IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 February 2024, accepted 28 April 2024, date of publication 2 May 2024, date of current version 13 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3396410

RESEARCH ARTICLE

DB-CBIL: A DistilBert-Based Transformer Hybrid Model Using CNN and BiLSTM for Software Vulnerability Detection

AHMED BAHAA[1,2], AYA EL-RAHMAN KAMAL[1], HANAN FAHMY[1], AND AMR S. GHONEIM[3]
[1]Department of Information Systems, Faculty of Computers and Artificial Intelligence, Helwan University, Helwan 11795, Egypt
[2]Department of Information Systems, Faculty of Computers and Artificial Intelligence, Beni-Suef University, Beni Suef 62521, Egypt
[3]Department of Computer Science, Faculty of Computers and Artificial Intelligence, Helwan University, Helwan 11795, Egypt

Corresponding author: Aya El-Rahman Kamal (Aya.Rahman@fci.helwan.edu.eg)

ABSTRACT Software vulnerabilities are among the significant causes of security breaches. Vulnerabilities can severely compromise software security if exploited by malicious attacks and may result in catastrophic losses. Hence, Automatic vulnerability detection methods promise to mitigate attack risks and safeguard software security. This paper introduces a novel model for automatic vulnerability detection of source code vulnerabilities dubbed DB-CBIL using a hybrid deep learning model based on Distilled Bidirectional Encoder Representations from Transformers (DistilBERT). The proposed model considers contextualized word embeddings using the language model for the syntax and semantics of source code functions based on the Abstract Syntax Tree (AST) representation. The model includes two main phases. First, using a vulnerable code dataset, the pre-trained DistilBert transformer is fine-tuned for word embedding. Second, a hybrid deep learning model detects which code functions are vulnerable. The hybrid model is built on two Deep Neural Networks (DNN). The first model is the Convolutional Neural Network (CNN), which is used for extracting features. The second model is Bidirectional-LSTM (BiLSTM), which has been used to maintain the sequential order of the data as it can handle lengthy token sequences. The utilized source code dataset is derived from the Software Assurance Reference Database (SARD) benchmark dataset. Final experimental findings show that the proposed model outperforms the state-of-the-art approaches' performance by improving precision, recall, F1-score, and False Negative Rate (FNR) by 2.41%-8.95%, 4.0%-16.28%, 1.85%-12.74%, and 18% respectively. The proposed model reports the lowest FNR in the literature, a significant achievement given the cost-based nature of vulnerability detectors.

INDEX TERMS Automatic vulnerability detection, BERT, deep learning, DistilBERT, transformers.

I. INTRODUCTION

Software vulnerabilities are a particular category of defects that make software less secure and make it possible for products to be used maliciously. Insecure coding is the primary cause of the significant number of security flaws that occur nowadays. Software vulnerabilities continue to be a major issue despite the effort put into developing secure programming. This can be explained by the fact that in 2016, there were roughly 6,500 vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database, which increased to approximately 25,200 vulnerabilities in 2022. As of the end of 2023, the number of published vulnerabilities had reached approximately 28,961 [1].

Since security vulnerabilities are always present in the source code and can have severe effects, it is necessary to identify them as early as possible. Automated tools can help software engineers explore software code, identify which part of the code is likely to contain vulnerabilities at different levels of granularity (package, file, class, or function), and raise the alarm about them. Such tools and prediction models can help prioritize effort and optimize the cost of inspection and testing. They aim to reduce the time it takes software engineers to find vulnerabilities and increase the likelihood

The associate editor coordinating the review of this manuscript and approving it for publication was Tai-Hoon Kim.

that software engineers will find and fix them early in the software life cycle [2].

Software vulnerability detection can guarantee software security early in the development process and eliminate the difficulty of manually reviewing complex software systems for potential vulnerabilities, which is intrinsic to the organization. There are two methods for detecting vulnerabilities. The first is to use traditional vulnerability detection techniques. The other involves deep learning (DL) or machine learning (ML) methods.

Most traditional techniques employ vulnerability detection methods based on patterns manually identified by security practitioners, such as static analysis tools. This process is challenging, exhausting and time-consuming. Currently, a large variety of static analysis tools are available (such as [3], [4], [5], and [6]) for source code analysis by automatically checking the source code against defined coding rules from standards or custom pre-defined rules to ensure it is compatible, safe, and secure.

On the other hand, techniques based on ML and DL have been explored [7], [8]. To identify vulnerability patterns, these ML/DL-based techniques [9], [10], [11], [12], [13] can dynamically learn the vulnerability patterns without manually specifying predetermined vulnerability patterns. ML (as opposed to DL techniques) requires features or patterns – defined by experts – to automate the detection of vulnerability patterns from code entries.

The DL-based techniques usually start with a data pre-processing module that includes code representation and tokenization. To use deep learning for code analysis, software code (source code) must be converted to vector representations that are understandable by DL algorithms (word embedding). Hence, follows is a word embedding procedure resulting in a vector representation that serves as an input to a DL model. The DL model extracts the relevant features and consequently classifies the source code entries as either vulnerable or clean.

Most previous studies for software vulnerability detection relied on static word embedding methods to generate vector representations to distinguish between vulnerable and benign source entities (see Related Work subsection). When used for vulnerability detection, static word embeddings – *such as Word2Vec [14], GloVe [15], and FastText [16]*– are only able to train word embeddings based on a small context window and are unable to capture meaning that depends on both the target words and their context as a whole [17]. Since software vulnerability detection is a process that requires context-sensitive code analysis to comprehend better the data flow, control flow, and relationships between data [11], [18], [19], it's crucial to capture the context of source code tokens appropriately. Therefore, in recent studies, researchers have begun to use language models based on natural language processing (NLP) for contextualized word embedding, such as ELMo [20], BERT [21], and GPT-2 [22]. Still, not all the state-of-the-art (SOTA) language models have been employed for software vulnerability detection – for instance, DistilBERT.

The majority of the literature that classifies code entries as vulnerable or clean is capable of detecting only two categories of vulnerabilities (for instance, CWE-399—*Resource Management Errors*, and CWE-119—*Failure to Constrain Operations within the Bounds of a Memory Buffer*, as in [11]). Incorporating more categories of vulnerabilities entails DL models capable of capturing varying patterns; this can be addressed by designing hybrid DL models with components capable of learning different patterns. Hybrid models for software vulnerability detection were not by far investigated within the literature, except in a handful of studies.

Despite the enormous progress of the emerging DL-based vulnerability detection, there are still certain limitations in detection performance regarding False Negative Rate (FNR).

Therefore, we leveraged deep-learning methods to build our vulnerability detection model based on a SOTA language model for contextualized word embedding (the DistilBERT word embedding). Furthermore, to the best of our knowledge, there has been no such approach that leverages a hybrid deep learning model of BiLSTM with CNN to detect software vulnerabilities.

In this paper, we propose DB-CBIL, a functions-level vulnerability detection approach, for vulnerabilities of varying categories. The approach is based on the Abstract Syntax Tree (AST) code representation, utilizing DistilBert Transformer (attention) technology for word embedding and using a hybrid DL model that integrates CNN and BiLSTM. The following are the paper's main contributions:

- We start by generating an AST for source code representation, then applying BERT tokenizer for code tokenization.
- For word embedding, we leverage the language model to capture deep contextualized word representations and create several embeddings according to different word contexts by fine-tuning the DistilBert Transformer.
- To capture context-aware code semantics and be capable of learning the long-term dependencies, reflecting potentially vulnerable function patterns in the source code, we designed a hybrid deep neural network based on CNN and BiLSTM.
- Experimental results show that DB-CBIL outperforms the SOTA methods [11], [23], [24], and [25] on the Software Assurance Reference Database (SARD) [26] dataset in terms of precision, recall, and F1-score, with 0.0% FNR which is the lowest amongst the rates reported in the literature.

The paper is organized as follows: Section II presents the relevant background and the most related works. Section III presents the research questions and the proposed model overview, DB-CBIL. Section IV describes the experimental setup, the experiments that were conducted, and the metrics used for evaluation. Section V discusses the experimental

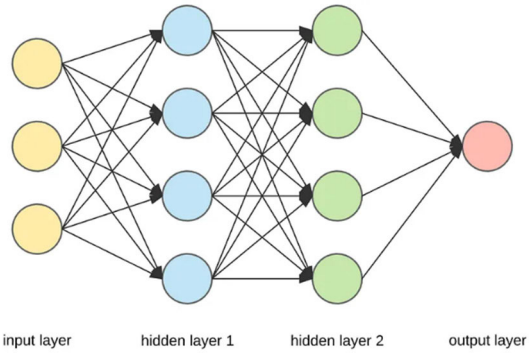


FIGURE 1. DNN architecture.

results. Section VI discusses the limitations of DB-CBIL. Section VII provides the conclusion and future work.

II. BACKGROUND AND RELATED WORK

This section briefly presents background and prior studies related to vulnerability detection. We begin by defining the research context and introducing all relevant concepts, such as DNN, CNN, RNN, AST, transformers and word embedding. We then conclude by referring to the most relevant research studies.

A. DEEP NEURAL NETWORKS (DNN)

DL is a subset of ML. Most DL models now depend on artificial neural networks (ANN) [27]. An ANN is a complicated neural network consisting of artificial neurons (ANs) inspired by neurons in the human brain. The AN gets inputs from several other ANs, processes them, and then generates an output that is transferred to other neurons. Each AN processes data in a simple way, but the general behavior of the network as a whole - which results from the interaction of its ANs allows complex problems to be solved. The ANN consists of three types of layers:

1) Input layer: The initial data for the ANN.
2) Hidden layers: The intermediate layer between the input and output layers is where all the computation takes place.
3) Output layer: Generate the result for the given input.

A DNN, as shown in Figure 1, is an ANN with more than one layer between the input and output layers. DNNs implement deep architectures in ANNs. "Deep" means more complex functionality regarding the number of layers and units in each layer.

DL has been effectively used in a variety of disciplines, including software security [28], [29], [30], [31] as well as traditional fields like object detection [32], NLP [33], image classification [34], and recommendation systems [35]. Comparing DL to pattern-based approaches in detecting software vulnerabilities reveals significant benefits:

1) DL techniques eliminate a lot of overhead because they don't require experts to define the features or patterns and can achieve automatic extraction of vulnerability patterns.
2) It is challenging to identify a precise pattern encompassing several vulnerabilities, even those of the same type (because vulnerabilities of the same type exist in different forms).
3) DL techniques have been demonstrated to simultaneously detect multiple types of vulnerabilities, in contrast to pattern-based techniques, which often can only detect one specific type of vulnerability [11].

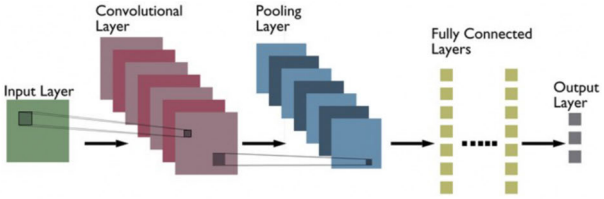


FIGURE 2. CNN architecture.

B. CONVOLUTIONAL NEURAL NETWORK (CNN)

A convolutional neural network (CNN) is a specific type of ANN that is employed in different applications such as text classification, image recognition, and NLP. CNN can be classified as one-dimensional (1-D) CNN for NLP tasks and two-dimensional (2-D) CNN for image recognition tasks [36]. Figure 2 shows the architecture of CNN. A deep CNN consists of three layers: a convolutional layer, a pooling layer and a fully connected (FC) layer. The convolutional layer is the core building block of a CNN. Most computations take place in the convolutional layer. It indicates how many convolutional filters (or kernels) there are. It is used on the input data to create feature maps. CNN uses the max pooling layer to reduce the dimensionality of the output by reducing the number of input parameters. The FC layer and sigmoid represent the output layer in the end, which produces the classification results based on the features extracted in the preceding layers [28].

C. RECURRENT NEURAL NETWORK (RNN)

Recurrent Neural Network (RNN) [37] is another type of ANN that can be used to process sequential data (Figure 3). In contrast to feedforward ANNs, which only allow information to flow from input to output in one direction, RNNs enable the connection between neurons in the same hidden layer to allow information to flow in feedback loops or cycles, enabling them to capture temporal dependencies. These loops help RNNs process sequences of input data [38]. Recurrent connections are the key feature of RNNs that enable the network to preserve information about the past and use it to leverage the current prediction. This is called the internal memory state of the RNN and is often referred to as the "hidden state". It serves as a summary or memory of previous inputs that the network has encountered. RNNs are particularly suitable for time series or natural language

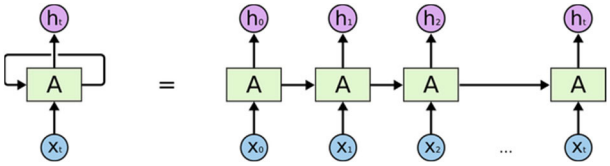


FIGURE 3. RNN architecture.

tasks [39]. RNNs have a known problem called the Vanishing Gradient (VG) problem [40], which causes RNNs to lose long-term dependencies that can lead to ineffective model training. RNN can alleviate the VG problem by using a technique known as long short-term memory (LSTM) [41]. The LSTM network is a sequential ANN that preserves information because it is primarily designed to avoid long-term dependency problems.

A Bidirectional LSTM, also referred to as a BiLSTM, is an RNN used mainly in NLP. It is a sequence processing model consisting of two LSTMs, one to process the input in the forward direction and the other to process it in the backward direction. Due to BiLSTM's ability to process data in both directions, the model can better understand the relationship between sequences and obtain context information that can lead to more insightful outputs.

D. ABSTRACT SYNTAX TREE (AST)

Abstract Syntax Tree (AST) is typically amongst the first intermediate representations generated by code parsers of compilers and thus forms the basis for the generation of many other code representations. This tree accurately represents the nesting of statements and expressions that build programs. ASTs are structured trees with both inner and leaf nodes. leaf nodes that represent operands (such as variables or constants) and inner nodes that are used for operators (such as additions or assignments). AST is used to capture various syntax and semantic information of source code functions. Examine Figure 4b, which depicts an AST for the code sample shown in Figure 4a, as an illustration.

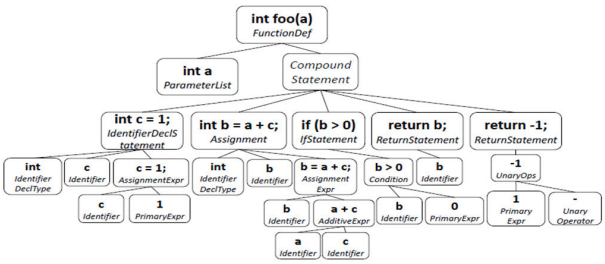
E. WORD EMBEDDING

The static word embedding (non-contextualized embedding) methods [42] typically treat each word as an independent unit, disregarding the relationships between words, which makes it challenging for these algorithms to learn expressive and rich context-related semantics.

For example, Word2Vec [14] can only produce one embedding for a single word. It cannot generate various embeddings for a word with diverse meanings dependent on its contexts. The inability to produce diverse embeddings based on multiple contexts may lead to inaccurate or insufficient representations of code contexts, which may ultimately impair the effectiveness of subsequent code analysis activities. Due to the drawbacks of static word embeddings, recent deep neural language models such as ELMo [20], BERT [21], and GPT-2 [22] are optimized to develop context-sensitive word representations [20] for a variety of NLP tasks. Effective vector representation guarantees that the semantics and syntax of the software code are maintained, better supports the learning process, and ultimately benefits code analysis tasks.

```
int foo(int a){
    int c = 1;
    int b = a + c;
    if (b > 0) {
        return b;
    }
    return -1;
}
```

(a)



(b)

FIGURE 4. a. Function code example. b. AST representation.

Figure 5 illustrates two examples of vulnerable (a) and benign (b) code examples to show the vulnerable context: a local variable called testStr, defined at line 8, has a string longer than the buffer's size, which is defined at line 7. No size check is performed on the variable testStr when assigned to the buffer in the sprintf function at line 9, causing a buffer overflow vulnerability. A detection method should be able to comprehend the semantics of code tokens and identify contextual dependencies, ranging from the declaration or assignment of a local variable to the calling of the sprintf function. The if statement determines whether the variable testStr is large enough before calling the sprintf (code sample (b) at line 9).

The vulnerability would be patched by this modified code segment (b) since it prevents the sprintf from being called if the testStr variable's size surpasses the array size. Therefore, the presence or absence of validation statements in the preceding code environment determines if the sprintf function is vulnerable. Therefore, sprintf should be declared differently depending on the situation (with or without parameter validation) and being able to recognize these variants is the key to identifying vulnerable patterns.

F. TRANSFORMERS

Transformer [43] is a DL model and is a type of ANN architecture that uses an attention mechanism to process sequential data, such as sentences of text and time-series data. Compared with traditional RNN and CNN, the transformer differs in its ability to capture contextual information and long-range dependencies. It has an encoder-decoder architecture. There are exactly as many decoders as there are encoders, which

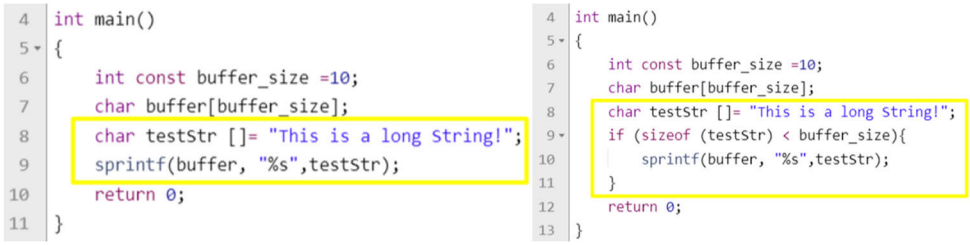


FIGURE 5. The different contexts of the sprintf token (a) The vulnerable context of the sprintf token, (b) The patched context of the sprintf token.

can be thought of as a stack of layers. Self-attention, positional embeddings, and multi-head attention are the three key features that make the transformers extremely powerful. The self-attention mechanism enables the transformer to identify the relationship between pairs of input tokens (e.g., if the input content is text, then the tokens are words), even if they are far from each other. It can then evaluate how important those relationships are, which enhances context understanding. The position of each token in a sequence is encoded by positional embeddings, which incorporate this positional information into the word embeddings. By adding positional embeddings into the input representation, the model will be able to capture the order and position of words in a sequence. The transformer uses multi-head attention, which enables a greater ability to encode nuances of word meanings by repeating computations in parallel within the attention module. The similar attention computations are combined to generate the attention score.

1) BERT

BERT (Bidirectional Encoder Representations from Transformers) [21] is a variant of the original transformer architecture. It consists of multilayer bidirectional transformer encoders with 12 layers of transformer blocks. It is a pre-trained model on a very large dataset and has the capability to understand the context of the input sentence. The model can be fine-tuned on the specific tasks datasets to achieve good outcomes. It can discover contextual relationships between words (or sub-words) in a text. BERT could be applied to a wide range of NLP tasks, by adding an additional layer to the fundamental model. It attaches special tokens to the sequence, such as the classifier token [CLS] and the separator token [SEP]. [CLS] token is employed in sequence classification (i.e., classifying the entire sequence in contrast to individual tokens).

It is the first token of the sequence when built using special tokens. [SEP] token is used when creating a sequence out of multiple sequences (i.e., sequence classification). It is used as the last token of a sequence.

2) DistilBERT

DistilBERT transformer (Lightweight Distilled Bidirectional Encoder Representations from Transformers) [44] employs a smaller BERT-base network with six layers of transformer blocks, and it is trained using the proposed Knowledge Distillation technique [45]. DistilBERT is 60% faster and 40% smaller than BERT-base.

G. RELATED WORK

Static vulnerability detection based on source code can be classified into two methods: code similarity-based and pattern-based. Detectors based on code similarity can only detect vulnerabilities that occur due to code clones [46], [47].

Pattern-based techniques are also subdivided into rule-based and ML-based methods. Rule-based approaches employ vulnerability patterns, which are manually created by human specialists, to detect vulnerabilities (e.g., Checkmarx [3], Flawfinder [4], Rats [5], and cppcheck [6]). These tools often result in high false positive and false negative rates. False positive (in this context) refers to benign entries classified as vulnerable, while false negative refers to vulnerable instances classified as non-vulnerable.

ML-based techniques [48] can be divided into three subcategories:

1) Vulnerability detection methods based on traditional software metrics, such as code complexity, coupling, and cohesion [49], [50], [51], [52].
2) Anomaly detection methods: These methods identify vulnerabilities through abnormal patterns (e.g., missing checks [53], API usage [54]), but they cannot handle regular patterns that are rarely used.
3) Vulnerable code pattern identification techniques: These techniques extract vulnerability patterns related to code representation (e.g., from AST [55], code property graphs [56]), or system calls [57], and identify vulnerabilities using these patterns.

Recent years have seen researchers start looking into vulnerability detection utilizing ML and DL methods (Figure 6). Harer et al. [58] employ ML techniques to detect vulnerabilities using two approaches. The first makes use of features extracted from programs' intermediate representations (IR), which are created during the build and compilation phases — the second works directly with source code. Yamaguchi et al. [59] built an AST for each function, and then applied latent semantic analysis using the bag-of-words model. This approach uses the matrix singular value decomposition (SVD) technique, which captures the structured pattern in the syntax tree.

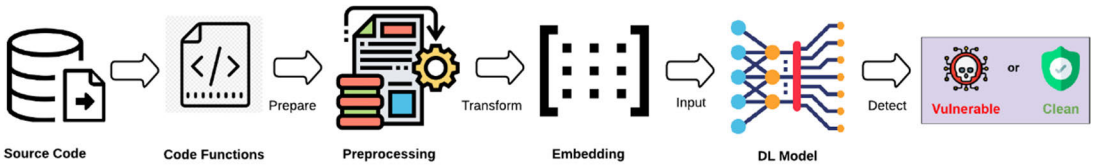


FIGURE 6. A generic model of software security vulnerability detection using a DL approach.

One of the initial investigations on the use of DL for vulnerability detection was conducted by Zagane et al. [60], it used code metrics as features for software vulnerability detection based on DL. The research revealed that while code metrics are good, they are not the best attributes to employ in DL-based vulnerability detection as gathering such software metrics requires manual labor and takes time.

In order to automatically learn vulnerability patterns from historical data, numerous DL-based techniques have been proposed [9], [12], [13], [61]. Russell et al. [12] proposed a DL-based detection system that works on the function-granularity level. In order to extract the relevant meaning of crucial tokens from the unprocessed source code of each function, they created a unique lexer representation. To extract useful features, they used an ensemble classifier (namely, a random forest) of multiple neural representation (based on ''CNNs'' and ''RNNs''). Lin et al. [62] proposed POSTER, which is a framework for detecting function-level vulnerabilities across different projects. They represented functions using the AST. They used Bidirectional Long Short-Term Memory (BiLSTM) for capturing the features of a function. Another BiLSTM-based approach is presented by Li et al. [63], the method identifies vulnerable and secure functions in source code based on the extraction of semantic features from the function names. They worked on CVE entries submitted between 2008 and 2018. The method has a lower false-positive rate and can identify different types of vulnerabilities. Farid et al. [28] proposed CBIL, a hybrid model that combines CNN and Bi-LSTM to improve software testing and review of code by identifying defective areas in source code based on Word2Vec model.

VulDeePecker [11] presents a DL-based vulnerability detection system that can automatically extract and identify features using ''code gadgets,'' which are semantically connected code lines that are not necessarily sequential. These code gadgets are transformed into symbolic representations, and then variable-length vector representations of code gadgets are transformed into fixed-length vectors using Word2Vec embedding. VulDeePecker 's design mainly addresses vulnerabilities related to calls to library/API functions. It used the BiLSTM model. The method does not accommodate control dependency; only data dependency is supported. The dataset used only contains two types of vulnerabilities: buffer error and resource management error vulnerabilities.

SySeVR [10], provides a program representation that can include information about the vulnerability's syntax and semantics by focusing on semantic information induced by data and control dependency. To generate vectors SySeVR also used Word2Vec embedding. However, SySeVR addressed the shortcomings of VulDeePecker outlined above, but the algorithms used for generating syntax and semantics need to be improved to accommodate more semantic information for vulnerability detection.

A transformer-based model for detecting vulnerabilities in the software program is proposed by Hou et al. [23]. The approach employs a Multi-Layer Perceptron (MLP) model and is based on the AST for source code representation to extract both data and control dependencies between the sentences. This approach is superior to the traditional ML-based vulnerability detection methods in that it uses the structure of a transformer and the attention mechanism to learn and detect system vulnerabilities.

Another transformer-based model for detecting vulnerabilities in source code is VulBERTa [24], which is based on RoBERTa model [64]. Firstly, VulBERTa pre-trains the RoBERTa model based on Masked Language Modelling (MLM) [21] to create its code representation. Then, the pre-trained VulBERTa model is used with the other two models, MLP and CNN, to implement two different classification methods (VulBERTa-MLP and VulBERTa-CNN) for fine-tuning vulnerability detection models. This approach is evaluated on binary and multi-class vulnerability detection tasks using different datasets (such as VulDeePecker, Draper, and REVEAL).

VulDeBERT [25], is another DL-based vulnerability detection system for C and C++ source code based on BERT model. They created their own code gadget generation method to detect the vulnerabilities related to system function calls. Using the generated code gadgets, the pre-trained BERT model is fine-tuned. They mainly worked on two vulnerabilities (CWE-119 and CWE-399).

To the best of our knowledge, no research has been done on applying a hybrid deep learning model to extract features directly from source code in a big codebase based on contextual word embedding in order to identify different types of vulnerabilities while taking into consideration code syntax and semantics through the use of transformers.

III. METHODOLOGY

The research questions, proposed model, and several steps involved in conducting the study are presented in this section.

A. RESEARCH QUESTIONS

To evaluate the effectiveness of the proposed model, we need to answer the following three research questions (RQs) by conducting experiments:

RQ1: How accurate is DB-CBIL for detecting software vulnerabilities at the function level?

RQ2: Does DB-CBIL achieve better vulnerability detection performance than state-of-the-art (SOTA) methods?

RQ3: Can hybrid deep learning based on a language model be leveraged to detect multiple types of vulnerabilities?

RQ4: Can the proposed model achieve an improved Recall and False-Negative Rate compared to the SOTA methods?

B. DB-CBIL MODEL OVERVIEW

The DB-CBIL model will be presented in this section. Figure 7 depicts the entire model architecture. Our code is made available at [65].

C. DATA PREPARATION

Before building the model, the dataset needs to be prepared to produce feature representations that help the DNN models comprehend the semantics and patterns of code functions. Data preparation comprises two steps: parsing the source code into an AST, followed by tokenization using BERT.

1) PARSING SOURCE CODE INTO AST

Using the source code entities for training/testing the proposed model involves two primary data pre-processing steps; namely, the generation on an AST representation of the code, followed by eliminating the irrelevant nodes from the generated AST.

The AST is generated using the ANTLR ''*CodeSensor*'' parser [66], [67]. ANTLR generates an AST per function, for each function available in the utilized dataset. The AST are serialized; that is, the tree structure is converted into a linear format (such as XML or JSON). This process usually involves traversing the AST in a depth-first manner and encoding each node's type, value, and relationships to other nodes in a format similar to that shown in Figure 8.

Then, each of the resulting raw serialized-AST files is processed further by selecting the important AST nodes as tokens (therefore, eliminating any unnecessary nodes from further consideration to avoid impairing the accuracy of the model). Listed in Table 1 are the node-types to be selected as tokens further consideration:

- Control-flow nodes *(e.g., "if", "while", and "do")*,
- Function-call nodes *(usually represented using solely the function-name, without any parenthesis or arguments)*, and finally,
- Declaration nodes (e.g., for declaring methods, parameters-lists, and variables), which are listed alongside their datatypes/return-types.

As an example, the representation of a function's AST nodes mapped to tokens can be:

- [*return_type, void, function_name, parameter_list, statements, decl, int,...*].

TABLE 1. The selected nodes of the serialized ASTs.

Control Flow Nodes	▪ If Statement ▪ While Statement ▪ Do Statement ▪ For Statement ▪ Assert Statement ▪ Break Statement ▪ Continue Statement ▪ Return Statement ▪ Throw Statement ▪ Synchronized Statement ▪ Try Statement ▪ Switch Statement ▪ Block Statement ▪ Switch Statement Case ▪ For Control ▪ Enhanced For Control ▪ Try Resource ▪ Catch Claus ▪ Catch Clause Parameter
Declarations Nodes	▪ Formal Parameter ▪ Method Declaration ▪ Constructor Declaration ▪ Variable Declaration
Method Invocations Nodes	▪ Method invocation ▪ Super Method Invocation

In which the first and second nodes ('*return_type*' and '*void*) indicate the return type of the function, and the subsequent nodes ('*parameter_list*', '*decl*', and '*int*') indicate that the first parameter for the function in an integer, .. etc.

2) TOKENS ENCODING (BERT TOKENIZATION)

Tokenization involves transforming the text tokens into integer tokens. That is, for each function, the selected text-tokens from a serialized AST are transformed into a sequence of integer values (namely, Token IDs). This step is crucial for the subsequent generation of an embedding vector (per AST), which is the input to the hybrid DL model.

For efficient tokenization, the pre-trained BERT's tokenizer (which involves a WordPiece model [55]) is fine-tuned using the selected dataset. The fine-tuning process aims to further train a pre-trained model on a more specific downstream task using relatively smaller datasets, which in this study is the software vulnerability detection task, using the SARD dataset.

The tokenizer encodes the input text (i.e., the selected AST tokens) into a sequence of integers known as ''Input tokens'', by representing each token with a unique integer termed a ''Token ID''. In addition, to indicate the start and end for a sequence of tokens representing a single function, for each function, the [CLS] token is added at the beginning of its sequence of tokens, and [SEP] is added at the end of each sequence. Finally, padding is utilized to make the sequences for all functions of the same length. Since the body of the functions in the dataset varies in length, padding ensures they all conform to a length of 512.

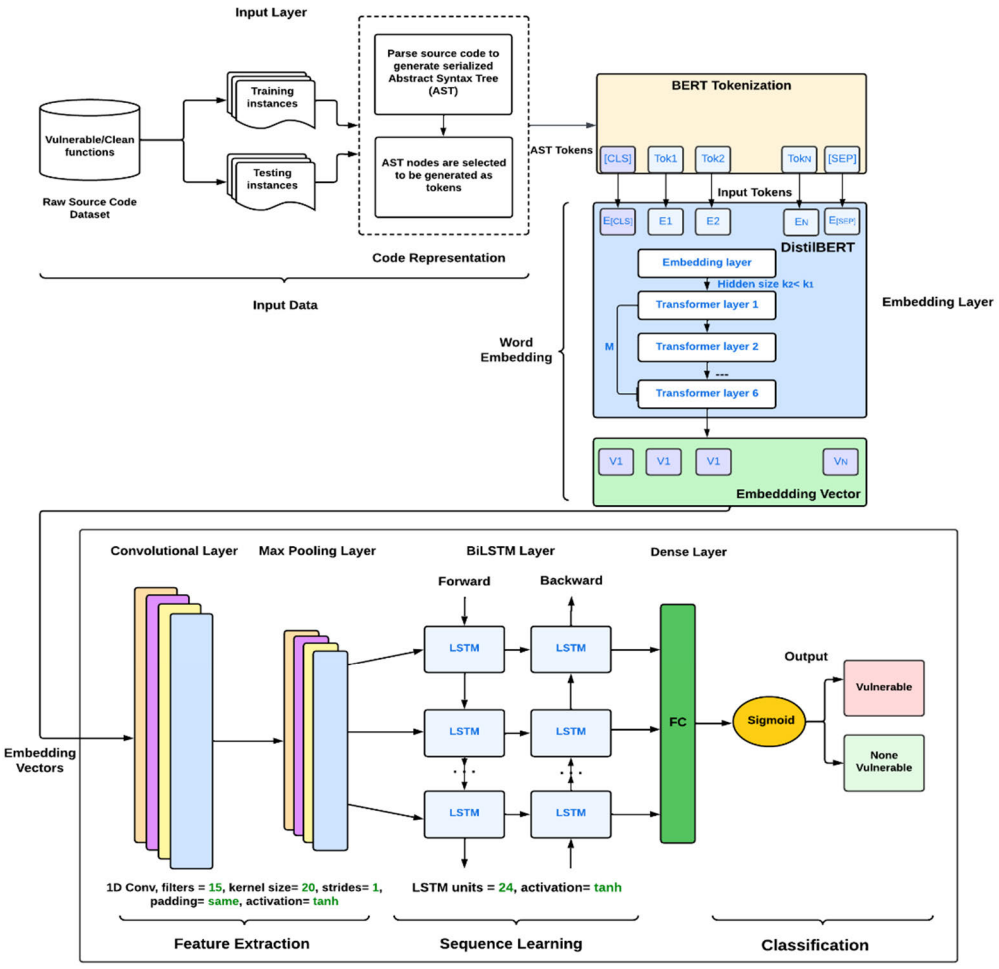


FIGURE 7. The proposed DB-CBIL model architecture.

```
SOURCE_FILE     1:0     1:0     0
FUNCTION_DEF    1:0     87:0    1
RETURN_TYPE     1:0     1:0     2       void
TYPE_NAME       1:0     1:0     3       void
LEAF_NODE       1:0     1:0     4       void
FUNCTION_NAME   1:5     1:5     2       CWE121_Stack_Based_Buffer_Overflow__CWE129_connect_socket_08_bad
LEAF_NODE       1:5     1:5     3       CWE121_Stack_Based_Buffer_Overflow__CWE129_connect_socket_08_bad
PARAMETER_LIST  1:69    1:70    2       ( )
LEAF_NODE       1:69    1:69    3       (
LEAF_NODE       1:70    1:70    3       )
LEAF_NODE       2:0     2:0     2       {
STATEMENTS      3:4     65:4    2
SIMPLE_DECL     3:4     3:12    3       int data ;
VAR_DECL        3:4     3:8     4       int data
TYPE    3:4     3:4     5       int
TYPE_NAME       3:4     3:4     6       int
LEAF_NODE       3:4     3:4     7       int
NAME    3:8     3:8     5       data
```

FIGURE 8. Example of serialized AST representation.

3) PROPOSED WORD EMBEDDING USING DISTILBERT

The sequences of integer tokens cannot be used as input to DL models; they must be transformed – *or translated* – to word embedding vectors. However, not all word embedding approaches can precisely capture and represent the context of each AST. Therefore, to effectively learn the context for each token, this study proposed and investigated the application of DistilBERT transformer as a contextual word embedding approach. That is, to represent the semantic information in the function sequence, the output of DistilBERT is the contextualized vector representations for each input sequence. The contextual embeddings are produced by the transformer encoder using a self-attention mechanism.

In this study, the pre-trained DistilBERT transformer is fine-tuned using the SARD dataset, and then each sequence of integer tokens is converted into a dense vector using the fine-tuned DistilBERT. Thus, for each function, the DistilBERT model takes the sequence of numeric representation of the selected tokens as an input and generates the corresponding contextual embedding vector. Each integer vector must have a specified length. The vector is masked by 0 if its length is less than the identified length. Additionally, any excess length will be truncated if the vector length is greater than the identified length (i.e., 512).

4) PROPOSED HYBRID MODEL: CNN + BiLSTM

In addition to the DistilBERT Embedding layer, the proposed DB-CBIL model consists of three additional components: the CNN layers, a BiLSTM layer, and a Dense – *fully-connected* – layer.

a: CNN LAYERS

The architecture starts with two layers, a convolutional layer, and a max pooling layer. The convolutional layer is used to learn the features of software vulnerabilities automatically. Since the source code entries are in fact textual data, this study employed a 1-D convolutional layer which is suitable when processing sequential data. Within the convolutional layer, the number of filters *(set to 15 in the experimental setup)* determines how many dimensions there are in the output space, while the filter length *(set to 20)* reveals how long the 1-D convolution window is. To select the most effective settings for the convolutional filters, multiple experiments using the SARD dataset have been conducted.

After performing the convolutional operation by the convolutional layer and extracting the data feature maps, the dimensions of the extracted features are still very high. Therefore, to solve this problem and reduce the cost of training the model, the convolutional layer is followed by a max pooling layer to reduce the dimensionality of the generated feature maps, by retaining only the most important information. The following equations explain their calculations:

$$l_t = tanh(x_t * w_t + b_t) \tag{1}$$

where l_t refers to the output value from the convolutional layer, *tanh* is the activation function, x_t is the input sequence, w_t is the convolution kernel's weight, and b_t is its bias. The maximum pooling is chosen by the pooling layer. After convolution and pooling, the output vector is as shown in the equation:

$$X_t = f_{maxpooling}\{tanh(x_t * w_t + b_t)\} \tag{2}$$

where X_t is the output from the CNN layer and $f_{maxpooling}$ is the operation of the maxpooling. Then the output of the CNN layers is passed to the BiLSTM layer.

b: BiLSTM LAYER

The sequential order of the data (Input sequence) is maintained by the BiLSTM layer. BiLSTM implementation helps in detecting long-term dependencies that can effectively capture key features. The BiLSTM layer receives the max pooling results as input to select the valuable information. To detect both contextual – *forward and backward* – information, it employs two LSTM networks. The LSTM consists of a "memory cell" that can track and store long-term dependency information in memory for long periods. The LSTM unit consists of three gates: an input gate, a forget gate, and an output gate. The LSTM is configured with 24 units. The basic structure of the LSTM unit is shown in Figure 9. It includes the logistic sigmoid function σ, the activation function *tanh*,

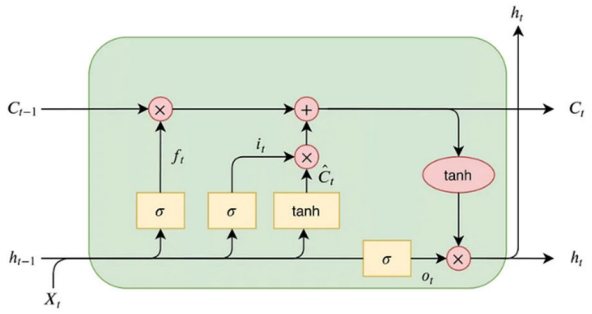


FIGURE 9. The Architecture of the LSTM unit.

the new input vector X_t, the output from the previous timestep h_{t-1}, the output h_t, the old cell state C_{t-1}, and the new cell state C_t.Additionally, f_t, i_t, and o_t, stand for the forget, input, and output gates, respectively. The processes performed in the input, forget, and output gates of the LSTM unit to calculate the output h_t can be expressed mathematically by the equations 3 to 8, where *W*and *U* are the weight matrices of the three gates, *b* is the bias, and $C_t^{\wedge}$ is the candidate value.

As shown in Figure. 9, The information will be added or deleted in the cell state using the three gates. The input gate i_t manages the information flow to the memory cell. The forget gate f_t manages the information flow out of the memory cell. The information flow out of the LSTM and to the output is then managed by the output gate o_t. The input data will be passed through the sigmoid activation function. The two inputs X_t (the input at the current timestep) and h_{t-1} (the output from the previous timestep) are fed to the forget gate and multiplied, respectively, by the weight matrices W_f and U_f,followed by an addition of the bias b_f.

$$f_t = \sigma(W_f X_t + U_f h_{t-1} + b_f) \tag{3}$$

The result is passed through the activation function, which gives a binary output [0 or 1]. If the cell state output is 0, the information will be forgotten, and if the output is 1, the information will be retained for future use.

Useful information is then added to the cell state by the input gate. Using the inputs X_t and h_{t-1}, the input gate processes the information using the sigmoid function, and the values are either retained or not (thus behaving likewise the forget gate).

$$i_t = \sigma(W_i X_t + U_i h_{t-1} + b_i) \tag{4}$$

Next, using the weight matrices W_c and U_c, and the bias value b_c, a vector $C_t^{\wedge}$ is generated using the *tanh* activation function that results in output ranging from -1 to $+1$.

$$C_t^{\wedge} = tanh(W_c X_t + U_c h_{t-1} + b_c) \tag{5}$$

The current cell state C_t is calculated by adding the $C_t^{\wedge}$ to i_t, while the previous cell state C_{t-1} is multiplied by f_t (and thus either included or ignored).

$$C_t = f_t * C_{t-1} + i_t * C_t^{\wedge} \tag{6}$$

The output gate o_t is responsible for extracting the useful information from the current cell state to present it as an output. Initially, a vector is generated by applying the *tanh* activation function to the current cell state C_t. Additionally, the information – i.e., X_t and h_{t-1} – is filtered using a sigmoid function. Finally, the output h_t – which acts as the input to the next cell – is determined by multiplying the retained values o_t by the vector $tanh(C_t)$.

$$o_t = \sigma(W_o X_t + U_o h_{t-1} + b_o) \quad (7)$$

$$h_t = o_t * tanh(C_t) \quad (8)$$

c: DENSE LAYER

Finally, the dense layer – *also known as a fully connected (FC) layer* – is connected to the BiLSTM layer to obtain the final vulnerability detection results. The neurons of the dense layer are connected to every neuron of the BiLSTM layer which means that each neuron receives input from all the neurons of BiLSTM layer. The dense layer in the proposed DB-CBIL model consists of a single unit which reflects the dimensionality of the output space. Thus, the dimensions of the vectors resulting from the BLSTM layer is reduced by the dense layer. We used *sigmoid* (*logistic function*) as an activation function within the dense layer for a binary classification, where the dense layer's low-dimension vectors are fed into the *sigmoid* function responsible for generating the classification result (a value between 0 and 1). The value can then be interpreted as the probability of the sample belonging to one of the classes (vulnerable, or non-vulnerable).

IV. ENVIRONMENT AND EXPERIMENTAL WORK

Several experiments are conducted to evaluate the performance of the proposed DB-CBIL model. The results are then compared to other DL models used to address software vulnerability detection. Details of the dataset used (the Software Assurance Reference Database – *SARD* –) are discussed in the dataset subsection. Experiments are performed on the configuration and setup described in detail in the Experimental Setup subsection. For each experiment, we report multiple evaluation indicators detailed in the evaluation metrics subsection, including all metrics reported in the previous work that utilized the SARD dataset. By doing so, our comparison will be fairer.

A. DATASET

We have selected a benchmark dataset which is frequently used by security researchers to evaluate methods for detecting vulnerabilities in C/C++ source code functions. It is made up of samples from the SARD [26]. A publicly available subset of SARD employed by numerous SOTA studies contains a total of 33,360 functions, including 12,303 functions that are vulnerable and 21,057 functions that are not vulnerable. The dataset covers different categories of vulnerabilities such as Buffer errors, Numeric errors, Resource management errors, and more. Table 2 shows the list of CWEs (Common Weakness Enumeration) that are included in this dataset. The dataset is randomly divided into a training set, a validation set, and a testing set, with the ratios of 6:1:1. The utilized dataset has been made available at [68].

TABLE 2. Vulnerabilities in the dataset.

#	CWE ID	Name	# Samples
1	CWE-78	OS Command Injection	963
2	CWE-121	Stack-based Buffer Overflow	3,560
3	CWE-122	Heap Based Buffer Overflow	2,122
4	CWE-124	Buffer Under-write	1,230
5	CWE-126	Buffer Over-read	682
6	CWE-127	Buffer Under-read	1,015
7	CWE-134	Uncontrolled Format String	2,208
8	CWE-194	Unexpected Sign Extension	24
9	CWE-195	Signed to Unsigned Conversion	468
10	CWE-197	Numeric Truncation Error	1
11	CWE-590	Free Memory Not on Heap	15
12	CWE-690	NULL Deref from Return	15

B. EXPERIMENTAL SETUP

The DNN models were implemented utilizing keras (version 2.10.0) with a TensorFlow backend (version 2.11.0) [69], using Python 3.9.12. The Transformer package employed is version 4.28.1. The server is running an Ubuntu OS 20.04.6 LTS, with a NVIDIA A100 GPU processor, and 32 GB RAM.

For the proposed DB-CBIL model, the hyperparameters are set as follows:

- The BERT tokenizer has a maximum sequence length of 512.
- The CNN layer is composed of 15 filters, each with a length of 20, in addition to the *tanh* activation function. However, varying numbers of filters and filter lengths were investigated, with the results reported in the next section.
- The number of LSTM units in the BiLSTM layer is set to 24 and utilizing the *tanh* activation function. Also, varying numbers of LSTM units were investigated, with the results reported in the next section.
- The loss function of the model is set to *binary_crossentropy*, *Adam*optimizer is employed, and a learning rate of *3e-5*. Once more, varying learning rates were investigated, with the results reported in Table 3.
- The dense layer utilized a *sigmoid* activation function.
- With all the aforementioned settings, the total number of *epochs* is set to 20, with a *batch size* of 20 per epoch.

C. EVALUATION METRICS

The DB-CBIL model's performance is assessed using several evaluation metrics, including the AUC–ROC *(Area Under the Receiver Operating Characteristics Curve)* and the F-measure. These evaluation metrics are widely used to evaluate prior research studies on software vulnerability detection models [7]. Figure 10 presents the confusion matrix which provides the model's predicted outcomes by generating the results of the following indicators: true positives (*TP*), true

TABLE 3. Detailed performance measures of the DB-CBIL model.

Learning rate	Accuracy	Recall	Precision	F1-Score	FNR (%)	FPR (%)	AUC	Specificity
1e-5	99.79%	1.0%	99.48%	99.71%	0.0%	0.5%	99.77%	99.50%
2e-5	99.80%	1.0%	99.50%	99.72%	0.0%	0.4%	99.79%	99.60%
3e-5	**99.81%**	**1.0%**	**99.51%**	**99.75%**	**0.0%**	**0.3%**	**99.84%**	**99.70%**
4e-5	99.81%	1.0%	99.51%	99.75%	0.0%	0.3%	99.84%	99.70%
5e-5	99.81%	1.0%	99.51%	99.75%	0.0%	0.3%	99.84%	99.70%

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

FIGURE 10. The overview of the confusion matrix.

TABLE 4. Time complexity analysis of the proposed DB-CBIL Model.

Learning rate	Training Time (secs)	Testing Time (secs)
1e-5	1079.68	348.87
2e-5	1075.68	344.21
3e-5	1069.68	126.51
4e-5	1070.41	315.34
5e-5	1132.99	260.56

TABLE 5. SOTA studies' performances compared to the DB-CBIL model.

Model	Precision	Recall	F1-score	FNR
VulDeePecker [11]	90.56%	83.72%	87.01%	18.0%
Transformer-MLP [23]	95.04%	88.89%	91.86%	11.10%
VulBERTa-MLP [24]	95.76%	89.89%	93.03%	10.10%
VulBERTa-CNN [24]	95.26%	90.86%	90.86%	9.14%
VulDeBERT (CWE-119) [25]	97.1%	92.2%	94.6%	7.8%
VulDeBERT (CWE-399) [25]	99.9%	96.0%	97.9%	4.0%
DB-CBIL	**99.51%**	**100.00%**	**99.75%**	**0.0%**

negatives(TN), false positives (FP), false negatives (FN). These indicators of the confusion matrix are used to assess the performance of the proposed model's classification as follows:

- *Sensitivity*/*TPRate*/*Recall* refers to the percentage of actual vulnerable functions that the model correctly classified.

$$Sensitivity/Recall = \frac{TP}{TP + FN} \tag{9}$$

- *Specificity*/*TNRate* refers to the percentage of actual non-vulnerable functions correctly identified by the model.

$$Specificity/TNRate = \frac{TN}{TN + FP} \tag{10}$$

- *FPRate* refers to the percentage of actual non-vulnerable functions that are misclassified by the model as vulnerable.

$$FP = \frac{FP}{FP + TN} = 1 - Specificity \tag{11}$$

- *FNRate* refers to the percentage of actual vulnerable functions misclassified by the model as non-vulnerable.

$$FN = \frac{FN}{FN + TP} \tag{12}$$

- *Precision* refers to how often the proposed model is correct when predicting vulnerable functions.

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

- $F - measure(orF1 - Score)$ combines both recall and precision of the model; that is, it measures how effectively the model can identify venerable functions while minimizing the misclassification of vulnerable/non-vulnerable functions.

$$F1 = \frac{2 * (precision * recall)}{(precision + recall)} \tag{14}$$

- The *ROC* is an evaluation metric used for binary classifiers. It is like a report card for a vulnerability detector. It shows how well the detector distinguishes between vulnerable and non-vulnerable functions. Imagine it as a graph where we measure how often the detector correctly identifies vulnerabilities (i.e., T) without mistaking non-vulnerable ones (i.e., FP). The closer the *ROC* curve is to the top-left corner, the better the detector is at its job.
- Thus, the *AUC* in the *ROC* graph is like grading the overall performance of the vulnerability detector. It is used as a summary of the *ROC* curve, and thus it also represents the ability of the detector to distinguish between vulnerable and non-vulnerable functions across different thresholds. A higher *AUC* (i.e., *AUC* values that are close to 1) means the detector is doing a better job overall – it is a measure of the detector's effectiveness across various scenarios.

Note that a vulnerability detector is a cost-based classifier. That is, in vulnerability detection, a misclassified vulnerable

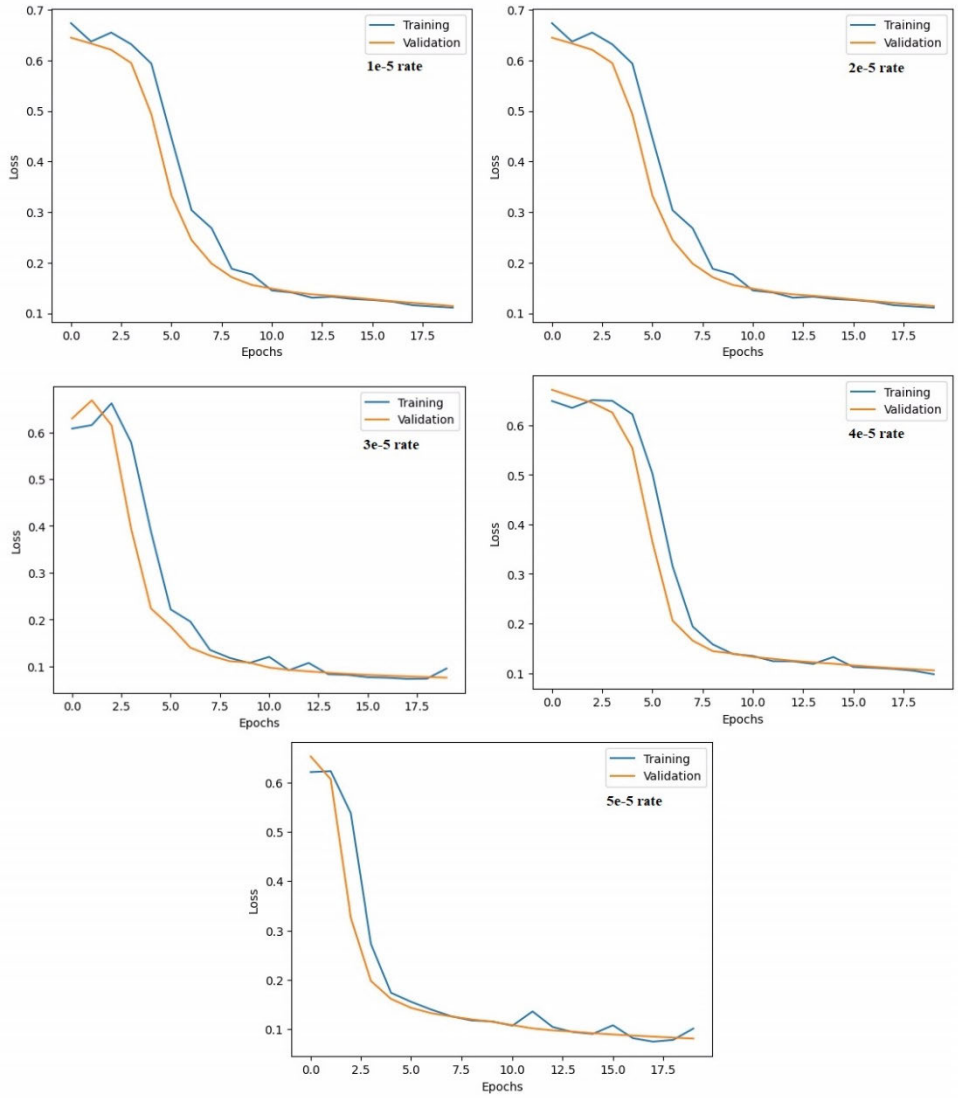


FIGURE 11. Loss versus the epochs for each of the five learning rates.

function (i.e., a false negative—predicting ''non-vulnerable'' for a function that is indeed vulnerable) is often more costly given its direct impacts on security than a misclassified legitimate/clean function (i.e., a false positive—predicting vulnerable for a function that is non-vulnerable), where the decision is merely inconvenient and will result in further unnecessary manual checks. Accordingly, both the False Negative Rate and Recall should be favourably considered when assessing the performance of vulnerability detectors.

V. RESULTS AND DISCUSSION

This section presents the results of the proposed DB-CBI model and discusses the research questions. Experiments were carried out on the PROMISE dataset described in Table 2. The experiments were conducted five times with varying learning rates (1e-5, 2e-5, 3e-5, 4e-5, and 5e-5). The learning rate is a hyperparameter used in optimization algorithms like SGD (Stochastic Gradient Descent) to minimize the loss function that enhances model performance. In our study, we applied the Adam optimization algorithm, an SGD extension. Table 3 summarizes the results of the evaluation of these experiments. The proposed model has a maximum Recall of 100.00%, Accuracy of 99.81%, Precision of 99.51%, AUC of 99.84%, F1-score of 99.75%, Specificity of 99.70% and FNR of 0.0% using learning rates 3e-5 and above. To measure the cost, since the proposed model is cost-based, the FNR is used to evaluate the cost factor. Figure 11 presents a loss per epoch graph (using the binary cross-entropy loss) of the five runs (each using a different learning rate). This is an effective way to visualize the progress while training our hybrid neural network model. In addition, Table 4 shows the time complexity analysis of the DB-CBIL model for both the training and testing phases. It is worth noting that the minimum training and testing times are based on the 3e-5

learning rate. We verified the effectiveness of the proposed model by comparing it to other SOTA vulnerability detection methods, shown in Table 5. The proposed model – DB-CBIL – outperforms the SOTA methods in terms of precision by 2.41%-8.95% and also outperforms those methods in terms of the F1-score by 1.85%-12.74%. But most importantly, given the cost-based nature of vulnerability detectors, the proposed model outperformed the SOTA methods in Recall by even more significant improvements, 4.0%-16.28%.

This improvement results from achieving an FNR of 0.0% (the lowest among the rates reported in the literature).

Hence, this study successfully addressed Section III's four Research Questions (RQs). Addressing RQ1, the DB-CBIL model achieved the top Accuracy of 99.81%, which is more accurate than the other SOTA approaches. To address RQ2, the DB-CBIL model achieves an F1-score of 99.75%, which is better than the other SOTA approaches. Regarding RQ3, the model can effectively detect seven different vulnerabilities, which are: buffer overflow issues [CWE-121, CWE-122, CWE-127, CWE-124], OS command injection [CWE-78], conversion errors [CWE-195], and uncontrolled format strings [CWE-134] by using a hybrid DNN of CNN and BiLSTM layers based on contextual word embedding using DistilBERT model. As for RQ4, the DB-CBIL realized an FNR of 0% and outperformed the SOTA methods in Recall.

VI. LIMITATIONS

The research study poses the following limitations that should be addressed in future work:

- DB-CBIL is applied to detect software vulnerabilities in source code written in C/C++ languages at present, where we used (1) a dataset of C/C++ code entries and (2) a specific parser for these languages. In theory, our approach can also be applied to other programming languages, provided that corresponding datasets and parsers are available in those languages. Theoretically, the proposed approach is also applicable to other programming languages. Thus, one of the future works will be to adapt our method to different languages.
- DB-CBIL can detect many different types of vulnerabilities, such as different buffer overflow issues, OS command injection, conversion errors, and uncontrolled format string issues. We need to conduct experiments on all available types of vulnerabilities.
- The evaluation of DB-CBIL is performed only on the SARD dataset due to the lack of labelled vulnerability datasets with their source code functions, as the existing vulnerability datasets suffer from the wrong labels and coarse-grained level vulnerabilities.

VII. CONCLUSION AND FUTURE WORK

In conclusion, this paper introduces a novel model employing deep learning techniques and contextual word embedding for software vulnerability detection. The newly designed model, DB-CBIL, leverages language models' advanced contextual embedding capabilities using DistilBert transformer to identify semantic features of source code functions to identify software security vulnerabilities. Furthermore, DB-CBIL integrates DistilBert with a hybrid deep-learning model of CNN and BiLSTM. For extracting semantic features from AST tokens, DB-CBIL uses the CNN model. Then, the sequential order of the data can be maintained using the BiLSTM model, and in addition, BiLSTM finds information about long-term dependencies. Experimental results on the benchmark dataset show the effectiveness of the proposed model in improving the performance and accuracy of software vulnerability detection in C/C++ source code functions. DB-CBIL outperforms the state-of-the-art studies for vulnerability detection using the structure of transformers.

In the future, we plan to conduct experiments on additional datasets and expand our method to other programming languages as it is applied only to C/C++ source code functions to make DB-CBIL more generalisable. Since the proposed model works on binary classification (vulnerable or non-vulnerable), it can be enhanced by converting it into a multi-class classification model that classifies the different types/issues of vulnerabilities. In addition, it is promising to adapt DB-CBIL to other software engineering tasks, such as defect detection and code clone detection, due to its ability to handle NLP tasks based on the language models, which we will work on in the future.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions, which helped them significantly improve the manuscript.

The author Aya El-Rahman Kamal would also like to thank his colleague Enas Mohamed Fathy (Author of CBIL), an Assistant Professor with the Faculty of Computers and Artificial Intelligence, Helwan University, who supported her a lot; and grateful to her for providing the necessary resources for this research project.

REFERENCES

[1] *CVE*. Accessed: Dec. 30, 2023. [Online]. Available: http://cve.mitre.org/
[2] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
[3] *Checkmarx*. Accessed: Feb. 10, 2023. [Online]. Available: https://checkmarx.com/
[4] *Flawfinder*. Accessed: Feb. 10, 2023. [Online]. Available: https://dwheeler.com/flawfinder/
[5] *Rats: A Rough Auditing Tool for Security*. Accessed: Feb. 10, 2023. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/
[6] *CPPCheck*. Accessed: Feb. 10, 2023. [Online]. Available: https://cppcheck.sourceforge.io/
[7] A. B. Farid, A. E.-R. Kamal, and A. Ghoneim, "A systematic literature review on software vulnerability detection using machine learning approaches," *FCI-H Inform. Bull.*, vol. 4, pp. 1–9, Jan. 2022.
[8] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2237–2248.
[9] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[10] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul. 2022.

[11] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 18–21.

[12] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[13] Y. Zhou, S. Liu, J. Siow, Y. Liu, and X. Du, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 10197–10207.

[14] T. Mikolov, I. Sutskever, K. Chen, and G. S. Corrado, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 3111–3119.

[15] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[16] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, Dec. 2017.

[17] O. Melamud, J. Goldberger, and I. Dagan, "context2vec: Learning generic context embedding with bidirectional LSTM," in *Proc. 20th SIGNLL Conf. Comput. Natural Lang. Learn.*, 2016, pp. 51–61.

[18] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020.

[19] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3289–3297, Jul. 2018.

[20] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, vol. 1, 2018, pp. 2227–2237.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Language Technol.*, 2019. [Online]. Available: https://aclanthology.org/N19-1423, doi: 10.18653/v1/N19-1423.

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

[23] F. Hou, K. Zhou, L. Li, Y. Tian, J. Li, and J. Li, "A vulnerability detection algorithm based on transformer model," in *Artificial Intelligence and Security*. Cham, Switzerland: Springer, 2022.

[24] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2022, pp. 1–8.

[25] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim, "VulDeBERT: A vulnerability detection system using BERT," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2022, pp. 69–74.

[26] (2018). *Software Assurance Reference Dataset*. [Online]. Available: https://samate.nist.gov/SRD/index.php

[27] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.

[28] A. B. Farid, E. M. Fathy, A. S. Eldin, and L. A. Abd-Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)," *PeerJ Comput. Sci.*, vol. 7, p. e739, Nov. 2021.

[29] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.

[30] M. S. Akhtar and T. Feng, "Detection of malware by deep learning as CNN-LSTM machine learning techniques in real time," *Symmetry*, vol. 14, no. 11, p. 2308, Nov. 2022.

[31] E. U. H. Qazi, M. H. Faheem, and T. Zia, "HDLNIDS: Hybrid deep-learning-based network intrusion detection system," *Appl. Sci.*, vol. 13, no. 8, p. 4921, Apr. 2023.

[32] B. Xue and N. Tong, "DIOD: Fast and efficient weakly semi-supervised deep complex ISAR object detection," *IEEE Trans. Cybern.*, vol. 49, no. 11, pp. 3991–4003, Nov. 2019.

[33] J. Du, C.-M. Vong, and C. L. P. Chen, "Novel efficient RNN and LSTM-like architectures: Recurrent and gated broad learning systems and their applications for text classification," *IEEE Trans. Cybern.*, vol. 51, no. 3, pp. 1586–1597, Mar. 2021.

[34] Y. Zhou and Y. Wei, "Learning hierarchical spectral–spatial features for hyperspectral image classification," *IEEE Trans. Cybern.*, vol. 46, no. 7, pp. 1667–1678, Jul. 2016.

[35] M. Fu, H. Qu, Z. Yi, L. Lu, and Y. Liu, "A novel deep learning-based collaborative filtering model for recommendation system," *IEEE Trans. Cybern.*, vol. 49, no. 3, pp. 1084–1096, Mar. 2019.

[36] M. He, B. Li, and H. Chen, "Multi-scale 3D deep convolutional neural network for hyperspectral image classification," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 3904–3908.

[37] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Phys. D, Nonlinear Phenomena*, vol. 404, Mar. 2020, Art. no. 132306.

[38] M. Sundermeyer, H. Ney, and R. Schlüter, "From feedforward to recurrent LSTM neural networks for language modeling," *IEEE/ACM Trans. Audio, Speech, Language Process.*, vol. 23, no. 3, pp. 517–529, Mar. 2015.

[39] S. Fernández, A. Graves, and J. Schmidhuber, "An application of recurrent neural networks to discriminative keyword spotting," in *Proc. Artif. Neural Netw. (ICANN)*, 2007, pp. 220–229.

[40] Y. Bengio, *Learning Deep Architectures for AI*. USA: Now Foundations and Trends, 2009.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[42] A. Miaschi and F. Dell'Orletta, "Contextual and non-contextual word embeddings: An in-depth linguistic investigation," in *Proc. Assoc. Comput. Linguistics*, 2020, pp. 110–119.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[44] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[45] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.

[46] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 595–614.

[47] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 201–213.

[48] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–36, Jul. 2018.

[49] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proc. ACM Symp. Appl. Comput.*, Mar. 2010, p. 1963.

[50] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[51] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2008, p. 315.

[52] M. Zagane and M. Kamel Abdi, "Evaluating and comparing size, complexity and coupling metrics as web applications vulnerabilities predictors," *Int. J. Inf. Technol. Comput. Sci.*, vol. 11, no. 7, pp. 35–42, Jul. 2019.

[53] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2013, pp. 499–510.

[54] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *Proc. 19th Int. Symp. Softw. Test. Anal.*, Jul. 2010, pp. 119–130.

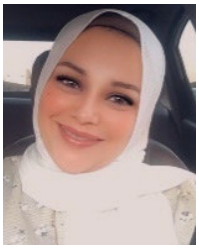
[55] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, Dec. 2012, pp. 359–368.

[56] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 797–812.

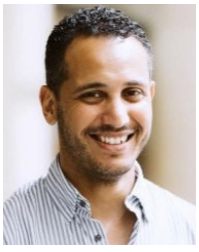
[57] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016, pp. 85–96.
[58] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*.
[59] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proc. 5th USENIX Conf. Offensive Technol.*, 2011, p. 13.
[60] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep learning for software vulnerabilities detection using code metrics," *IEEE Access*, vol. 8, pp. 74562–74570, 2020.
[61] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 292–303.
[62] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, Oct. 2017, pp. 2539–2541.
[63] R. Li, C. Feng, X. Zhang, and C. Tang, "A lightweight assisted vulnerability discovery method using deep neural networks," *IEEE Access*, vol. 7, pp. 80079–80092, 2019.
[64] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.
[65] *DB-CBIL Model*. Accessed: May 28, 2023. [Online]. Available: https://github.com/AyaElRahman/DB-CBIL
[66] *ANTLR*. Accessed: Aug. 15, 2023.[Online]. Available: https://www.antlr.org/
[67] *Code Sensor ANTLR Parser*. Accessed: Aug. 15, 2023. [Online]. Available: https://github.com/fabsx00/codesensor
[68] *DB-CBIL-Dataset*. Accessed: May 23, 2023. [Online]. Available: https://github.com/AyaElRahman/DB-CBIL-Dataset
[69] *TensorFlow*. Accessed: Jan. 16, 2023. [Online]. Available: https://www.tensorflow.org/

AHMED BAHAA received the Ph.D. degree in information systems from the Faculty of Computers and Information, Helwan University. He has been the Microsoft Regional Director, since 2010. He is currently the Vice Dean of Community Service and Environmental Affairs of the Faculty of Computers and Artificial Intelligence, Beni-Suef University. He is a Professor with the Faculty of Computers and Artificial Intelligence, Helwan University. You can reach his Microsoft Official Bio at the following link:https://mvp.microsoft.com/en-us/RD/profile/44fd8fa8-3c9a-e411-93f2-9cb65495d3c4.

AYA EL-RAHMAN KAMAL was born in Cairo, Egypt, in 1993. She received the B.S. degree in information systems from Helwan University, in 2016, where she is currently pursuing the M.Sc. degree in software security. She is a Teaching Assistant with the Faculty of Computers and Artificial Intelligence, Helwan University. She has more than six years of experience in the field of software quality control. Her research interests include artificial intelligence, machine learning, software engineering, and software security.



HANAN FAHMY received the Ph.D. degree in information systems from the Faculty of Artificial Intelligent, Helwan University, Egypt. She is currently an Associate Professor with the Department of Information Systems, Faculty of Artificial Intelligent, Helwan University. Her research interests include security, machine learning, artificial intelligence, big data, data analytics, the IoT data analytics, advanced database management, and software engineering.



AMR S. GHONEIM received the B.S. and M.Sc. degrees in computer science from Helwan University and the Ph.D. degree in computer science from the University of New South Wales (UNSW), Australia. He is currently an Assistant Professor with the Faculty of Computers and Artificial Intelligence, Helwan University. His research interests include artificial intelligence, artificial neural networks, machine learning, ensemble learning, data mining, and pattern recognition.