-7281-8671-9/22/$31.00 ©2022 IEEE

show that VulBERTa is effective in learning a deep representation of source code, and using that knowledge to detect software vulnerabilities.

In summary, our main contributions are:

- A custom tokenisation pipeline which combines the BPE algorithm with novel pre-defined code tokens (standard C/C++ keywords, punctuation and library API calls), providing better code encodings while maintaining the syntactical structure of source code.
- A small and simplified pre-trained model, (VulBERTa) that provides transferability of pre-trained embedding weights, which is reusable in less complex architectures such as MLP and CNN.
- Software vulnerability detection models, VulBERTa-MLP and VulBERTa-CNN that achieve state-of-the-art (SOTA) detection performance across different datasets and top-3 positions in two benchmarks (CodeXGLUE and D2A).

Our code and data are open source, and made available at *https://github.com/ICL-ml4csec/VulBERTa*.

## II. Related Work

This Section discusses related work on software vulnerability detection using deep learning, and more in general on pre-training models for representing programming languages.

### A. Vulnerability detection using deep learning

Software vulnerability detection remains a challenging problem for security researchers across academia and industry. Much work has been done to reliably and efficiently detect security vulnerabilities in source code [19]–[21]. However, recent advances in deep learning and its application across different domains have induced security researchers to test the effectiveness of deep learning on vulnerability detection tasks.

One of the earliest works that use deep learning techniques to detect software vulnerabilities on raw source code is [18]. The unusual aspect of this work is that instead of using CNN and RNN to train a classifier model, they used it as a feature extractor. The extracted features are then passed to a Random Forest (RF) classifier trained to detect software vulnerabilities. This approach achieves an AUC score of 90.4% when tested on real-world dataset.

Vuldeepecker [3] proposes a structure called code gadgets that is based on the extraction of library/API function calls from the source code. However, a limitation of this work is that it only considers vulnerability that involves library/API function calls. To overcome such limitations, the authors extended their work and proposed SySeVR [4] as a systematic framework for detecting vulnerabilities and can learn from syntax and semantic of the source code. They updated the code gadgets approach to accommodate both data and control dependency information from the code. Following the success of code gadgets in Vuldeepecker and SySeVR, muVuldeepecker [5] was introduced to detect different types of vulnerabilities through multi-class classification task. In addition, this work proposed a mechanism to pinpoint the location of a specific vulnerability in the source code.

Independently from the work on code gadgets, other research investigated graph-based approaches for vulnerability detection. Devign [6] implemented a Graph Neural Network (GNN) model to learn data and control dependency code graphs and proposed a novel Conv module that extracts interesting features from source code. DeepWukong [22] embeds code fragments in a compact and low-dimensional representation to detect ten different types of vulnerabilities using GNN. Beyond C/C++, DeepTective [23] detects SQLi, XSS and command injection vulnerabilites in PHP source code by combining Gated Recurrent Units (GRU) and Graph Convolutional Networks. Hybrid neural networks were also explored in [24], where the authors constructed a Hybrid Deep Learning Network to detect code injection attacks in HTML5-based applications.

Another interesting line of research for deep learning-based vulnerability detection focuses on dataset quality. REVEAL [25] leverages the SMOTE re-sampling and duplicate removal method to address the problem of imbalanced datasets. D2A [26] proposes a curated benchmark dataset based on a differential analysis approach, by analysing version pairs of source code from multiple open-source projects.

### B. Pre-trained models of source code

Pre-training is a technique used by the Transformer [8] neural architecture, initially introduced in the Natural Language Processing (NLP) domain to learn deep representation knowledge of textual data, for language-specific tasks such as text translation, text completion and text generation [9]–[12].

To investigate and test the hypothesis of whether programming languages can be understood using NLP techniques, [17] presented C-BERT, a pre-training architecture that is based on Abstract Syntax Trees (AST). The main idea behind this is to learn AST-based features automatically from source code during pre-training. In the fine-tuning task, C-BERT outperformed existing approaches across AST node tagging and vulnerability detection tasks.

CodeBERT [27] implemented BERT [9], one of the earliest pre-training architectures, on programming languages. The authors pre-trained the model using bimodal and unimodal data, consisting of code and natural language pairs from different programming languages such as Java and Python. CodeBERT achieved high BiLingual Evaluation Understudy (BLEU) score as compared to RoBERTa [15] models.

DOBF [28] proposes a new pre-training objective that is based on code deobfuscation. The deobfuscation objective focuses on the structural aspect of programming language, whether pre-training helps the model to learn syntactic and structural information of the source code. Besides that, in order to learn and investigate large graph-based representation, GraphCodeBERT [29] proposed a pre-trained model that incorporates graph structure into the Transformer-based model using graph-guided masked attention to filter irrelevant signals. In the evaluation, GraphCodeBERT showed a strong performance compared to other pre-trained models (RoBERTa and

CodeBERT) in different tasks such as code clone detection, code translation, and code refinement.

## III. VulBERTa

In this section, we introduce VulBERTa, our pre-training architecture for detecting vulnerabilities in C/C++ source code at function-level granularity. The architecture is divided into three key components: a tokenisation technique that parses and tokenises code using a custom vocabulary; a pre-training session that builds a representation model for code; and a fine-tuning session that refines the model to target a concrete classification task. Figure 1 visualises the 8 main steps of the VulBERTa training pipeline.

### A. Tokeniser

Our tokenisation pipeline aims at preserving syntactic structure and selected semantic identifiers. It consists of a parser, a tokeniser and an encoder. These components stack onto each other to transform raw source code into a structure understandable by a neural network. Figure 2 visualises the overall tokenisation pipeline of VulBERTa from raw code to encoded output sequence. Below, we describe each step of the pipeline.

*1) Parser:* We remove comments from the source code of each function using several regular expressions. Then, we parse the source code using Clang [14], a robust C/C++ parser that can parse code without including any libraries or external dependencies. Clang allows us to preserve the syntactic structure of the source code while breaking it down into a sequence of code tokens.

*2) Tokenisation:* The tokens produced by the Clang parser are further processed by the BPE algorithm, modified to take into account our pre-defined tokens, to further break the parsed input down into fine-grained tokens for encoding.

**Byte Pair Encoding (BPE)** is a subword tokenisation algorithm proposed by [13] that replaces a similar pair of consecutive bytes with a byte that does not appear in the data. Subword tokenisation also reduces the possibility of encountering out-of-vocabulary tokens as most subword tokens are available in the vocabulary. Following several implementations of BPE, such as in [17], [27], we define a vocabulary size of 50000 as our maximum number of entries in the vocabulary.

**Pre-defined tokens** are tokens we explicitly include in the vocabulary, thus excluding them from the subword tokenisation process. Our goal is to preserve their syntactic or semantic meaning. We have considered using token bucketing, normalization and standard C/C++ tokens. We found that by pre-defining C/C++ keywords, punctuation, and standard API names, we preserve more information about the meaning of the source code during pre-training. Table I summarises our pre-defined tokens, which are excluded from BPE. The full list consists of 451 pre-defined tokens. The other tokens consist of literals and identifiers, and are passed to BPE to undergo the tokenisation process.

TABLE I
CUSTOM PRE-DEFINED TOKENS.

| Token type | Total | Examples |
|---|---|---|
| BPE reserved tokens | 5 | `<pad>`, `<unk>`, `<mask>` |
| Standard C/C++ keyword tokens | 104 | `int`, `if`, `void` |
| Standard C/C++ punctuation tokens | 54 | `=`, `++`, `->` |
| Standard C/C++ API call tokens | 288 | `strlen`, `scanf`, `memcpy` |

*3) Encoder:* Encoding is the process of converting the code tokens into tensors. For pre-training, we have set the maximum sequence length to 512 to maximise and generalise the learning throughout the data, as the pre-training dataset comprises different codebases. Meanwhile, we increase the maximum sequence length to 1024 for the fine-tuning task so that it can contain, without truncation, more than 90% of the actual samples on average. We pad shorter sequences to the right, with a special padding token (`<pad>`).

### B. Pre-training

Pre-training is the initial training session where we train a standard RoBERTa [15] model with MLM objective in order to learn an informative general representation of C/C++ code across different software project. We will refer to the model obtained from this pre-training session as the **VulBERTa model**. Combining different software projects is beneficial to the learning process, as it generalises the representation knowledge of the code even further across different coding styles. This also helps to increase the robustness of the model during pre-training. We set the embedding size to be 768 dimensions, following RoBERTa-base. This is the core embedded knowledge of the model, that will be useful for downstream fine-tuning tasks.

### C. Fine-tuning

In fine-tuning, we further train the pre-trained model on a specific downstream task, which in our case is software vulnerability detection. Figure 3 shows our fine-tuning pipeline for vulnerability detection. We implement two different classification approaches for fine-tuning. The first approach is a standard multilayer perceptron (MLP) on top of the pre-trained model, and the second approach uses a Text Convolutional Neural Network (TextCNN), which is cheaper and faster to fine tune due to the robustness of CNN architecture.

**VulBERTa-MLP:** We implement a fully-connected layer with 768 neurons and one output layer 2 or 41 neurons based on whether our fine-tuning dataset is a binary or multi-class classification dataset. During fine-tuning, we reuse the pre-trained weights from VulBERTa and continue the training for several epochs. This approach is the most common approach for fine-tuning pre-trained models as it takes advantage of the whole VulBERTa architecture with little modification [9], [15].

**VulBERTa-CNN:** We extract the embedding weights of the pre-trained VulBERTa model and use them as the embedding weights for a Hybrid TextCNN [30]. The TextCNN architecture consists of three 1-dimensional CNN, each with its max-pooling layer. The outputs are concatenated and flattened,

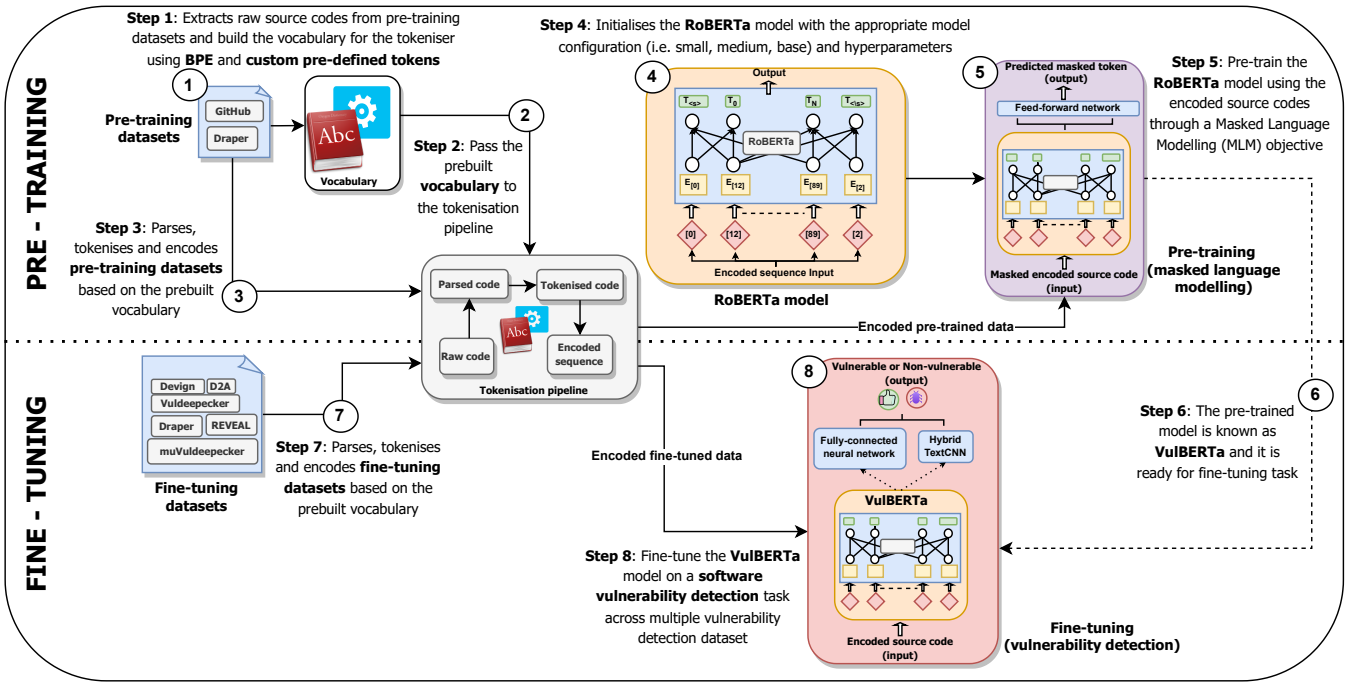


Fig. 1. VulBERTa training pipeline. Steps are taken in order from 1 to 8.

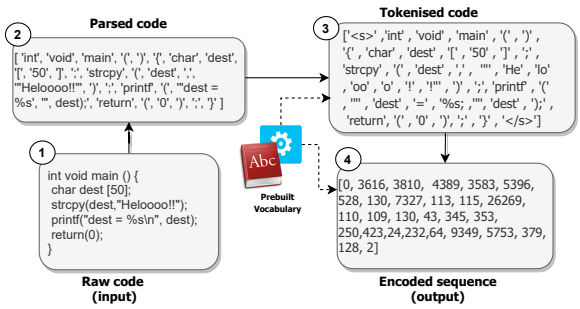


Fig. 2. Tokenisation pipeline.

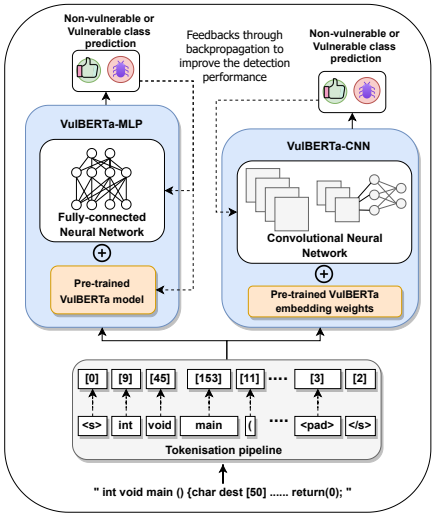


Fig. 3. Fine-tuning (vulnerability detection) pipeline.

and then fed to two fully-connected layers (256 and 128 neurons) with one output layer for classification. Since the embeddings are already pre-trained on a large language model, we freeze them during the training task. This technique allows the TextCNN model to inherit and use the representation knowledge of the embeddings and focus on tuning the weights of the task-specific CNN layers.

## IV. Datasets

This Section describes the datasets used in the rest of the paper. They consist of function-level C/C++ source code from various codebases, including open-source repositories and synthetic code samples. We divide these datasets in two categories based on their prevalent use for either pre-training or fine-tuning. All datasets mentioned below are in the public domain, and available for download.

### A. Pre-training datasets

For pre-training, we use the Masked Language Modelling (MLM) task on the GitHub and Draper datasets.

*1) GitHub:* The GitHub dataset consists of the source code of 1,101,075 C/C++ functions extracted from 1060 open-source repositories on GitHub. We compiled this dataset using the public GitHub API and the `PyGithub` [31] package. First we gathered all the names of repositories containing C/C++ source code, together with their stars count. Then we sorted these repositories based on their star counts, and started fetching files from the repositories. This process took around

two days to complete due to API rate-limit restrictions. Finally we used Joern [19], an open-source code querying engine for C/C++, to efficiently extract individual functions from each downloaded file. This step is essential as our approach aims to detect function-level security vulnerabilities.

*2) Draper:* The Draper dataset is a software vulnerability detection dataset initially introduced by [18]. This dataset consists of 1,274,366 C/C++ functions gathered from various locations such as Debian Linux distributions, open-source GitHub repositories and the Juliet test suite [32]. This dataset ranges from highly documented production code to synthetic test samples.

### B. Fine-tuning datasets

For the fine-tuning task, which in our case is vulnerability detection, we have selected a number of datasets which security researchers have compiled in order to evaluate their respective vulnerability detection approaches.

*1) Vuldeepecker:* The Vuldeepecker dataset is a vulnerability detection dataset introduced in [3]. It consists of real-world samples from the National Vulnerability Database (NVD) [33] and synthetic samples from the Software Assurance Reference Dataset (SARD) [34] project. These two projects are actively maintained by the National Institute of Standards and Technology (NIST). This dataset is frequently used as a benchmark dataset to evaluate vulnerability detection techniques for C/C++ source code.

*2) Draper:* This is the same dataset we described in Section IV-A2, however for fine-tuning we include the (binary) labels. This dataset was checked by three static analysers and labelled by a team of security experts.

*3) REVEAL:* The REVEAL dataset is a real-world software vulnerability detection dataset introduced in [25] in response to existing datasets that contain lots of data duplication and unrealistic distribution of vulnerable classes. This dataset is a binary detection dataset consisting of source code from two open-source projects: Linux Debian kernel and Chromium.

*4) muVuldeepecker (MVD):* The muVuldeepecker dataset is a multiclass vulnerability detection dataset introduced in [5]. It is remarkably similar to the Vuldeepecker dataset as this dataset also comes from the NVD and SARD. However, the main difference is that this dataset consists of code gadgets instead of the usual function-level source code.

*5) Devign:* The Devign dataset is a real-world software vulnerability detection dataset initially introduced in [6]. This dataset consists of function-level C/C++ source code from two popular open-source software projects, QEMU and FFmpeg. The labelling and verification have been done manually by a team of security researchers over a two-round process.

*6) D2A:* The D2A dataset is a real-world vulnerability detection dataset curated and introduced by the IBM Research team [26]. This dataset consists of several open-source software projects like FFmpeg, httpd, Libav, LibTIFF, Nginx and OpenSSL. It was created using a differential analysis technique to label issues reported by static analysers.

## V. Software vulnerability detection

In this Section we describe how we pre-train the VulBERTa model, build fine-tuned VulBERTa-MLP and VulBERTa-CNN models, and evaluate them across several vulnerability detection datasets and benchmarks.

### A. Experimental setup

*1) Hardware and software:* We use PyTorch 1.7 [35] with CUDA 10.2 on top of Python 3.7 for all fine-tuning experiments. For pre-training we use Google Compute Engine (GCP) VMs that have 48 vCPUs, 240GB RAM and 2 NVIDIA Tesla A100 40GB GPUs. For fine-tuning, we use a machine with 48 cores Intel Xeon Silver CPU, 292GB RAM and 2 NVIDIA GTX TITAN Xp GPU. Each GPU has 12GB of video memory to accommodate different model configurations.

*2) Performance criteria:* For every experiment, we report several evaluation metrics, including those used in the initial work for each dataset. This way, we can perform a fairer comparison. The metrics are true negatives (TN), false negatives (FN), true positives (TP), false positives (FP), accuracy, precision, recall, F1-score, Receiver Operating Characteristic area under the curve (ROC-AUC), precision-recall AUC (PR-AUC), and Matthews Correlation Coefficient (MCC).

*3) Baselines methods:* In the performance evaluation, we compare VulBERTa with two baselines techniques on top of existing approaches for each dataset. These two baselines are widely used to analyse sequence-based input for vulnerability detection and have been used for example in [6] and [17].

**(i) Baseline-BiLSTM:** This technique is a variation of LSTM, which implements a two layers Bi-directional LSTM [36] and several fully-connected layers to learn from sequences of source code for vulnerability detection. The bi-directional LSTM learns forward and backward relationships of code sequence simultaneously.

**(ii) Baseline-TextCNN:** This is a variant of CNN [30], where the input data is natural language text instead of images. In this case, we use source code as the input data and feed it into CNN. This technique deploys three convolutional layers with pooling, and concatenates them into a single layer before passing the results to several fully-connected layers.

*4) Model pre-training:* We pre-train the VulBERTa model with Draper and GitHub datasets using MLM. We experiment with different RoBERTa configurations (i.e. *small*, *medium* and *base*) to see how the number of model parameters affects the pre-training performance of the model. The duration of each pre-training session falls between 72 to 96 hours depending on the model configuration. The training session is done up to 500,000 steps and a learning rate scheduler is used to reduce the learning rate over time as the training loss plateaus. Based on the results, we found that the Base model gives the lowest loss during training after 500,000 steps. Therefore, we choose the VulBERTa model with Base configuration ($\sim$125M parameters) as our reference pre-trained model for fine-tuning.

*5) Model fine-tuning:* We fine-tune our pre-trained VulBERTa-MLP and VulBERTa-CNN models separately, on each dataset described in Section IV-B, using vulnerability

detection as the fine-tuning objective. We set the maximum number of epochs to 10, which is more than sufficient as the models tend to start overfitting to the training set after 4-5 epochs. We set the learning rate to 0.00003 with a learning rate scheduler to reduce the learning rate as the training loss plateaus. We follow the original split for each dataset, but if the split information is unavailable, we split the dataset 80/10/10 (training/validation/testing). Each fine-tuning session lasted between 5 and 10 hours, depending on the size of the dataset and model.

### B. Evaluation on selected datasets

The vulnerability detection experiments are separated by dataset. For each dataset, we selected the preferred evaluation metric (PEM) used for comparison in the original paper. Table II shows the evaluation results, where we highlighted the highest PEM score for each dataset.

*1) Vuldeepecker:* The precision score reported by Vuldeepecker is 91.9%. However, using VulBERTa-MLP, we achieved a precision score of 95.76%, which beats the original score by 3.86%. On top of that, our model obtained a higher F1 score, 93.03%, compared to the score obtained by Vuldeepecker, 92.9%, which shows that our model is balanced when classifying vulnerable and non-vulnerable samples. The low rate of false positives (0.39%) and negatives (9.14%) indicates that VulBERTa-MLP detects vulnerabilities both in synthetic and real-world code with a low misclassification rate.

*2) Draper:* The authors of [18] use several evaluation metrics to evaluate their model on the Draper dataset. We choose MCC as a basis for comparison because it is suitable for imbalanced datasets, and the the vulnerable and non-vulnerable classes of the Draper dataset are imbalanced. VulBERTa-CNN obtained an MCC score of 55.86% on this dataset. That is a 2.26% increase on the performance reported in [18], and is significant as we can provide better detection while maintaining the class balance across prediction.

*3) REVEAL:* The VulBERTa-MLP model achieved a higher F1 score of 45.27% than the one of 41.25% reported in [25], despite not using the data rebalancing techniques proposed there. Instead, we assigned weights to each class (vulnerable and non-vulnerable) during fine-tuning to reduce the class imbalance problem without altering the dataset. Our approach also obtained a true positive rate (TPR) higher by 2.57%. This indicates that VulBERTa-MLP is more effective in detecting vulnerable samples correctly while also maintaining the balance with the non-vulnerable class.

*4) muVuldeepecker:* Differently from the previous experiments, this is a multi-class classification task. The muVuldeepecker dataset contains separate classes for 40 CWEs, so each positive sample can be mapped to a specific security vulnerability. VulBERTa-MLP achieved a very high weighted F1 score of 99.59% compared to the 96.28% reported in [5]. It also reduced the false negatives rate (FNR) from 5.53% to 0.41%, which is significant as the additional samples detected as vulnerable need to be assigned to the correct class. Furthermore, looking into the prediction for specific classes such as CWE-190 and CWE-191 (integer overflow and underflow), we can see that VulBERTa-MLP is able to correctly assign more than 90% of the vulnerable samples in their respective classes.

Our models manage to surpass the performance of the compared models despite using the latter's datasets and preferred evaluation metrics. This happens across a variety of datasets with synthetic and real world data, balanced and imbalanced classes, binary and multi-class classification tasks.

### C. Evaluation on benchmarks

We also evaluated VulBERTa-MLP and VulBERTa-CNN on two publicly available benchmarks which are used by the community as a basis for comparing different source code models on standardised tasks. In both cases, the PEM for the vulnerability detection task is Accuracy.

*1) CodeXGLUE:* The CodeXGLUE [37] benchmark is the first and most popular benchmark for programming language understanding, introduced by Microsoft Research. We focus on the *defect detection* task, which consists of vulnerability detection on the Devign dataset, described in Section IV-B5. Table III-A shows the standings of the CodeXGLUE benchmark leaderboard at the time of publication, including our results[1]. VulBERTa-MLP achieved an accuracy of 64.75%, ranking 3rd below CoText and C-BERT. Note that VulBERTa-MLP has a significantly lower number of model parameters (55.07%) than CoTexT, and is trained on a fraction of the data of the top 2 models. Impressively, VulBERTa-CNN achieves a slightly lower performance yet with a model size less than 1% of CoTexT and 2% of C-BERT.

*2) D2A:* D2A [26] is a benchmark for vulnerability detection introduced by IBM Research. It is based on the D2A dataset, described in Section IV-B6. Table III-B shows the standings of the D2A benchmark leaderboard at the time of publication, including our models[2]. VulBERTa-MLP tops the leaderboard for the "function" task (the only one relevant to our approach) with an accuracy of 62.30%, and VulBERTa-CNN is second with 60.68%. Interestingly, on this dataset both our models outperform C-BERT, despite its larger pre-training dataset, and the larger number of parameters (in comparison to VulBERTa-CNN).

### D. Discussion

Table II shows that VulBERTa-MLP has the best PEM score on 3 out of 4 datasets and VulBERTa-CNN has the best PEM score on the remaining dataset. However, if we compare our two models on the full spectrum of fine-tuning results, we notice that the difference between the two is negligible. Similarly, we can see a close relationship between VulBERTa-MLP and VulBERTa-CNN on the benchmark evaluation. Both models consistently ranked next to each other and achieved state of the art performance across both benchmarks. A possible explanation is that the code representation knowledge

[1]Leaderboard accessible at *https://microsoft.github.io/CodeXGLUE*.

[2]Leaderboard accessible at *https://ibm.github.io/D2A*.

TABLE II
DATASET EVALUATION RESULTS: SUMMARY OF EVALUATION METRICS

| Dataset | PEM | Model | FN | FP | TN | TP | Acc (%) | Prec (%) | F1 (%) | Recall (%) | PR-AUC (%) | ROC-AUC (%) | MCC (%) | Weighted F1 (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Vuldeepecker | Precision | Vuldeepecker [3] | - | - | - | - | - | 91.90 | 92.90 | - | - | - | - | - |
| | | VulBERTa-MLP | 99 | 39 | 15002 | 881 | - | **95.76** | 93.03 | - | - | - | - | - |
| | | VulBERTa-CNN | 89 | 44 | 14997 | 885 | - | 95.26 | 90.86 | - | - | - | - | - |
| | | Baseline-BiLSTM | 76 | 810 | 14231 | 898 | - | 52.58 | 66.97 | - | - | - | - | - |
| | | Baseline-TextCNN | 58 | 527 | 14514 | 916 | - | 63.48 | 75.80 | - | - | - | - | - |
| Draper | MCC | Russell et al. [18] | - | - | - | - | - | - | 56.60 | - | 51.80 | 90.40 | 53.60 | - |
| | | VulBERTa-MLP | 4693 | 4598 | 114568 | 3555 | - | - | 43.34 | - | 36.24 | 87.52 | 39.44 | - |
| | | VulBERTa-CNN | 2168 | 6675 | 112491 | 6085 | - | - | 57.92 | - | 56.72 | 92.11 | **55.86** | - |
| | | Baseline-BiLSTM | 1497 | 13836 | 105330 | 6756 | - | - | 46.84 | - | 46.75 | 91.35 | 46.97 | - |
| | | Baseline-TextCNN | 1522 | 12266 | 106900 | 6731 | - | - | 49.40 | - | 49.02 | 91.83 | 49.24 | - |
| REVEAL | F1 | REVEAL [25] | - | - | - | - | 84.37 | 30.91 | 41.25 | 60.91 | - | - | - | - |
| | | VulBERTa-MLP | 84 | 269 | 1775 | 146 | 84.48 | 35.18 | **45.27** | 63.48 | - | - | - | - |
| | | VulBERTa-CNN | 59 | 402 | 1642 | 171 | 79.73 | 29.84 | 42.59 | 74.35 | - | - | - | - |
| | | Baseline-BiLSTM | 63 | 457 | 1587 | 167 | 77.13 | 26.76 | 39.11 | 72.61 | - | - | - | - |
| | | Baseline-TextCNN | 48 | 561 | 1483 | 182 | 73.22 | 24.50 | 37.41 | 79.13 | - | - | - | - |
| muVuldeepecker | Weighted F1 | muVuldeepecker [5] | - | - | - | - | - | - | - | - | - | - | - | 96.28 |
| | | VulBERTa-MLP | 94 | 48 | 8604 | 27583 | - | - | - | - | - | - | - | **99.59** |
| | | VulBERTa-CNN | 94 | 65 | 8587 | 27583 | - | - | - | - | - | - | - | 99.56 |
| | | Baseline-BiLSTM | 8 | 8642 | 10 | 27668 | - | - | - | - | - | - | - | 65.93 |
| | | Baseline-TextCNN | 17 | 8639 | 13 | 27660 | - | - | - | - | - | - | - | 65.95 |

TABLE III
BENCHMARK EVALUATION RESULTS

| Rank | Model | Pre-training data size (# of functions) | Model size | Accuracy (%) |
|---|---|---|---|---|
| | **A**: CodeXGLUE leaderboard for defect detection as of 23 May 2022. | | | |
| 1 | CoTexT [16] | 375M | 220M | 66.62 |
| 2 | C-BERT [17] | 8.1M | 110M | 65.45 |
| 3 | **VulBERTa-MLP** | **2.28M** | **125M** | **64.75** |
| 4 | **VulBERTa-CNN** | **2.28M** | **2M** | **64.42** |
| 5 | PLBART [38] | 680M | 140M | 63.18 |
| 6 | Code2vec [39] | - | - | 62.48 |
| 7 | CodeBERT [37] | 8.5M | 125M | 62.08 |
| 8 | RoBERTa [37] | 2.4M | 125M | 61.05 |
| 9 | TextCNN | - | - | 60.69 |
| 10 | BiLSTM | - | - | 59.37 |
| | **B**: D2A leaderboard for the "function" category as of 23 May 2022. | | | |
| 1 | **VulBERTa-MLP** | **2.28M** | **125M** | **62.30** |
| 2 | **VulBERTa-CNN** | **2.28M** | **2M** | **60.68** |
| 3 | C-BERT [17] | 8.1M | 110M | 60.20 |
| 4 | AugSA-S [26] | - | - | 55.20 |
| 5 | AugSA-V [26] | - | - | 45.60 |

inherited from the pre-trained VulBERTa model plays a crucial role for vulnerability detection.

Model size and pre-training data size commonly play an essential role in learning better code representations. However, VulBERTa contains only 125M parameters, and our pre-training data contains only 2.28M C/C++ functions compared, for example, to CoTexT (375M) and C-Bert (8.5M). We are on par with these approaches by delivering SOTA detection performance with smaller models and pre-training data.

Based on our preliminary analysis of testing and implementing different tokenisation techniques, we believe that our tokenisation approach plays an important role in making syntactic and semantic information easily available to the simple neural architectures that use it for fine tuning.

In fact, model simplicity seems to be part of the solution. VulBERTa-CNN, our simple TextCNN-based approach (with only 2M parameters) has an MCC score of 55.86% on the Draper dataset, higher than the 52% obtained by the complex 3GNN model recently proposed in [40], which uses a Crystal Graph Convolutional Network and Self Attention Pooling.

### *E. Limitations*

Despite the encouraging results, we identify a number of limitations in our approach. A common and still unsolved problem with vulnerability detection datasets is that manual inspection reveals occasional label inaccuracies. While deep learning should be resilient to label noise in training, the presence of noise during testing somewhat undermines the quantitative preformance results. Although our models are relatively small, they are still expensive to train. Due to limited resources, we could not explore significantly larger model configurations and combinations, including performing hyper-parameter sweep. Therefore it is possible that different VulBERTa configurations could achieve a higher performance. Finally, the main limitation of our work is the lack of a systematic attempt to detect novel 0-days vulnerabilities in open-source projects *in-the-wild*. This was due to the challenge of manually reviewing false positives, which we would like to address in future work, leveraging explainability techniques.

## VI. CONCLUSIONS

We have presented VulBERTa, a pre-training model which learns a deep representation of C/C++ code. A key component of our model is a custom tokenisation pipeline which aims to preserve syntactic and semantic knowledge from source code without recurring to overly complicated neural architectures. We used the VulBERTa model as the basis to fine-tune two models for vulnerability detection which achieve state of the art performance on several datasets and benchmarks. This models stand out in virtue of their conceptual simplicity and low complexity as measured in terms of number of parameters.

**Acknowledgments.** This work was partially supported by the Google Cloud Research Credits program with the award GCP19980904.

## REFERENCES


[1] MITRE, "Browse CVE vulnerabilities by date," 2021. [Online]. Available: https://www.cvedetails.com/browse-by-date.php

[2] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[3] L. Zhen et al., "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, San Diego, CA, USA, 2018.

[4] ——, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, p. 1–1, 2021.

[5] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "muvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.

[6] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Vancouver, Canada, 2019.

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

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[10] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations*, 2020.

[11] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," in *2021 International Conference on Learning Representations*, May 2021.

[12] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355.

[13] P. Gage, "A new algorithm for data compression," *C Users J.*, vol. 12, no. 2, p. 23–38, Feb. 1994.

[14] LLVM, "libclang: C interface to clang," 2021. [Online]. Available: https://clang.llvm.org/doxygen/group__CINDEX.html

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

[16] P. Long et al., "CoTexT: Multi-task learning with code-text transformer," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Association for Computational Linguistics, Aug. 2021, pp. 40–47.

[17] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang, and G. Domeniconi, "Exploring software naturalness through neural language models," 2020.

[18] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Orlando, FL, USA: IEEE, 2018, pp. 757–762.

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

[20] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. NY, USA: Association for Computing Machinery, 2015, p. 426–437.

[21] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Transactions on Reliability*, vol. 66, no. 1, pp. 17–37, 2017.

[22] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, Apr. 2021.

[23] R. Rabheru, H. Hanif, and S. Maffeis, "DeepTective: Detection of PHP vulnerabilities using hybrid graph neural networks," in *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022, pp. 1–8.

[24] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *Journal of Systems and Software*, vol. 137, pp. 67–77, 2018.

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

[26] Y. Zheng et al., "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)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 111–120.

[27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[28] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "Dobf: A deobfuscation pre-training objective for programming languages," 2021.

[29] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021.

[30] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751.

[31] PyGithub, "PyGithub: Typed interactions with the GitHub API v3," 2021. [Online]. Available: https://github.com/PyGithub/PyGithub

[32] NIST, "Juliet test suite 1.3," 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[33] National Institute of Standards and Technology, "National Vulnerability Database," 2021, [Accessed November 10, 2021]. [Online]. Available: https://nvd.nist.gov

[34] ——, "Software Assurance Reference Dataset," 2021, [Accessed November 10, 2021]. [Online]. Available: https://samate.nist.gov/SARD

[35] P. et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[36] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005, iJCNN 2005.

[37] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[38] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Jun. 2021, pp. 2655–2668.

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

[40] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software vulnerability detection via deep learning over disaggregated code graph representation," 2021.