Exploring transformers for multi-label classification of Java vulnerabilities

Cláudia Mamede[1], Eduard Pinconschi[1], Rui Abreu[1,2], José Campos[1,3]
[1]Faculty of Engineering of University of Porto, Porto, Portugal
[2]INESC-ID, Porto, Portugal
[3]LASIGE, Faculty of Sciences of the University of Lisbon, Lisbon, Portugal
up201604832@up.pt, up202103584@up.pt, rui@computer.org, jcmc@fe.up.pt

***Abstract*—Deep learning (DL) techniques have demonstrated potential in reasoning complex patterns of vulnerable code from high-level abstractions. Recent advancements in the area, such as the introduction of transformer-based models, like BERT, help overcome the problem of the available vulnerability detection datasets being too small to enable most DL models to capture all relevant patterns. They mitigate the challenge by leveraging knowledge from a general domain to solve problems in specific domains. In this paper, we explore different BERT-based models for multi-label classification of vulnerabilities in Java on a synthetic dataset. The models yield up to 99% in accuracy and 94% in f1-score. We remove biases in the training dataset and observe drops of up to 13% of the f1-score. We further assess the generalizability of the models on realistic samples and notice that one model, in particular, predicted unknown vulnerabilities with an f1-score of nearly 85%.**

***Keywords*—Vulnerability detection; transformer; multi-label classification; bias; generalizability**

I. Introduction

As society becomes more dependent on technology, cyber-attacks are becoming more appealing, and so are the number of incursions and their sophistication. Consequently, companies are investing in shifting-left security to avoid risks of exploitation [36], [27]. Traditional static and dynamic analyses are the standard forms of scans for most companies, but they do not suit this principle. Challenges include delaying code scans until the end of development (as they require the code to be fully developed). In addition, they have high false-negative/false-positive rates and demand developers to manually define features, which is a hurdle for fast development [2].

Hence, recent works explore DL-based systems for vulnerability detection to tackle some of the issues associated with conventional approaches and anticipate the detection as much as possible. The transformer [34] is a recent model that aims to solve sequence-to-sequence tasks while efficiently handling long-range dependencies. It has rapidly become the dominant architecture for Natural Language Processing (NLP), surpassing alternative neural models in performance for natural language understanding and generation tasks [37].

This work explores the use of transformer-based models, specifically BERT-based architectures [5], to identify software vulnerabilities in Java code. We focus our research on multi-label classification as other mutually exclusive classifications may not always provide enough information for the developer regarding the identified flaws.

Thus, we test different model architectures and configurations to discover which ones perform better in synthetic and more realistic scenarios. We also analyze our dataset to find to what extent it impacts the ability of these models to learn and how it can implicitly introduce bias to the models. Then, using natural language techniques, we successfully identified problematic tokens and removed them to minimize these problems. We further investigate to what extent transformer-based models that are find-tuned on synthetic and vulnerable code can predict real vulnerabilities. Lastly, using Software Fault Patterns views[1], we were able to identify groups of vulnerabilities that share similar patterns of faulty computations. We leverage that mapping to assess the ability of transformer-based models to generalize their knowledge to classify unknown vulnerabilities.

Our contributions. This paper explores the usability of the transformer in detecting software vulnerabilities in Java code. We highlight the following contributions:

1) **Introduction of different multi-label classification systems for vulnerability detection in Java code.** This work explores different transformer-based architectures and configurations to find which outputs the best classifications. We also identify the benefits of multi-label classification in this context.
2) **Application of natural language techniques to identity biased scenarios in models trained on source code.** Using the Pointwise Mutual Information (PMI) score [29], it is possible to pinpoint problematic tokens in the datasets, which, consequentially, helps minimise the problem of bias.
3) **Assess the ability of models trained with synthetic data to classify real-world samples.** All models were trained with synthetic data. In this context, we explore the ability of these models to identify vulnerabilities in more realistic contexts.
4) **Explore the transformer-based models fine-tuned for vulnerability detection to discover unknown flaws.** We leverage the multi-label models to find out to what extent they can predict unknown flaws.

To foster reproducibility, all the artefacts are made publicly available on GitHub at https://github.com/TQRG/VDET-for-Java.

Paper organization. The remainder of this paper is structured as it follows. Section II explains fundamental concepts and discusses related work. Section III enumerates the research questions and describes implementation details and decisions made. Section IV presents and analyses the performance

[1]https://samate.nist.gov/BF/Enlightenment/SFP.html

results of the models obtained during validation and testing. Section V discusses the validity of the results presented in the previous section and describes the NLP techniques applied to identify and quantify bias in the model. Finally, section VI presents the conclusions and future work of this study.

II. Background and Related Work

This section introduces different perspectives for classifying software vulnerabilities, gives a brief overview of vulnerability detection approaches, follows up with the state-of-the-art, and finalizes with a description of the BERT architecture.

A. Software vulnerabilities and their classification

Vulnerabilities are usually grouped based on common properties and similarities to facilitate analysis and understanding. This classification helps determine exploitation conditions and prepare adequate countermeasures [18].

Common Weakness Enumeration (CWE)[2] is a well-known form of describing software security weaknesses in architecture, design, and code. All individual CWEs are held within a hierarchical structure, allowing different levels of abstraction. CWEs located at higher system levels provide a broad overview of a vulnerability type; CWEs at lower levels provide finer granularity.

The Software Fault Patterns (SFP) are a clustering of faulty computations (that is, CWEs) that share similar properties [24]. They focus more on the source code faults and the features that can facilitate automation and because of that, they are considered improvements for the CWE classification [38].

B. Vulnerability Detection Systems

More researchers are exploring different approaches to detect software security vulnerabilities. These techniques can be divided into Conventional and Machine learning-based approaches [11].

As the name suggests, the first group includes traditional procedures such as pattern matching and static and dynamic analysis. They require experts to define features, relying on human experience, their level of expertise and domain knowledge [16].

Regarding the second group, there are several forms of categorization. For example, Hanif *et al.* [11] proposes dividing ML-based approaches into supervised, semi-supervised, ensemble, and deep learning. On the other hand, Sonnekalb *et al.* [32] divided them into traditional ML, shallow learning neural networks and DL neural networks. Lastly, Ghaffarian *et al.* [9] proposed a division using a twofold categorization scheme. They first distinguish between approaches that analyze program syntax and semantics and then based on the purpose of the categorization. So, the ones that do not study program syntax or semantics fit into the group of Vulnerability Prediction Models based on Software Metrics. The ones that do can be divided into Vulnerable Code Pattern Recognition and Anomaly Detection Approaches. The techniques that do not fit any of the abovementioned groups belong to the Miscellaneous Approaches.

This paper focuses on the deep learning models currently trending in the vulnerability detection community [11], specifically the transformers. These DL models use their neural networks to extract features automatically and leverage knowledge from a general domain to solve problems in other specific domains, relieving experts from labour-intensive and possibly error-prone feature engineering tasks [22], [2], [11].

C. Transformer-Based Models

The literature prompted the Transformer architecture for vulnerability detection even before its application for the task. The adoption of the Transformer architecture has been encouraged due to its strong capacity of understanding natural language [22], its transfer learning capability [11], and overcoming existing challenges of previous architectures, such as the missing recurrence [32]. The transfer learning mechanism allows learning a specific task from a small dataset by reusing previous generic knowledge learned from a vast dataset with quality and reliable data.

Ziems *et al.* [39] are the first authors to introduce the use of Transformer-based models to solve the problem of vulnerability detection. They use the traditional BERT architecture pretrained on written English to identify flaws in computer code. They built a multi-class model, to identify CWEs in software. They focus on C/C++ programming languages and crafted their own dataset to test the model, addressing more than 100 CWEs. They use file-level granularity.

Hin *et al.* [12] leverage the pretrained CodeBERT model to extract features from real-world C/C++ code at the statement and function level granularity. In addition, the authors use Graph Neural Networks to preserve important dependency information from the data. For classification, the authors opt for a multi-layer perceptron model to simultaneously learn from the function-level and statement level code.

Hanif *et al.* [10] explores RoBERTa for vulnerability detection on real-world C/C++ projects. They first learned code representations via Masked Language Modelling (MLM), and then connected the pretrained model to a multi-layer perceptron and convolutional neural network to fine-tune it for vulnerability detection. This work assesses model performance for binary and multi-class classifications and compares results with other well-known projects, such as VulDeePecker [21] and μVulDeePecker [40].

Le *et al.* [19] use ML models to predict CVSS metrics of vulnerable functions, which helps prioritize critical software vulnerabilities. In particular, they train CodeBERT to build the vocabulary for feature extraction methods. They also crafted a dataset with 1782 vulnerability functions in 200 open-source Java projects.

Thapa *et al.* [33] compare the performance of multiple DL-based systems for vulnerability detection, from which we emphasize the traditional BERT architecture, DistilBERT, RoBERTa and CodeBERT. They focus on C/C++ programming languages and use VulDeePecker's released dataset to

[2]https://cwe.mitre.org/

train these architectures, considering binary and multi-class classifications.

Fu *et al.* [8] propose LineVul, a transformer-based line-level vulnerability prediction approach to address the limitations of graph-based neural networks. Using a C/C++ real-world dataset, they were able to show that their architecture had more accurate and more cost-effective predictions than graph-based strategies and highlighted the potential of the model in realistic scenarios.

D. Multi-label vs. mutually exclusive class classifications

In binary classifications, one wants to discover if a code sample is either vulnerable or non-vulnerable [21]. This classification does not help developers solve the problem as it lacks relevant details, such as the type of flaw. On the other hand, multi-class classification only identifies the type of flaw [40], assuming that they can always be exploited, i.e., they are vulnerable. This particular type of classification may not be accurate due to good source/bad sink and bad source/good sink scenarios.

For example, the method *action* in Listing 1 is vulnerable to CWE-190 (Integer Overflow or Wraparound), as there is no verification to prevent overflow from occurring. Assuming *data* ranges between `[MIN_VALUE, MAX_VALUE]`, an integer overflow scenario would only happen if `data = MAX_VALUE`. On the contrary, the flaw is tackled if *data* is a hardcoded non-max or if there was a check on the value of *data* before using the method. So, even though the construction still relates to a particular CWE, the flaw may be solved (in)voluntarily in other code sections.

```
public void action(byte data) throws Throwable {

    /* POTENTIAL FLAW: data == Byte.MAX_VALUE */
    byte result = (byte)(++data);

    IO.writeLine("result: " + result);
}
```

Listing 1. Example of method vulnerable to CWE-190.

In the real world, where companies are rushing to develop and release innovative software before competition, it is essential to distinguish between truly concerning issues so that security experts can efficiently solve them without hindering the speed of deployment. Hence, models should identify the type of flaw and understand if it is exploitable. This can be described as a multi-label classification problem, in which one wants to predict a particular CWE and if the code is vulnerable or not.

Conventional strategies to solve multi-label classification problems include problem transformation, problem adaptation, and ensemble methods. Fallah *et al.* [6] proposes a transformer-specific technique to solve the problem - (global) threshold selection.

E. BERT model

BERT [5] is a transformer-based model which uses only the encoder blocks from the original architecture. It is bidirectional and uses attention mechanisms.

Tokenization and model input: BERT models expect 1-dimensional vectors with a maximum length of 512 numerical tokens as input. Each token is an integer that ranges from 0 to vocabulary size (in this case, *vocabulary size = 30,522*), encoded with a word-piece tokenizer [28]. This tokenizer receives an input sequence and decides, based on token frequency, whether to keep every word as a whole word, split it into sub-words, or decompose it into individual characters.

Model architecture: The BERT model is composed of twelve layers (encoder blocks), with twelve attention heads [5]. The sequence flow inside the stack is not trivial, but the process is synthesized in Figure 1. After tokenization, each token is embedded into a 768-long vector, generating an input embedding for each token. Using different linear projections, each embedding vector produces twelve triplets of 64-long vectors (the *key*, *query*, and *value* vectors). Then, each triplet is forward to its own self-attention head that calculates the scaled dot-product attention for the received triplet, generating a 64-long vector that encodes information regarding the specific token. The outputs from all self-attention heads are concatenated together, and the result is projected through a feed-forward layer. The result is a sequence of transformed embedding vectors that go through the same layer structure eleven more times, improving representation each time. Finally, the model can output the last hidden state (all the improved input embeddings outputted by the last encoder block), multiple hidden states (the input embeddings outputted by a group of encoder blocks), or the first input embedding of the last encoder block, that is, the *pooler_output* (which matches the representation of the *[CLS]* token).

III. Methodology

This section introduces the research questions and describes implementation details and decisions made that allow the exploration transformers for vulnerability detection.

A. Research Questions

Our goal is to investigate the ability of BERT-based models in identifying software vulnerabilities in Java code. We aim at answering the following Research Questions (RQs):

- **RQ1** How do different output configurations impact the learning of BERT-based models?
- **RQ2** Which BERT-based model configuration achieves better vulnerability identifications?
- **RQ3** To what extent does implicit bias in datasets affect the ability of the model to learn?
- **RQ4** How do BERT-based models perform when exposed to real-world samples?
- **RQ5** To what extent BERT-based models can predict unknown vulnerabilities?

B. Dataset Selection

To feed a DL model, appropriate data should be collected, filtered and labelled and later transformed into a suitable format. In the existing literature, the majority of available datasets for vulnerability detection are suited for C/C++ programming

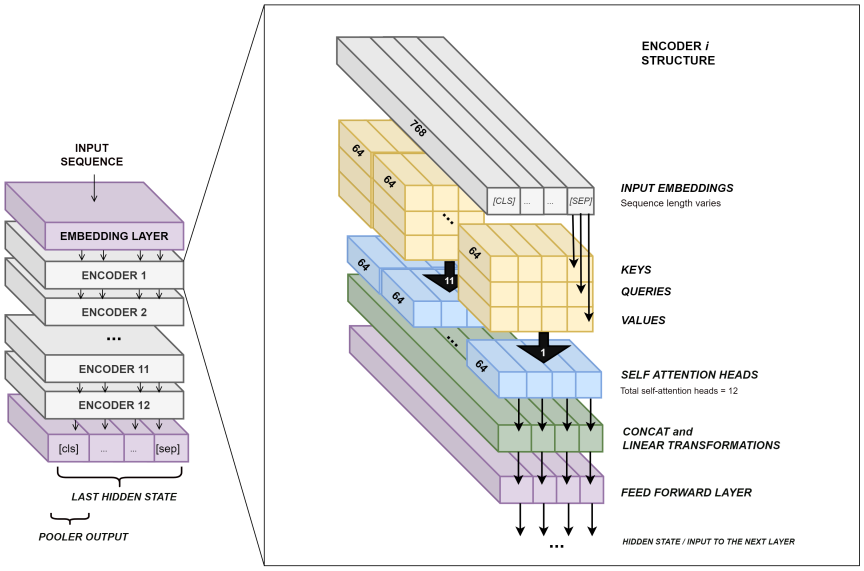

Figure 1. BERT architecture.

languages [2], [22], [32]. Hirsch *et al.* [13] identified 73 benchmarks for evaluating debugging approaches. Only 6 of the 73 benchmarks contain vulnerable Java programs and the majority include a relatively small set of programs or vulnerabilities. The NIST SAMATE Juliet Test Suite for Java[3] is a collection of synthetic test cases written in Java that includes bad-source/good-sink and good-source/bad-sink scenarios. Hence, models can be trained to distinguish both cases (that is, vulnerable or non-vulnerable) and improve their classifications. This test suite further identifies the related CWE for each code sample.

We use the curated dataset developed in [23] to train the models. It is built from the Juliet Test Suite for Java and uses function-level granularity. It contains 113 898 methods, 80 269 of which are non-vulnerable and 33 629 are vulnerable. Each sample is related to only one particular CWE. Figure 2 illustrates the distribution of CWEs in this dataset.

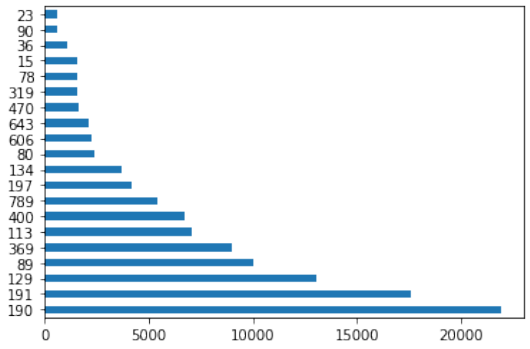

Figure 2. CWE distribution in the dataset.

[3]https://samate.nist.gov/SARD/test-suites/111

C. Model implementation

JavaBERT [4] and CodeBERT [7] are BERT-based models. The former was trained on nearly 3 million Java files retrieved from open-source projects on GitHub, on which a certain amount of tokens are masked and must be predicted (Masked Language Modeling task). On the other hand, CodeBERT was trained on both natural and programming languages. This model is trained with a dataset [14] containing 6.4 million unimodal codes in different programming languages, including Java. Feng *et al.* [7] trained CodeBERT with two objectives: Masked Language Modeling and Replaced Token Detection.

As far as we know, JavaBERT is the only model trained on just the Java programming language so it is relevant to explore its use in this context. Similarly, we consider CodeBERT due to its prominence in recent vulnerability detection literature [25], [26], and because of its promising performance with small sized datasets on different tasks. We use the Hugging Face and PyTorch libraries for this purpose. Regarding model implementation, we address the following components:

1) Model configuration: BERT-based architectures can output different data structures that serve as input for the new classification layers, influencing the final predictions. So, we investigate two configurations to find out which one produces the most accurate predictions for vulnerability detection:

- **4HS:** BERT authors confirmed that combining the outputs of the last four hidden layers produces the best results [5]. So, m4HS uses the concatenation of the first tokens (corresponding to the representation of the *[CLS]* tokens) of the last 4 hidden states (due to computational constraints, we cannot use the complete state output of the encoders).
- **PO:** The *pooler output* is usually the go-to solution in most cases, providing reasonable results in other problems. Therefore, mPO uses the *pooler output* as its output.

2) Loss function: It computes the distance between the current output of the algorithm and the expected output. Kurate *et al.* [17] investigated how different loss functions influence multi-label classifications and concluded that using Binary Cross-Entropy (BCE) (combined with a sigmoid activation function in the output layer) achieved superior results in comparison to other approaches. We follow the prior work and use BCE with Logits Loss function (implemented with *BCEWithLogitsLoss* in PyTorch).

3) Classification layer: The classification layer has 22 output neurons, corresponding to each label (CWEs and "Vulnerable/NonVulnerable"). Each output neuron (and by extension, each label) is considered to be independent of each other. After applying an activation function over the logits to limit the values to [0,1], it is possible to apply Fallah *et al.*'s threshold strategy [6] and obtain the final predictions. Using a *threshold = 0.5*, labels with probabilities higher than that are selected, and those below are ignored. Hence, the model should be able to identify more than one CWE in code and confirm its exploitability.

D. Model training and validation

The dataset is split using a split ratio of 80:10:10 for training, validation (obtained from the training data at each epoch) and test sets. In this context, having a validation set is relevant because we are adjusting hyperparameters at each iteration, based on loss values. There are no repeated samples through sets, and each collection has representatives from all classes.

After, both sets are subdivided into batches of size 12 (due to computational limitations, this is the maximum batch size supported). We use a smart-batching strategy to avoid redundant computations and speed training. So, instead of grouping our samples in batches of a fixed size, we adapted Chris McCormick's Uniform Length Batching Strategy[4] to fit our data. By sorting the dataset by sequence length and group samples of similar sizes in batches of 12, we reduced the number of tokens in our samples by 66% and sped up the training process.

Lastly, we train and validate all models for 10 epochs (maximum number of completed iterations for all models due to computational limitations) before their evaluation.

E. Evaluation

1) Diagnosing models' behaviour with learning curves: A learning curve is a mathematical representation of the learning process as a task repetition occurs [1]. It helps in model selection and comprehending whether or not the models can capture meaningful features of the training data [35]. For the problem of vulnerability detection, we intend to minimize loss during training. Train loss provides insight into how well the model learns, and the validation loss helps understand how models generalize. Thus, we analyze the learning curves to identify promising models for vulnerability detection and eliminate architectures that would not perform well.

[4]https://mccormickml.com/2020/07/29/smart-batching-tutorial/

2) Model evaluation, testing with synthetic samples: Because the number of samples per label differs in the dataset, accuracy cannot be considered the main metric for performance assessment. Therefore, weighted average values for precision, recall and f1-score are computed, considering each class's support. In addition, mean false-positive (FPR) and false-negative (FNR) rates are also calculated.

3) Analysis of the training data: When training, validation, and testing sets share the same data source, implicit biases in them may impact the ability of models to learn [2]. Thus, we apply natural language techniques to software code, namely the Pointwise Mutual Information (PMI) [29], which enables one to discover problematic tokens in datasets. After that, we normalize the dataset to minimize its impact on model performance.

4) Model evaluation, testing with real-world samples: Russell *et al.* [30] stated that training models with just synthetic samples is insufficient as precision may severely decrease when facing real-world scenarios. We look into this issue and test the normalized models with real-world samples with the dataset from [20]. Since the dataset does not include the CWE identifier of the CVEs, we had to map each CVE to its respective CWE. We accomplished that by leveraging the dump of all the CVE published[5]. Then we selected the samples by the CWE that our model could identify. We end up with a the test set contains only 70 vulnerable methods, targeting 8 known CWES[6]. At last, we evaluate the normalized datasets with the resulting testing set.

- **Synthetic vs real-world datasets:** Synthetic data is similar to real-world data but not collected by real-life means; instead, it is programmatically generated. A clear advantage of it is the possibility to reproduce vulnerabilities that rarely occur and are hard to find in reality. On the other hand, synthetic code, particularly single-sourced synthetic samples, follow the same style and structure [30]. In the context of our problem, the dataset used to train the models is synthetic, and consequently, the samples share common elements, such as variable/method names and code structure. On the other hand, the real-world samples we are using for testing do not follow specific rules for naming variables/methods nor have a particular code structure. So, after minimizing the influence of these traits through the normalization of the synthetic test set, we expect models to classify real-world samples equally successfully.

5) Generalizability: Intuitively, generalizability measures how applicable the results of a study are to a broader group. In this context, a model is said to have good generalizability if it can be successfully applied to identify other unknown flaws.

To assess the generalizability of the models we consider testing them with samples of unknown vulnerability types that are related with the kind of vulnerabilities these have

[5]https://www.cve-search.org/dataset/

[6]CWE113, CWE190, CWE319, CWE400, CWE470, CWE78, CWE89 and CWE90

been trained. This is possible as certain vulnerabilities share similarities and CWE definitions capture the relationship between them. Furthermore, SFPs map to CWE elements and these arrange vulnerabilities by common faulty computations. We leverage from the CWE List the SFP view that maps CWE identifiers with SFP clusters. In Table I, we list the CWEs in the training dataset along with their respective SFP Secondary Cluster. Since CWE-129, CWE-789, and CWE-690 are not listed in the SFP view, we omit them in the table. We also omit CWE-400, CWE-470, and, CWE-319 since their mapping is one-to-one and listing them does not provide any additional information. As observed, the majority of the training dataset is composed of vulnerabilities related to "*Glitch in Computation (CWE-998)*" and "*Tainted Input to Command (CWE-990)*".

TABLE I
SFP SECONDARY CLUSTERS OF TOP CWES IN THE TRAINING DATASET.

SFP Secondary Cluster	CWE ID	#Samples
Glitch in Computation	CWE-190	4862
	CWE-191	3971
	CWE-369	1928
	CWE-197	1259
Tainted Input to Command	CWE-89	2198
	CWE-113	1580
	CWE-134	836
	CWE-80	771
	CWE-78	493
	CWE-643	459
	CWE-90	232
Tainted Input to Variable	CWE-606	482
	CWE-15	478
Path Traversal	CWE-36	342
	CWE-23	212

As testing set for this experiment we leverage the CWE-611 and CWE-79 samples in the dataset introduced by T. Le *et al.* [20]. These two kinds of vulnerability are unknown to the model but fall under the "*Tainted Input to Command (CWE-990)*" cluster, the second most representative cluster in our dataset. We also assess the impact of the amount of data on the model by classifying unknown CWEs that are relatable to training data but in less amount. For that, we select CWE-22 as it belongs to the "*Path Traversal (CWE-981)*" cluster with fewer samples in the training set but with many samples in *et al.* [20]. Furthermore, we also investigate the ability of the models predicting unknown and unrelatable vulnerabilities to the training dataset. For that, we select as candidate CWE-287 that belongs to the "*Authentication Bypass (CWE-947)*" cluster.

We constructed three small test sets. *ds_611_79*, which contains only CWE-611 and CWE-79 vulnerabilities, has 239 samples. *ds_22*, containing only CWE-22 representatives, has 179 samples. Lastly,*ds_287*, targeting only CWE-287 vulnerabilities, has 159 samples.

IV. RESULTS

This section presents our results and findings under various experiments. We use these results to answer the research questions enumerated in Section III-A.

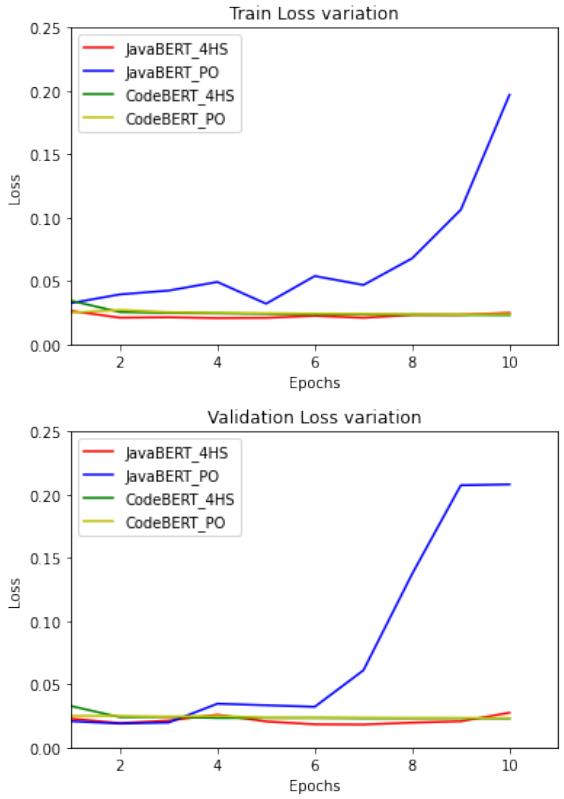

Figure 3. Learning curves (loss variations) during training (left) and validation (right) for all models.

RQ1: How do different output configurations impact the learning of BERT-based models?

Figure 3 illustrates the learning curves, for the first 10 epochs, of the four models. We identify the following in the graphics:

- **JavaBERT_4HS, CodeBERT_4HS and CodeBERT_PO have a good fit.** These models have low loss values, with training and validation curves decreasing slightly in the beginning and flatten, with a small gap between them.
- **JavaBERT_PO is (most likely) underfit.** Training loss has a minimum value of 0.023 (epoch 2) and a maximum value of 0.209 (epoch 10). Validation loss ranges from 0.019 (epoch 2) to 0.208 (epoch 10). Loss is a subjective metric that highly depends on the problem, and, in this context, a loss value of 0.2 is tremendous. The model has a low training loss at the beginning that gradually increases. Validation loss follows a similar pattern. We hypothesize that with more iterations, both values would drop to 0, and all evidence would point to an underfit scenario.

> ***Finding 1.*** *The pooler output configuration compromises the transfer learning capabilities of the JavaBERT model.*

RQ2: Which BERT-based model configuration achieves better vulnerability identifications?

We observe from the learning curve of JavaBERT_PO that the model is not stable, having considerably high values in the last three epochs and being incapable of learning the training set. As a result, we excluded it from subsequent experiments.

Table II lists the performance result for the most accurate epoch during the training of the JavaBERT and CodeBERT models for the appropriate configurations. The "#Epoch" column indicates the number of the epoch with the best training results. The "Acc." column shows accuracy. The "$\overline{w}F1$", "$\overline{w}Precision$", and "$\overline{w}Recall$" columns describe the weighted average of F1, precision, and recall, respectively. The "$\overline{FNR}$" and "$\overline{FPR}$" columns indicate the average FPR and FNR, respectively.

The JavaBERT model with the 4HS configuration presents the highest performance metrics. It achieves an accuracy of almost 99%, a precision of 95%, and a recall of 93%. This means that when the model predicts a label that is in fact the expected label (good precision), and when a particular label is expected, the model usually predicts it right (good recall). Consequently, the f1-score, that represents a balance between precision and recall, has also a high value (94%). This model is stable, with low loss values (ranging from 0.02095 to a maximum value of 0.04255) and high accuracy.

CodeBERT_4HS and CodeBERT_PO perform very similarly, and both have an f1-score of 93%. The models are also stable, with low loss values (ranging from 0.023 to 0.033 for CodeBERT_4HS and from 0.023 to 0.025 for CodeBERT_PO). Despite the difference between models not being as expressive as with JavaBERT, using the 4HS configuration is still the best approach.

> ***Finding 2.*** *Combining the outputs of the last four hidden layers yields more accurate predictions.*

RQ3: To what extent does implicit bias in datasets affect the ability of the model to learn?

The Pointwise Mutual Information (PMI) [29] score permits the discovery of problematic tokens in the dataset. The results, depicted in Table III, prove that some tokens, such as "bad" and "good" are still present and are tightly related to the "Vulnerable" and "Non Vulnerable" classes. We hypothesize that these tokens are most likely over-represented, causing the model to make wrong predictions in samples. The same study was made for all the other classes. Similarly, there are problematic tokens in some of the CWE classes. This time, it is confirmed that the problematic tokens consist of the numbers of the CWEs shown in Table III. The complete list of problematic tokens is available in our GitHub repository.

Then, normalized the dataset and repeated training for JavaBERT_4HS, CodeBERT_4HS and CodeBERT_PO. observed some performance differences, which are represented in Figure 4.

In both cases, performance severely decreases. Despite JavaBERT_4HS having a slightly higher f1-score when trained on a non-normalized dataset, both models end up with the same values after normalization. More specifically, both models decreased almost 12% in the f1-score metric.

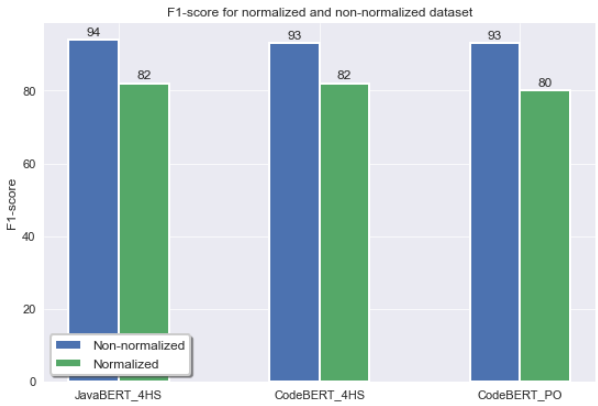

Figure 4. Performance difference before and after dataset normalization.

> ***Finding 3. We can use the Pointwise Mutual Score (PMI) to identify problematic tokens in software code.***

> ***Finding 4. Removing token that bias the model substantially reduces the f1-score (up to 13%)***

RQ4: How do BERT-based models perform when exposed to real-world samples?

In Table IV, we list the performance of the models on real samples. For each metric, we compare them with their previous performance results reported in Table II. We can observe that the accuracy is high, while the f1-score and recall have low values, and precision is reasonable.

On the one hand, recall identifies the proportion of actual positives which were correctly classified. In this case, both models struggle to determine (the majority of) positive instances in the dataset, outputting more false negatives. The models can recognise vulnerable patterns but stumble in selecting the type of vulnerability (CWE) and that is why recall has lower values. On the other hand, precision analyses the number of samples correctly predicted as positive. The models do not identify many false positives and because of that precision has more reasonable values for both models.

To simplify, we can understand these models as being "picky". They are usually correct whenever they identify a vulnerable piece of code and a particular CWE (high recall).

TABLE II
PERFORMANCE RESULTS FOR JAVABERT AND CODEBERT WITH DIFFERENT MODEL CONFIGURATIONS.

Model	#Epoch	Acc.	$\overline{w}F1$	$\overline{w}Precision$	$\overline{w}Recall$	$\overline{FNR}$	$\overline{FPR}$
JavaBERT_4HS	8	98.93%	94.0%	95.0%	93.0%	7.12%	0.98%
CodeBERT_4HS	10	98.68%	93.0%	95.0%	91.0%	12.28%	1.02%
CodeBERT_PO	9	98.67%	93.0%	95.0%	91.0%	12.39%	1.06%

TABLE III
TOP TOKENS WITH HIGHEST PMI SCORES FOR EACH LABEL.

Label	Token	PMI
Vuln.	##ad	0.98
	bad	0.88
	##hor	0.61
	http	0.60
Non-Vuln	good	1
	#BS	1
	##GS	1
	false	1

Label	Token	PMI
CWE-134	##13	1
	#4	1
CWE-15	##15	1
CWE-23	##23	1
CWE-400	##40	1
CWE-470	##47	1
CWE-643	##64	1
CWE-80	##47	1

However, they still "prefer" missing predictions, reflecting the high FNR values.

> ***Finding 5. Models trained on synthetic data have a tendency to identify true vulnerable samples as non-vulnerable.***

RQ5: To what extent BERT-based models can predict unknown vulnerabilities?

Detecting unknown real-world vulnerabilities is paramount for vulnerability detection models. With this research question, we investigate if synthetic code's syntactic and semantic characteristics that the models learned are adequate for predicting unknown vulnerabilities.

Although the models cannot output labels they have not seen before (like other CWEs), we hypothesize they can still pinpoint "Vulnerable" and "Non Vulnerable" patterns. Hence, we focus on these last two labels, recalling the models are fine-tuned for multi-label classification and, consequently, the labels are not mutually exclusive (i.e., probabilities do not sum up to 1).

Table V lists the performance results of all models for the three test sets. For *ds_611_79*, the CodeBERT model suffers a considerable drop in the performance metrics for both configurations. The low accuracy and high FN/FP rates indicate that fine-tuning CodeBERT models with synthetic and vulnerable code do not suit vulnerability detection. In contrast, JavaBERT manages to maintain a relatively good accuracy with tolerable FN/FP rates. For *ds_22*, CodeBERT models once again perform poorly. On the contrary, JavaBERT drops performance, but it can still deliver accuracy values above 50%. We believe this decline is because the original training set had fewer examples of vulnerabilities related to the *Path Traversal* SFP. Consequently, models have not learnt enough to make a better prediction.

> ***Finding 6. JavaBERT fine-tuned on synthetic and vulnerable data can successfully predict unknown and relatable vulnerability types.***

On the other hand, for *ds_287*, all models achieve low-performance metrics, which indicates their inability to detect unknown and unrelatable vulnerabilities with patterns different from the ones learned.

> ***Finding 7. BERT-based models fine-tuned on synthetic and vulnerable data are unable to predict unknown and unrelatable vulnerability types.***

V. DISCUSSION

Considering the results for the models tested in realistic contexts, reported in Table IV, we observe that all of them suffer from high FNR.

Although both are risky, FP are only annoying as reviewers have to filter them to identify the relevant ones. Instead, FN are more dangerous as they lead to a false sense of security and can be neglected [27], [3]. Thus, vulnerability detection systems with high false-positive rates may not be usable. Similarly, systems with high false-negative rates may not be useful [21].

Therefore, it is essential to lower the FNR so that any of these models can be integrated into a future tool. This could be accomplished by introducing more training samples that could be either synthetic or realistic. On the one hand, synthetic samples make it possible to generate vulnerabilities that rarely occur and are hard to find. On the other hand, models trained on this kind of data are usually used to a specific code structure/pattern, which could severely impact how models learn and reduce performance [2], [30].

A. Threats to validity

Despite the positive results, it is important to approach them with some reservations as there are some threats to the validity of the model. They are:

1) **Reduced sample size and imbalanced training data:** All DL models highly rely on the quality and quantity of a dataset. The training samples are all synthetic and single-sourced, with 80% of them being non-vulnerable and only 20% vulnerable. As Chakraborty *et al.* [2] mentioned, a model trained on such an uneven dataset is most likely biased towards the majority class.

TABLE IV
PERFORMANCE RESULTS FOR THE MODELS TESTED WITH REAL-WORLD SAMPLES

Model	Acc.	$\overline{w}F1$	$\overline{w}Precision$	$\overline{w}Recall$	$\overline{FNR}$	$\overline{FPR}$
JavaBERT_4HS	90.06% (-8.87%)	44.0% (-50%)	68.0% (-27%)	35.0% (-58%)	36.03% (+28.91%)	4.12% (+3.14%)
CodeBERT_4HS	86.88% (-11.8%)	23.0% (-70%)	82.0% (-13%)	23.0% (-68%)	37.74% (+24.46%)	5.39% (+4.37%)
CodeBERT_PO	85.86% (-12.81%)	20.0% (-73%)	59.0% (-36%)	12.0% (-79%)	39.52% (+27.13%)	9.85% (+8.79%)

TABLE V
PERFORMANCE RESULTS FOR ALL MODELS TESTED WITH UNKNOWN VULNERABILITIES.

			Class "Vulnerable"						Class "Non Vulnerable"			
Model	**Dataset**	**#Samples**	**Acc.**	FNR	FPR	$F1$	$Prec.$	$Recall$	**Acc.**	FNR	FPR	$F1$
JavaBERT_4HS	*ds_611_79*	239	74.47%	25.52%	0%	85.37%	100%	74.48%	75.73%	0%	24.26%	undef.
	ds_22	179	55.86%	44.13%	0%	71.69%	100%	55.87%	60.89%	0%	39.10%	undef.
	ds_287	159	38.36%	61.63%	0%	55.45%	100%	38.36%	44.03%	0%	55.97%	undef.
CodeBERT_4HS	*ds_611_79*	239	17.57%	82.4%	0%	29.89%	100%	17.57%	18.82%	0%	81.17%	undef.
	ds_22	179	12.85%	87.15%	0%	22.77%	100%	12.85%	15.08%	0%	84.92%	undef.
	ds_287	159	4.40%	95.60%	0%	8.43%	100%	4.40%	4.40%	0%	95.60%	undef.
CodeBERT_PO	*ds_611_79*	239	18.41%	81.59%	0%	31.10%	100%	18.41%	15.89%	0%	84.1%	undef.
	ds_22	179	19.55%	80.45%	0%	32.71%	100%	19.55%	20.67%	0%	79.33%	undef.
	ds_287	159	13.21%	86.79%	0%	23.33%	100%	13.20%	13.84%	0%	86.13%	undef.

Prec. - Precision; undef. - undefined;

2) **Use cross-validation:** Although we use a validation set to analyse the learning curves of the models, we do not perform cross-validation. It would permit the analysis of each fold in-depth and give more insights into how well the model would perform with unseen data.
3) **Preprocessing:** In this work, we identified problematic tokens that added bias and removed them to improve performance. Other preprocessing techniques, such as sampling, massaging, reweighing and optimized data transformation, may be applied to the dataset to improve results [15].
4) Current machine-learning based software vulnerability detection methods are primarily conducted at the function-level. However, a key limitation of these methods is that they do not indicate the specific lines of code contributing to vulnerabilities [12].

VI. CONCLUSIONS AND FUTURE WORK

In this work we leveraged the transformer architecture to address the shortcomings of current vulnerability detection systems. We explore multi-label classification in this context as it provides more information regarding the security conditions of software code compared to other mutually exclusive strategies. For this, we built four multi-label classification systems, using the JavaBERT and CodeBERT architectures and altering their configuration. We evaluated their performance with a synthetic test set and all models performed well, achieving high performance metrics, up to 94% (f1-score).

Then, we searched with PMI for problematic tokens in the dataset and remove them as bias promote models with high accuracy because of features specific to that dataset. By normalizing the dataset and repeating training, we verified that there was a reduction of 13% in the f1-scores, which demonstrates that the dataset was, in fact, adding bias to the models.

Lastly, we evaluated the performance of the models under more realistic contexts. We concluded that, JavaBERT_4HS, a model employing the 4HS configuration, is more stable and capable of performing well over synthetic and real-world samples. On the contrary, CodeBERT models performed poorly in realist contexts. Moreover, we assessed their ability to generalize and predict other unknown vulnerabilities. We found that only JavaBERT_4HS could predict unknown and related vulnerabilities with and accuracy of 74.47% and better than random chance.

We identify the following potential directions for future researchers:

1) A potential strategy to improve model performance includes mixing real and synthetic data collected through known vulnerability databases and open-source repositories to generate a more extensive training set [31].
2) Currently, the distribution of samples per CWE is highly imbalanced, as illustrated in Figure 2. In addition, around 80% of the training samples are non-vulnerable which also impacts model knowledge. To improve results, the dataset must be balanced.
3) All real-world samples from the test sets are vulnerable. Non vulnerable data should be collected so that a more complete analysis of the models can be conducted.
4) Explore the ability of the models to identify more than one CWE per code sample. Although we expect the model to perform well under these scenarios, we still need to assess the veracity of this claim. Hence, a test set containing the proper information should be curated for this purpose.
5) As mentioned in Section III-C1, we could not use the

concatenation of the last four hidden states as explained by BERT authors due to computational constraints. Considering the good performance of JavaBERT_4HS, it would be interesting to evaluate a model using the "complete" configuration.

6) Evaluate the performance of the models with other test sets, e.g. the OWASP benchmark[7].

ACKNOWLEDGMENTS

This work was supported by FCT under the PRT/BD/152197/2021 scholarship (funded by the CMU Portugal Program) and by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

REFERENCES

[1] Michel Jose Anzanello and Flavio Sanson Fogliatto. Learning curve models and applications: Literature review and research directions. *International Journal of Industrial Ergonomics*, 41(5):573–583, 2011.

[2] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? 9 2020.

[3] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy Magazine*, 2:76–79, 11 2004.

[4] Nelson Tavares de Sousa and Wilhelm Hasselbring. Javabert: Training a transformer-based model for the java programming language, 2021.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[6] Haytame Fallah, Patrice Bellot, Emmanuel Bruno, and Elisabeth Murisasco. Adapting transformers for multi-label text classification. In *CIRCLE (Joint Conference of the Information Retrieval Communities in Europe) 2022*, 2022.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[8] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022.

[9] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques. *ACM Computing Surveys (CSUR)*, 50:1 – 36, 2017.

[10] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection, 2022.

[11] Hazim Hanif, Mohd Hairul Nizam Bin Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *J. Netw. Comput. Appl.*, 179:103009, 2021.

[12] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks, 2022.

[13] Thomas Hirsch and Birgit Hofer. A systematic literature review on benchmarks for evaluating debugging approaches. *J. Syst. Softw.*, 192:111423, 2022.

[14] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019.

[15] Faisal Kamiran and Toon Calders. Data preprocessing techniques for classification without discrimination. *Knowledge and information systems*, 33(1):1–33, 2012.

[16] Seokmo Kim, R. Kim, and Young Park. Software vulnerability detection methodology combined with static and dynamic analysis. *Wireless Personal Communications*, 89:1–17, 08 2016.

[17] Gakuto Kurata, Bing Xiang, and Bowen Zhou. Improved neural network-based multi-label classification with better initialization leveraging label co-occurrence. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 521–526, 2016.

[7]https://owasp.org/www-project-benchmark/

[18] Triet H. M. Le, Huaming Chen, and M. Ali Babar. A survey on data-driven software vulnerability assessment and prioritization. 7 2021.

[19] Triet Huynh Minh Le and M. Ali Babar. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 621–633, 2022.

[20] Triet Huynh Minh Le and M. Ali Babar. On the use of fine-grained vulnerable code statements for software vulnerability assessment models. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 621–633, 2022.

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. Internet Society, 2018.

[22] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[23] Cláudia R. Mamede. A transformer-based ide plugin for vulnerability detection. Master's thesis, Faculty of Engineering of University of Porto, Porto, Portugal, 2022.

[24] Nikolai Mansourov and Djenana Campara. *System Assurance: Beyond Detecting Vulnerabilities*. Morgan Kaufmann, First edition, 2010.

[25] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs, 2021.

[26] Cong Pan, Minyan Lu, and Biao Xu. An empirical study on software defect prediction using codebert model. *Applied Sciences*, 11(11):4793, 2021.

[27] Frank Piessens. *The Cyber Security Body of Knowledge v1.0, 2019*, chapter Software Security. University of Bristol, 2019. KA Version 1.0.

[28] Abigail Rai and Samarjeet Borah. Study of various methods for tokenization. In *Applications of Internet of Things*, pages 193–200. Springer, 2021.

[29] Francois Role and Mohamed Nadif. Handling the impact of low frequency events on co-occurrence based measures of word similarity. In *Proceedings of the international conference on Knowledge Discovery and Information Retrieval (KDIR-2011). Scitepress*, pages 218–223, 2011.

[30] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. 7 2018.

[31] Tim Sonnekalb, Thomas S. Heinze, and Patrick Mäder. Deep security analysis of program code: A systematic literature review. *Empirical Softw. Engg.*, 27(1), jan 2022.

[32] Tim Sonnekalb, Thomas S. Heinze, and Patrick Mäder. Deep security analysis of program code. *Empirical Software Engineering*, 27:2, 1 2022.

[33] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection: Performance, model's security and platforms, 2022.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 6 2017.

[35] Tom Viering and Marco Loog. The shape of learning curves: a review, 2021.

[36] Laurie Williams. *The Cyber Security Body of Knowledge v1.0, 2019*, chapter Secure Software Lifecycle. University of Bristol, 2019. KA Version 1.0.

[37] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. pages 38–45. Association for Computational Linguistics, 2020.

[38] Yan Wu, Irena Bojanova, and Yaacov Yesha. They know your weaknesses–do you?: Reintroducing common weakness enumeration. *CrossTalk*, 45, 2015.

[39] Noah Ziems and Shaoen Wu. Security vulnerability detection using deep learning natural language processing. 5 2021.

[40] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, page 1–1, 2019.