Semiconductor Optoelectronics
ISSN: 1001-5868

FINDING SOFTWARE FLAWS WITH DEEP NEURAL NETWORKS: A COMPARISON AND OPTIMIZATION

Gayatri Bajantri
Department of Computer Science & Engineering, BLDEA's V. P. Dr. P. G. Halakatti College of Engineering & Technology, Vijayapur-586103, Karnataka, INDIA
cse.gayatri@bldeacet.ac.in

Dr. Noorullah Shariff
Department of Artificial Intelligent and Machine Learning, Ballari Institute of Technology and Management, Ballari-583104, Karnataka, INDIA
cnshariff1@gmail.com

Abstract— The present detection performance has to be substantially enhanced because of the numerous flaws in complex software. Several methods of finding security flaws in code have been presented. There is a family of methods that use DL its means Deep Learning techniques that show promise. This study is an attempt to use Code BERT, a deep contextualised model, as an embedding solution to make it easier to find security flaws in C open-source projects. Code BERT's use for code analysis unearths previously unseen patterns in software, which may improve the efficiency of subsequent processes like software vulnerability identification. Based on BERT's design, Code BERT offers a bidirectional stacked encoder of transformers to make it easier to study security flaws in code via long-range dependency analysis. Furthermore, transformer's multihead attention method allows for several essential variables of a data flow to be concentrated on, which is vital for evaluating and tracking potentially sensitive data faults, and ultimately results in optimum detection performance. Word2Vec, Glove, and Fast Text are four mainstream-embedding approaches that are compared to one another in order to determine the efficacy of the proposed Code BERT-based embedding solution for creating software code embeddings. The experimental findings demonstrate that when it comes to downstream vulnerability detection tasks, Code BERT-based embedding works better than other embedding methods. In order to further improve efficiency, we recommended include synthetic vulnerable functions and performing fine tuning using both real-world and synthetic data to aid the model in learning susceptible code patterns in C. While doing so, we investigated which settings would best fit Code BERT. Evaluation results reveal that the updated model outperforms various state-of-the-art identification techniques on our dataset.
Keywords—BERT, Vulnerability, Embedding

I. INTRODUCTION

The vulnerability of software has been a major problem in cybersecurity research [1-3]. These security flaws pose a risk to the internet and computer systems of businesses and governments.

More and more security flaws are being uncovered. In 2010, there were over 4,600 CVEs (Common Vulnerabilities and Exposures) disclosed. By 2021 though, it will have increased to around 153 955. Similar to natural catastrophes, the threat posed by software vulnerabilities [4-6] is growing in frequency, magnitude, and severity, and may have far-reaching repercussions. Millions of computers might be compromised if an exploitable flaw in a critical system is ever used [7].

Software vulnerabilities are and will remain a major issue [8], despite attempts to reduce the likelihood of making mistakes during development. There has been a dramatic shift in vulnerability defence in recent years, from an emphasis on passive detection to an effort to actively forecast whether or not a code snippet includes a vulnerability.

In recent years, there has been a lot of focus on using deep learning to uncover vulnerabilities. These approaches have been applied to the field of networking and communications with encouraging results [9, 10]. As a result, researchers are motivated to enhance the usefulness of deep learning-based vulnerability detection solutions from a variety of perspectives. Neural networks are applied for automated feature extraction in vulnerability detection tasks, which helps to improve the generalization ability of a model capable of latent features and extracting high-level automatically. In order to apply deep learning methods to the task of vulnerability detection, one must first gather data, then prepare that data, then construct a model, and finally test and evaluate the effectiveness of the model. Existing vulnerability detectors using conventional embedding techniques like GloVe, Word2Ve, and FastText generally suffer from low accuracy and recall [11]. . Code's bimodal nature [12] necessitates a model that can account for distant contextual relationships. Nonetheless, recent research often use standard embedding approaches, such as Word2Vec, which can only generate an individual embedding for a specific code token and not embeddings based on diverse contexts. Due to this, we think of employing Code BERT as a feature generator and code embedding.

Code BERT may generate several embeddings according on the situation. Code BERT, on the other hand, is predicated on a bidirectional transformer that may record dependencies between code sequences that are physically separated. Information loss is reduced, relationships between contexts are protected, and hidden patterns of susceptible code are identified. Code BERT, on the other hand, is built on the same principles as multihead attention, allowing it to zero in on various landmarks inside a code sequence. The value of a variable in a code fragment with a loop is dynamic and depends on the loop condition; yet, the vector associated with this variable in the noncontextual embedding techniques remains static. The produced vector representation does not reflect the value change that occurred to the variable, which is indicated by the.is symbol. One possible explanation for the relatively low precision achieved by the work [11] Word2Vec is used because noncontextual embedding models can only generate a single representation with one word, and hence cannot provide alternative representations based on the varied coding contexts.

The acronyms used throughout this study are shown in Table 1, and we address this omission by using a Code BERT-based embedding solution for vulnerable function identification. Using natural language approaches like Word2Vec, GloVe, and FastText, recent studies have produced outstanding results in embedding source code. Code BERT's foundation in the bidirectional transformer makes it a useful tool for gaining a broad grasp of code semantics. Additionally, Code BERT utilises 12 parallel attention heads, a method inherited from multihead attention, which enables the model to simultaneously attend to susceptible features from distinct representation subspaces at different places.

.

Code BERT has the potential to generate more meaningful code embeddings than noncontextual embedding approaches like Word2vec, GloVe, and FastText because of this. In a wide range of code processing/analysis tasks, including defection detection and clone detection, and natural language code search, Code BERT showed encouraging results. But it hasn't been applied to finding C language security flaws. in conclusion

There are three main takeaways from this paper.

I. We provide a comprehensive analysis of many popular code embedding systems, including GloVe,Word2vec, FastText, and Code BERT. We find that in terms of vulnerability identification in C open-source projects, the greatest results may be achieved by employing Code BERT as a code embedding solution.

II. To overcome the data imbalance issues commonly encountered in vulnerability detection, we employ synthetic vulnerability data collected from Software Assurance Reference Dataset (SARD) to fine tune the settings of Code BERT. Using simulated vulnerability data has been shown to optimise vulnerability detection outcomes and further enhance Code BERT's utility in experiments.

III. Important Code BERT settings for code feature extraction are analysed, and the ideal parameters for vulnerability detection are rated for performance.

This paper's remaining sections are structured as follows. Some previous research on vulnerability identification using deep learning is presented in Section 2. In Section 3, we discuss the study's conceptual framework, the learning of a code representation, the fine-tuning procedure, and the effect of changing the sequence length on the algorithm's performance. Our findings and interpretation of the experiments are presented in Section 4. As a wrap-up to this work, we explain some of the caveats of the suggested approach and some questions that still need to be answered in the future.

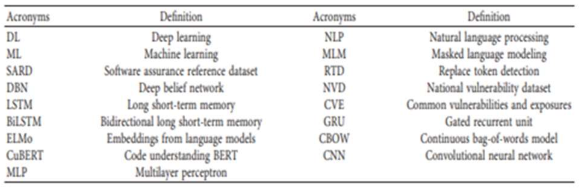
II. CONNECTED TASKS

Numerous methods have been developed for finding security flaws in software. There are multiple overviews that provide systematic assessments of several techniques in this subject from different vantage points [5, 28-32].

Meanwhile, we provide a broad overview of recent studies that have used deep learning to find flaws in a range of neural networks (see Table 2). The convolutional neural network (CNN) [13–16], the deep belief network (DBN) [17], the multilayer perceptron (MLP) [18, 19], the long short-term memory (LSTM) [20–22], and the gated recurrent unit (GRU) [11] are just a few of the numerous deep learning architectures. In addition, a sizable body of research has concentrated on the use of various embedding methods for creating vector representations to be used in the training procedure. Table 3 provides a synopsis of the studies that were evaluated. Word2Vec was used by Pradel and Sen [23] to produce code vectors from domain-specific Abstract Synthetic Trees (ASTs). These vectors were used for the training of deep learning models to find JavaScript vulnerabilities. Word2Vec was used to generate vector representations of C/ C++ code [13].

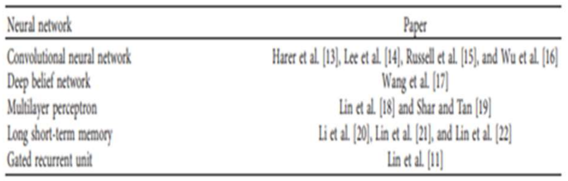
The GloVe model was used by Henkel et al. [24] to generate vectors from C programme Abstracted Symbolic Traces as an alternative to the Word2Vec tool. In addition, FastText was used in FastEmbed [25] to anticipate vulnerabilities using a collection of machine learning models.

TABLE I. ACRONYMS USED IN THE DOCUMENT ARE LISTED BELOW.



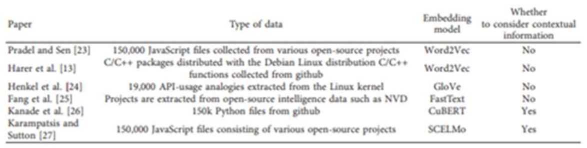
Acronyms	Definition	Acronyms	Definition
DL	Deep learning	NLP	Natural language processing
ML	Machine learning	MLM	Masked language modeling
SARD	Software assurance reference dataset	RTD	Replace token detection
DBN	Deep belief network	NVD	National vulnerability dataset
LSTM	Long short-term memory	CVE	Common vulnerabilities and exposures
BiLSTM	Bidirectional long short-term memory	GRU	Gated recurrent unit
ELMo	Embeddings from language models	CBOW	Continuous bag-of-words model
CuBERT	Code understanding BERT	CNN	Convolutional neural network
MLP	Multilayer perceptron		

TABLE II. ANALYZED RESEARCH PAPERS THAT UTILISED VARIOUS NEURAL NETWORKS TO IDENTIFY SECURITY FLAWS IN SOFTWARE.



Neural network	Paper
Convolutional neural network	Harer et al. [13], Lee et al. [14], Russell et al. [15], and Wu et al. [16]
Deep belief network	Wang et al. [17]
Multilayer perceptron	Lin et al. [18] and Shar and Tan [19]
Long short-term memory	Li et al. [20], Lin et al. [21], and Lin et al. [22]
Gated recurrent unit	Lin et al. [11]

TABLE III. ANALYZED RESEARCH PAPERS THAT USED DIFFERENT EMBEDDING APPROACHES IN SOFTWARE DEVELOPMENT.



Paper	Type of data	Embedding model	Whether to consider contextual information
Pradel and Sen [23]	150,000 JavaScript files collected from various open-source projects	Word2Vec	No
Harer et al. [13]	C/C++ packages distributed with the Debian Linux distribution C/C++ functions collected from github	Word2Vec	No
Henkel et al. [24]	19,000 API-usage analogies extracted from the Linux kernel	GloVe	No
Fang et al. [25]	Projects are extracted from open-source intelligence data such as NVD	FastText	No
Kanade et al. [26]	150k Python files from github	CuBERT	Yes
Karampatsis and Sutton [27]	150,000 JavaScript files consisting of various open-source projects	SCELMo	Yes

In order to generate code representations, numerous contextualised embedding models have been used recently. CuBERT (Code Understanding BERT), introduced by Kanade et al. [26], trains a BERT model on software source code to provide contextual embeddings. To create code representations in context, Karampatsis and Sutton [27] devised a methodology called SCELMo. Both papers demonstrate the usefulness of contextualised embedding models for several code analysis applications. For the purpose of vulnerability identification, we employ Code BERT, which is also a pretrained contextualised model on six programming languages.

III. METHODOLOGY

A. Procedures for Conducting Research.

We compile a ground truth dataset consisting of a variety of software projects with varying characteristics, and we choose numerous synthetic vulnerabilities at the function level. Our process is depicted in Figure 1. Specifics comprise three phases, as shown below. We begin by loading the original code and converting it to a JSON format that Code BERT can understand. In traditional models, sequence data and labels are generated by reading and executing the corresponding source code files. After the data has been turned into code embedding vectors, it is sent on to the next step of the process, where it will be partitioned and given to the GRU neural network to be trained. In order to assess three classic embedding strategies, we used a technique for training code vulnerability detectors. The system was developed using the benchmark for open-source APIs suggested in [11]. Second, we use the synthetic data to train Code BERT, and then we construct the test set's vulnerability probabilities based on this trained model. . In a strange twist, we assessed the effect of adjusting parameters such input sequence length, epoch, batch size, and learning rate depending on fine tuning. Finally, the optimal input sequence length is established for vulnerability feature extraction from C programmes. Tasks geared at addressing the following three research questions are carried out to ensure the suggested approach is effective in delivering the intended outcome. The outcomes will be used to answer each inquiry.

To what extent is Code BERT superior to alternative embedding techniques? This is our first research question (RQ1). . This is a relevant research subject since it has been argued that Code BERT cannot be used to extract features from C programmes. For

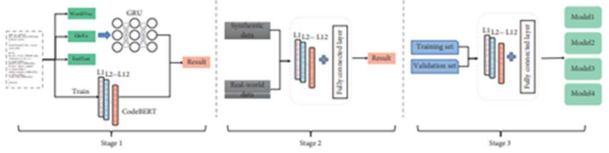


Fig. 1. There are three steps in our workflow. The first step involves a comparison of four models' outputs. Stage 2 involves blending a synthetic dataset created using SARD projects with the actual dataset in order to fine-tune Code BERT's settings. In the third phase, we analyse critical Code BERT settings for extracting features from code.

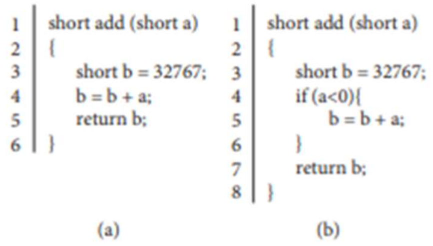
In order to address this question, we will evaluate Code BERT in comparison to other methods such as Word2vec, GloVe, and FastText. Our second research question (RQ2) is: "How can we enhance Code BERT's vulnerability detection performance?"To address this, we do tests comparing and contrasting Code BERT with and without the use of a real-world dataset.Input sequence length and its effect on Code BERT is the topic of our third research question (RQ3). We examine the tweaked model's efficiency on the identical classification tasks using sequences of varying lengths.

B. Understanding Code Representations

Embedding models like Word2Vec, Glove, Fast Text, and Code BERT are used to transform text tokens into meaningful vectors; they have certain commonalities but also have some key distinctions. Using the following criteria, we compare four distinct models. Code BERT, in comparison to the other three models, has a more complicated structure from a modeling standpoint. Skip-gram and CBOW are the two models found in Word2Vec. Both CBOW and skip-gram use context words to predict middle words; skip-gram uses middle words to anticipate neighboring words. Each model consists of an input layer, a mapping layer, and an output layer. . Since the hidden layer is linear in its structure, it can be trained quickly. Word2Vec creates a separate vector for every word in the corpus, but it doesn't take into account how words like "apple" and "apples" are related to one another. The two terms have identical internal morphology, making them synonymous. Yet Word2Vec processes both words, therefore this internal form is not taken into account. FastText employs n-grams based on individual characters to prevent this issue. Similar to the FastText model, which likewise consists of an input layer,and an output layer, a hidden layer, the network structure of the FastText model is rather straightforward. Matrix factorization is used by GloVe, and the training time is quite quick. The produced word vector, however, lacks in semantic information and is hence only useful for specific applications like similarity computation. Code BERT's model structure is more involved than that of the three models just discussed. Code BERT uses a multilayer two-way transformer as its foundation. Code BERT is structured as follows: 12 layers, 12 self-attention heads per layer, 64-by-64-pixel heads, and a 768-by-768-pixel hidden dimension. Vulnerability assessment will be impacted by these distinctions in model structure and computer system.

Given that a target word may represent many contents, a noncontextual embedding model would be too limiting, but Code BERT can produce several vectors for a word based on context information. GloVe,Word2Vec, and FastText cannot.

Knowing the context is essential when assessing any kind of vulnerability. Figure 2 depicts an inspiring real-world example. Since a and b are both short types in the susceptible code (shown in Figure 2(a)), the result of b b + a might cause b to go outside of its range. To prevent this mistake, the range of a was constrained in the corrected function (shown in Figure 2(b)) to be smaller than 0. The values of an are outside of the same range in the two functions. Variables might have several meanings depending on their surroundings, and this work uses Code BERT to extract high-level code representations for spotting susceptible C routines. The accurate comprehension of code semantics will be impacted if conventional models like GloVe, Word2Vec,and FastText are used to transform a word into a fixed vector [33]. Code BERT is a pretrained model for natural language and programming languages that uses Masked Language Modeling (MLM) and token replacement in its training (RTD). In addition, RTD makes extensive use of unimodal data, such as code bases without corresponding natural language data. We use the second goal, vulnerability discovery, to our advantage. In [34], the idea of a transformer-based embedding model called Code BERT is introduced. We also make use of the encoder's multihead attention layer and feedforward network encoder. The utilisation of residual connections across two layers is intended to enhance memory retention. While Code BERT has not been trained in C language syntax and semantics, it is able to effectively apply the knowledge it has gained in other programming languages through the process of transfer learning.



```
short add (short a)
{
    short b = 32767;
    b = b + a;
    return b;
}
```

(a)

```
short add (short a)
{
    short b = 32767;
    if (a<0){
        b = b + a;
    }
    return b;
}
```

(b)

Fig. 2. using C functions as an illustration. An exposed operation, a. This new function is (b). Vulnerable patterns often include declarations, assignments,other operational logic control flow, and examining the contiguous information is frequently inadequate to build semantically meaningful vector representations [36]. Therefore, it is essential to focus on a variety of pivotal locations inside the weak functions. As a result of the algorithm's ability to split its attention across numerous targets, multihead attention makes it easier to identify and exploit weak spots in the code. Furthermore, the vulnerability pattern is time-dependent, which means that long-distance contextual information is essential for identifying sensitive functions. Words like Xin and Xi+n of the term Xi might be helpful for susceptible functions. Positional embedding was developed to help find out how a word is related to its neighbours. To facilitate the addition of the two parameters, the input embedding layer is supplemented by a positional encoding layer, the dimension of which is identical to that of the embeddings (dmodel).
Because the source code structure follows the principles of logic or formal argument and is concerned with meaning in language or logic, it is interconnected and tightly tied.

As a result, the presence of a potentially exploitable code snippet is typically attributable to its constituent parts' links or connections to other portions of the code, either in the immediate or distant past, or both. The lines of code in a susceptible code snippet are typically rather long and can span an entire function [37]. It requires a lot of work or expertise with standard embedding models like Word2Vec [38], but the transformer's design makes it easy to verify the distance between each word. Therefore, Code BERT's architecture may facilitate the model's detection of a long-term reliance of both backward and forwardby providing services that can effectively capture the susceptible programming patterns. While Code BERT has not been specifically trained in C, it is able to transfer its knowledge of relevant patterns from other programming languages to a related detection job through a process called transfer learning.
In order to extract features from the source code, we investigated four different embedding models. Word2Vec: 100-dimensional feature vector, 5-word maximum distance between current and anticipated words in the same phrase.
A word is disregarded if it appears less than five times.
The maximum distance between the current and predicted word inside a phrase is set to 5 in GloVe, and the dimensionality of the output word vectors is set to 100. Additionally, 40 iterations are performed on the corpus, and the learning rate for training is set to 0.001. In FastText, a 100-by-100-by-5-point gap between a present and forecast word inside a phrase is used as the default. Don't count words that occur less than five times in 20 epochs. Code BERT: default number of epochs is 5, default sequence length is 400, default learning rate is 2e 5, default amount of data samples recorded in a single training session is 4, and we utilised the Adam optimizer throughout.
the layer into which this model receives the open-source code
Source code embedding: reducing the dimensionality of the vector used to represent the code.
In order to learn a high-level representation of features, we use a series of 12 encoder layers.
Layer with all connections made: used exclusively for tuning purposes.
In other words, this is the layer that is responsible for disseminating the high-quality feature description for vulnerability detection that has been learnt.

C. Precision Tuning using Artificial Data

In this research, we present a fine-tuned method that helps Code BERT understand the syntax and structure of C programmes while also capturing their meanings. The C programming language is not one of the six languages that Code BERT has been pretrained in. To utilise Code BERT, you need to be familiar with the C language's code pattern and follow a fine-tuning technique, but you don't need to train it on C language data.
Code BERT needs a lot of training data with labels in order to get it just right. It takes a lot of effort to manually identify the weak points in the system's most important functions. In order to fine-tune the Code BERT, it is difficult to gather a large amount of real data. The following are some of the justifications for including a synthetic artificial dataset: To begin, there is enough data and variety in the simulated set. The syntax and coding conventions used in its simplest forms are included. In addition, it has reliable labels, which are better for model improvement and training. Specifically, we are employing the SARD dataset, where synthetic vulnerable functions are poured off the training set and validation set, respectively, to create a total of 10,000 functions, 10% of which are susceptible.

The problem of data disparity is therefore also addressed. In practise, data are rarely perfectly balanced for multi-classification tasks like financial fraud detection or fault isolation. The majority class's characteristics are emphasised during the model's training, while the minority class's characteristics are ignored, thanks to the data distribution's limitations. The model's ability to classify data becomes less accurate as a result of this. The capability of the model to extract vulnerability traits can be bolstered by the inclusion of artificially created vulnerability data. The real-world dataset is split into validation set, a training set, and test set, with 1189, 395, and 399 susceptible functions, respectively, in accordance with the mixed dataset settings [39]. To ensure that susceptible functions constitute exactly 10% of the total functions in the synthetic dataset, 7486 vulnerable functions are hand-picked for the training set and 2495 vulnerable functions are poured into the validation set. The test data set did not contain any susceptible synthetic functions.

D. Examine the Effects of Penalties

Code BERT Fine-Tuned for a Range of Sequence Lengths. Initial investigations show that employing varied sequence lengths has a profound effect on detection outcomes, and this is likely due to the fact that source code functions have varying durations when transformed to sequences. In order to strike a good balance between excessively lengthy vectors and information loss, it is necessary to truncate the overly long codes while converting codes to vectors of the predetermined length. If a function's code sequence is too short, it is extended by 1 s. The efficiency of the Code BERT model is very sensitive to the length of the input.e sequence. A large number of functions will be truncated when the input sequence is too short, leading to data loss. .e model is unable to properly comprehend the nuances of functions. What it picks up could be as minimal as the variables' declarations and definitions. When the input sequence is too lengthy, many functions will end up being filled with 1, and most of the space will be taken up by meaningless data. Some potentially exploitable functions may be located in other phrases than the one where the focus is currently being held, hence there are also long-distance dependencies inside the programmes. Code BERT's bidirectional design means that the model can lose track of its past observations if the separation is too great. Therefore, the length of the input sequence is crucial, since it allows the model to zero in on the specific susceptibility trait.

Table 4 depicts the lengths of functions from the real-world dataset, which has a total of 132,018 functions. We found that around 45.4% of samples are within 128 elements in length, and that 67.7% of all functions have a length of fewer than 256.

A 128-element sequence may not be optimal, despite the fact that this range contains the largest percentage of all possible inputs. There's a tradeoff to be made between the relative importance of shorter and longer sequences of function codes.

Specifically, we optimised Code BERT with sequence lengths of 128, 256, 384, and 512. Initially, we fine-tuned the.e model, and then we used the Adam optimizer with a learning rate of 2e 5. The batch size is 4, which takes into account the GPU's memory constraints. One of the numerous hyperparameters we modified to reach the ideal values was the batch size, which plays a role in the model's generalizability. Given that Code BERT has already been pretrained on six distinct programming languages, but still need its weights to be initialised, some adjustment is required before the model can be used for the specific job at hand.

IV. ANALYSIS AND TESTING

A. Dataset

We theorised that security flaws are typically mirrored in the structure of the source code, especially at the function level. Therefore, the weaknesses we discuss in this study are those at the level of individual functions. Table 5 displays the statistics for the aforementioned data sets.

1) Authentic Dataset

The assessment dataset is comprised on 12 widely used open-source software projects and libraries, including Asterisk,ImageMagick, Httpd,LibPNG, OpenSSL,LibTIFF, Pidgin, qemu, samba, VLC Player, and Xen. vulnerabilities at both the file and function level, the.is dataset originally developed by Lin et al. [11] has been expanded to provide a dual-granularity vulnerability detection dataset. TheCommon Vulnerabilities and Exposures (CVE) and National Vulnerability Dataset were used to guide the human labelling of susceptible files and routines (CVE). We use the dataset, which contains 1,983 susceptible functions and 130,035 nonvulnerable ones, to conduct our tests. Classifiers may learn real-world susceptible patterns with the help of the real-world vulnerability dataset.

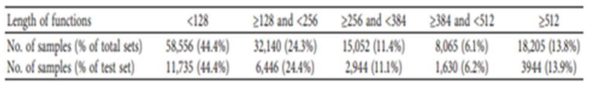
2) Made-Up Dataset

The Software Assurance Reference Dataset (SARD) project provided the source data for the function samples used in the synthetic vulnerability dataset. Source code patterns that are known to be susceptible have been used to artificially build code fragments for the samples. Meanwhile, each test case relies on a single major function to ensure the code is compilable. Classifiers may be trained on the synthetic dataset to recognise the susceptible, but simplified, patterns.

B. Conditions of the Experiment

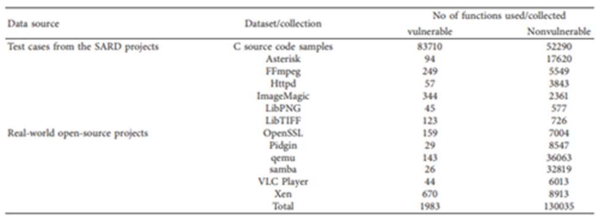
Python's Keras and TensorFlow's Word2Vec, GloVe, and FastText are used to realise Code BERT's backend, while the language's dictionaries are used to realise Word2Vec, GloVe, and FastText (1.14.0). Our testing platform is a PC outfitted with a 4.00 GHz Intel Core i7-6700k CPU and a 4.00 GHz NVIDIA GeForce GTX 1070 GPU. examination of four distinct designs: Code features extracted by four embedding models (Word2vec, FastText, GloVe, and Code BERT) are compared and contrasted on a real-world dataset to see which one performs best. We found 1,983 vulnerable functions out of a total of 132,018 across 12 open-source projects. Selected samples of source code functions are split 6:2:2 across the training set, the validation set, and the test set. Step five, the fine-tuning phase, saw a comparison experiment performed to prove the viability of the suggested approach that

TABLE IV. LENGTH DISTRIBUTIONS OF EXPERIMENTAL DATASETS AND TEST SETS IN THE ACTUAL WORLD. BASED ON THEIR LENGTH, E FUNCTIONS ARE SEPARATED INTO FIVE DISTINCT CLASSES.



Length of functions	<128	≥128 and <256	≥256 and <384	≥384 and <512	≥512
No. of samples (% of total sets)	58,556 (44.4%)	32,140 (24.3%)	15,052 (11.4%)	8,065 (6.1%)	18,205 (13.8%)
No. of samples (% of test set)	11,735 (44.4%)	6,446 (24.4%)	2,944 (11.1%)	1,630 (6.2%)	3944 (13.9%)

TABLE V. SARD



Data source	Dataset/collection	No of functions used/collected	
		vulnerable	Nonvulnerable
Test cases from the SARD projects	C source code samples	83710	52290
	Asterisk	94	17620
	FFmpeg	249	5549
	Httpd	57	3843
	ImageMagic	344	2361
	LibPNG	45	577
	LibTIFF	123	726
Real-world open-source projects	OpenSSL	159	7004
	Pidgin	29	8547
	qemu	143	36063
	samba	26	32819
	VLC Player	44	6013
	Xen	670	8913
	Total	1983	130035

aids in the development of Code BERT by providing feedback. Code BERT was trained and validated on Table 6's functions, and then input the test set into the model to generate a CSV file ordering the functions by their vulnerability likelihood. We utilised the same three sets from the fine-tuning phase as the evaluation dataset to determine the best values for the other parameters.

C. Parameters for Assessing Performance.

Precision and recall are often used measures of performance for categorization tasks.
However, there are many more nonvulnerable functions than vulnerable ones; the ratio is around 96:1. This extreme data imbalance might cause the classifier to prioritise the more numerous class when training, leading to inaccurate results. Top-k percentage precision (P@K%) and top-k percentage recall (R@K%) is extensively used in the study of information retrieval systems of the top-k retrieved documents, and we employ these metrics to accurately monitor the performance of the suggested approaches.
The test set data (P@K%) is the susceptible data that was successfully detected by the vulnerability scanner. It is possible to compute R@K% using the two mathematical formulas below, where K% is the fraction of the test set containing vulnerable data.

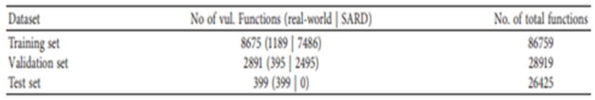
$$P@K\% = \frac{TP@K\%}{TP@K\% + FP@K\%},$$

$$R@K\% = \frac{TP@K\%}{TP@K\% + FN@K\%}.$$

D. Results and Analyses

A number of research projects have been undertaken to solve the problem of vulnerability detection; examples include the method presented in [11] and an open-source detector called Flawfinder [40]. Since we have complete access to the source code and the dataset, we utilise these systems as the benchmarks, and Flawfinder is a well-known open-source programme that is extensively used in practise. There is an order to the evaluation, which is achieved by answering three specific research questions.
Since choosing an appropriate embedding approach can have a significant impact on the efficacy of vulnerability detectors (RQ1), we evaluate Code BERT against more conventional embedding models to ascertain which is the best option for this purpose. In this article, we provide the findings of an experiment in which four embedding models were used to identify potentially exploitable C programming language functions. Figure 3(a) provides a detailed comparison of the accuracy achieved by four embedding models when extracting the top 1% of most likely susceptible functions: 46%, 28%, 41%, and 61%, respectively. As may be seen in Figure 3(c),
TABLE VI. PARAMETERS OF CODE BERT, IT IS IMPORTANT TO KNOW THE TOTAL NUMBER OF SUSCEPTIBLE AND NONVULNERABLE FUNCTIONS. IN ORDER TO ENSURE SUSCEPTIBLE FUNCTIONS ACCOUNT FOR JUST A TENTH OF

THE TOTAL NUMBER OF FUNCTIONS, WE SUPPLEMENTED THE ORIGINAL DATASET WITH SYNTHETIC DATA FOR USE IN THE TRAINING SET AND VERIFICATION SET.



Dataset	No of vul. Functions (real-world | SARD)	No. of total functions
Training set	8675 (1189 | 7486)	86759
Validation set	2891 (395 | 2495)	28919
Test set	399 (399 | 0)	26425

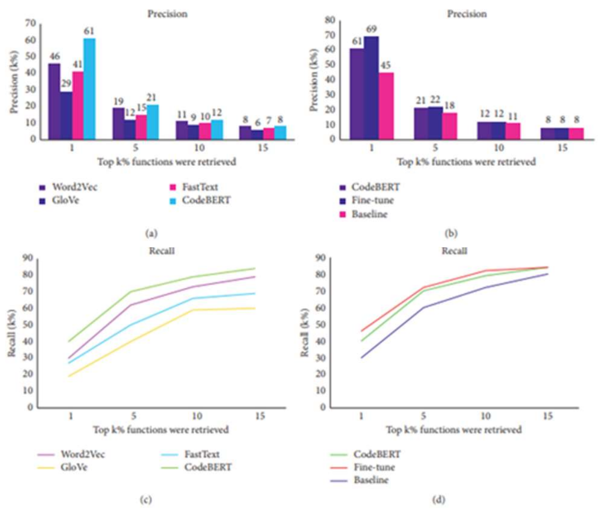


Fig. 3. The findings of two independent tests. Precision and recall for several embedding approaches (a) and three models (b) and (d).

Code Bert’s green line (recall) is higher than the other three lines (recall from GloVe,Word2Vec,and Fast Text). Code BERT was able to find 40% of all susceptible functions by fetching only 1% of functions. Only 30% of the overall susceptible functions were discovered using Word2Vec.

Fast Text and Glove models could only identify 27% and 20% of true susceptible functions, respectively. In comparison to the other three approaches, Code BERT's code embeddings make it easier to spot security flaws.

To round things up, we thought about how hard it would be to use Code BERT computationally. The encoder structure of the transformer and the multihead attention mechanism are both wrapped in the deep learning framework, making it difficult to directly assess the computational complexity of Code BERT (e.g., PyTorch). . Thus, we decided to evaluate the effectiveness of Code BERT and other embedding techniques by contrasting their training and testing durations.

Code BERT's computational complexity may be measured against other similar programmes. Table 7 describes Code BERT, the other three models, and their training and test times, each as measured in epochs.

We found that during the training phase, Code BERT required 6720 seconds to finish one epoch. In comparison, all three embedding models took about the same amount of time to train (about 300 seconds).

.

When comparing training times, the noncontextual models (Word2Vec, GloVe, and FastText) perform better than Code BERT. This is because the structure of the Code BERT model is more complicated.

the second research question is to see if fine-tuning Code BERT will increase its performance. At the output's base, we add a layer of complete connectivity. Meanwhile, we use a simulated dataset of software vulnerabilities to further adjust the settings.

Synthetic dataset used in the Assurance Reference Dataset (SARD) project to model vulnerable code patterns via testing. Figure 3(b) and (d) indicate that after fine tuning, the model is 8% more accurate when obtaining 1% of susceptible functions, and it is able to locate 84% of all vulnerable functions while getting 15% of possibly vulnerable functions. Also, the fine-tuned model performs better than other models, including the vulnerability detector [11]. The increases in accuracy (24%) and recall (16%) are rather large when only 1% of all functions are retrieved. Using the fine-tuned method was shown to increase detector performance.

The goal of this experiment is to learn how much of an effect the optimal sequence length has on RQ3. We design tests to prove that certain sequence lengths are optimal for detecting vulnerabilities. The outcomes of models with varying sequence lengths are displayed in Tables 8 and 9. Out of a total of 26,425 functions in the test set, 399 are known to have security flaws. We conclude that a sequence length of 256 is optimal for finding C-language software flaws. The model is optimized for sequences of length 256, which is true whether the goal is to get the top 1%, 5%, 10%, or 15%. The sequence length of 128 in the model only incurs a 26% accuracy loss when analysing the top 1% of all functions.

We compared the aforementioned findings to a baseline for consistency's sake. The findings of the suggested technique, as shown by Flawfinder, rank functions based on the likelihood that they are susceptible. As a result, we used the aforementioned measures of effectiveness to characterise both detectors. We report on a comparison of their detection abilities in the actual world.

Vulnerabilities: The comparative findings may be seen in Tables 8 and 9, where the model with a sequence length of 256 significantly beats the FlawFinder. When retrieving the top 1% of functions from the test set of 399 susceptible functions, Flawfinder only achieved 3% accuracy and 2% recall, whereas the same model with a sequence length of 256 incurs a P@1% of 70% and R@1% of 47%.

The Flawfinder returns subpar findings, as well, retrieving just 5%, 10%, and 15% of routines classified by vulnerability severity, respectively.

.

This scenario demonstrates one again how efficient the Code BERT-based embedding approach is for finding security flaws.

V. DISCUSSION

The compromise that must be made between ease of use and precision in modelling is discussed here. It is impossible to directly analyse the computational cost of Code BERT since the encoder structure of the transformers and the multihead ann model are both enveloped in the deep learning framework.As an illustration of the temporal cost of training, we use an era. Code BERT's training time was 6720 seconds, Word2Vec's was 287 seconds, GloVe's was 285 seconds, and FastText's was 286 seconds, thus it's evident that Code BERT requires the most time. The fundamental reason is that Code BERT may potentially store a vast body of information that the other models have not captured, such as certain hypothetical code patterns and semantic aspects, because to its complicated structure and high capacity. Also, the parameter is complicated and requires extensive computation, which adds extra time.

Still, Code BERT is preferable to other options since it enhances accuracy and recall in vulnerability discovery.

Tables 8 and 9 display statistical data illustrating how different sequence lengths fared while using Code BERT to extract code characteristics. Finding the right compromise between function length and input sequence length is another difficulty of the current work. We investigate the many factors that may account for the observed performance variations across sequence sizes. The 256-item sequence length fared best on the real-world dataset compared to all other lengths.

The susceptible functions have data like a head file, variable declaration, logic code,parameters, and a return value, which is why they are exploitable. The features acquired may be limited to things like header files and variable declarations if the sequence length is too short. As vulnerabilities are typically encoded in obfuscated logic, the model may end up learning aspects that aren't relevant to the problem at hand. When the sequence length is more than 256, performance suffers; this may be due to the fact that the majority of susceptible functions have lengths less than 256; if the sequence length is set too high, the sequence will be automatically filled with 1. Models' capacity to judge the vulnerability's aspects will be hampered if too much extraneous data is included in the analysis, while at the same time attention will be diverted to the extraneous data.

In most cases, lowering the sequence length to 256 will significantly cut down on wasteful repetition and data loss.

TABLE VII. TRAINING AND TESTING DIFFICULTIES WHEN EVALUATING FOUR MODELS

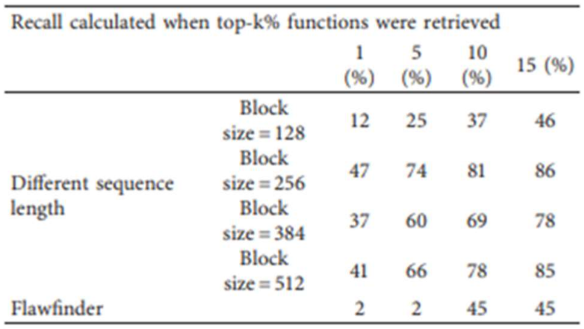
Models	Training time per epoch (s)	Test time per epoch (s)	Number of epochs	Total training time with all epoch completed (s)
CodeBERT	6720	600	5	33600
Word2Vec	287	32	150	43050
GloVe	285	32	150	42750
FastText	286	32	150	42900

		1 (%)	5 (%)	10%	15 (%)
Different sequence lengths	Block size = 128	26	7	6	5
	Block size = 256	70	22	12	8
	Block size = 384	56	18	10	9
	Block size = 512	63	20	12	8
Flawfinder		3	3	7	7

TABLE VIII. COMPARE FLAWFINDER'S ACCURACY ON THE IDENTICAL CLASSIFICATION TASKS WITH THAT OF SEQUENCES OF VARYING LENGTHS.

Precision calculated when top-k% functions were retrieved					
		1 (%)	5 (%)	10%	15 (%)
Different sequence lengths	Block size = 128	26	7	6	5
	Block size = 256	70	22	12	8
	Block size = 384	56	18	10	9
	Block size = 512	63	20	12	8
Flawfinder		3	3	7	7

TABLE IX. COMPARE FLAWFINDER'S RECALL ON THE IDENTICAL CATEGORIZATION TASKS WITH THAT OF SEQUENCES OF VARYING LENGTHS.



Recall calculated when top-k% functions were retrieved					
		1 (%)	5 (%)	10 (%)	15 (%)
Different sequence length	Block size = 128	12	25	37	46
	Block size = 256	47	74	81	86
	Block size = 384	37	60	69	78
	Block size = 512	41	66	78	85
Flawfinder		2	2	45	45

VI. CONCLUSIONS

In this research, we offer an embedding technique for vulnerability detection that is inspired by Code BERT. Code BERT has not been educated on the syntax and semantics of the C programming language, but it has been taught other languages and has the ability to quickly pick up the most useful characteristics of C. To fine-tune Code BERT, we have utilised both C open-source projects and custom-built C functions. In the meantime, we have built a practical real-world dataset for gauging the efficacy and quality of the solution and future iterations of deep learning-based vulnerability detectors. . The experimental findings demonstrate that the suggested embedding solution may return K (1, 5, 10, 15)% of all functions with accuracy that exceeds expectations, suggesting that the method can aid in the discovery of vulnerabilities in C open-source projects.

The following restrictions of the current system should be removed for it to be further enhanced. Code BERT, which has 12 encoder layers and almost 110 million parameters, is both time-consuming and costly to train and install.

.

Consequently, it may be helpful to provide a simplified model.
Second, current embedding approach for vulnerability detection can only handle software code in C# and C++. More study is needed to allow for integration with other computer languages, and a big real-world dataset with numerous detection granularities is currently lacking. In order to hasten the data collecting process, further study might be put into creating an automated vulnerability data labelling system.

REFERENCES

[1] M. Wang, T. Zhu, T. Zhang, J. Zhang, S. Yu, and W. Zhou, “Security and privacy in 6g networks: new areas and new challenges,” Digital Communications and Networks, vol. 6, no. 3, pp. 281–291, 2020.
[2] Y. Miao, C. Chen, L. Pan, Q.-L. Han, J. Zhang, and Y. Xiang, “Machine learning-based cyber attacks targeting on controlled information,” ACM Computing Surveys, vol. 54, no. 7, pp. 1–36, 2022.
[3] X. Chen, C. Li, D. Wang et al., “Android hiv: a study of repackaging malware for evading machine-learning detection,” IEEE Transactions on Information Forensics and Security, vol. 15, pp. 987–1001, 2019.
[4] G. Lin, W. Xiao, L. Y. Zhang, S. Gao, Y. Tai, and J. Zhang, “Deep neural-based

vulnerability discovery demystified: data, Table 7: .e complexity of training and test when we compare four models. Models Training time per epoch (s) Test time per epoch (s) Number of epochs Total training time with all epoch completed (s) Code BERT 6720 600 5 33600 Word2Vec 287 32 150 43050 GloVe 285 32 150 42750 FastText 286 32 150 42900 Table 8: Test precision of various sequence lengths against Flawfinder on the same classification tasks. Precision calculated when top-k% functions were retrieved 1 (%) 5 (%) 10% 15 (%) Different sequence lengths Block size � 128 26 7 6 5 Block size � 256 70 22 12 8 Block size � 384 56 18 10 9 Block size � 512 63 20 12 8 Flawfinder 3 3 7 7 Table 9: Test recall of various sequence lengths against Flawfinder on the same classification tasks. Recall calculated when top-k% functions were retrieved 1 (%) 5 (%) 10 (%) 15 (%) Different sequence length Block size � 128 12 25 37 46 Block size � 256 47 74 81 86 Block size � 384 37 60 69 78 Block size � 512 41 66 78 85 Flawfinder 2 2 45 45 10 Security and Communication Networks model and performance," Neural Computing and Applications, pp. 1–14, Springer, New York, NY, USA, 2021.

[5] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: a survey," IEEE Communications Surveys & Tutorials, vol. 20, no. 2, pp. 1397–1417, 2018.

[6] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: a comparison of software vulnerability discovery processes," in Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), pp. 374–391, IEEE, San Francisco, CA, USA, May 2018.

[7] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," IEEE/ACM Transactions on Networking, vol. 23, no. 4, pp. 1257–1270, 2014.

[8] S. Liu, G. Lin, L. Qu et al., "Cd-vuld: CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation," IEEE Transactions on Dependable and Secure Computing, 2020.

[9] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescape, "Mobile encrypted traffic classification using deep learning: experimental evaluation, lessons learned, and challenges," IEEE Transactions on Network and Service Management, vol. 16, no. 2, pp. 445–458, 2019.

[10] T. O'shea and J. Hoydis, "An introduction to deep learning for the physical layer," IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563–575, 2017G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. (references)