Journal of Information Hiding and Multimedia Signal Processing ©2025 ISSN 2073-4212
Ubiquitous International Volume **16**, Number **3**, September **2025**

DL-SCVDetect: An Attention-Driven Bidirectional LSTM with Random Forest Classifier for Intelligent Smart Contract Vulnerability Detection

Saad AL Azzam*

Institute of Computer Science and Digital Innovation
UCSI University, Kuala Lumpur, Malaysia
1002372379@ucsiuniversity.edu.my

Raenu AL Kolandaisamy

Institute of Computer Science and Digital Innovation
UCSI University, Kuala Lumpur, Malaysia
raenu@ucsiuniversity.edu.my

Ghassan Saleh ALDharhani

Institute of Computer Science and Digital Innovation
UCSI University, Kuala Lumpur, Malaysia
ghassan@ucsiuniversity.edu.my

*Corresponding author: Saad AL Azzam
Received May 27, 2025, revised July 8, 2025, accepted July 9, 2025.

Abstract. *The swift growth of blockchain (BC) has made decentralized transaction systems possible, encouraging many industries to adopt this technology due to its more security. Smart contracts (SC), the foundation of blockchain (BC), are self-operating agreements that execute based on preset conditions. High-level programming languages are used to write smart contracts (SC), translating agreement terms into lines of code. The rise of smart contracts has reshaped decentralized apps by automating transactions on blockchain networks, but has also exposed systems to security flaws and financial risks. A notable example is the 2016 DAO attack, which led to the loss of $60 million due to insecure SC. This research aims to detect vulnerabilities in SC systems by applying machine learning and deep learning (DL) techniques to BC systems. The proposed approach relies on a binary classification process using a Random Forest algorithm to determine whether the SC contains a vulnerability or not. If a node contains a vulnerability, multi-class classification is used to determine the vulnerability type. A hybrid model is used to embed words into a 128-bit vector, followed by a bidirectional LSTM supported by an attention layer to direct attention to the most pertinent elements within the input sequence. Dimensionality is then reduced using one-dimensional convolution and max pooling. The Random Forest achieved an accuracy of 94.19% (binary classification) and 95.9% (multi-class classification). This research contributes to improving the accuracy and efficiency of SC vulnerability detection and provides a proactive and adaptive solution to enhance the security of BC-based applications.*

Keywords: Blockchain; Smart Contracts; Vulnerability Detection; Machine Learning; Deep Learning

1. **Introduction.**

Smart contracts (SCs) have gained substantial traction in the research community, driven by their promise to transform traditional contract execution through blockchain-based automation. In recent years, blockchain's decentralized nature has provided a dependable framework for executing SCs without external intermediaries [1]. SC refers to a blockchain-based agreement that triggers its execution automatically once certain criteria are fulfilled. Put simply, SCs resemble 'if-then' constructs used in software, and their execution can have direct effects on tangible assets in the real world [2, 3]. SCs offer several key advantages over traditional contracts: (1) minimized transaction risks, (2) lower administrative and service expenses, and (3) enhanced business process efficiency [4, 5].

Detecting security vulnerabilities is a very challenging task due to the diversity of contract types, where each programmer has their way of formulating code, defining variables, defining functions, and choosing their names. This leads to the possibility of multiple contract types performing the same task, so traditional auditing processes, such as cosine similarity, face multiple problems in determining whether a contract contains a vulnerability or not [6]. Furthermore, the increasing sophistication of SCs, along with the expansion of decentralized finance (DeFi), has led to an expansion in the scope, diversity, and continuous evolution of attacks, necessitating adaptive security measures to mitigate these threats and deal with emerging vulnerabilities [7]. In response, many researchers have developed different techniques for discovering vulnerabilities. Some rely on static analysis (examining code without execution), others on dynamic analysis (runtime testing in a controlled environment), and others on machine learning (ML)-based approaches that rely on analyzing historical data that includes previously discovered decades that contain vulnerabilities and the types of these vulnerabilities. Each of these methods has specific strengths, and its effectiveness depends on the context in which the code is written and the type of vulnerability being analyzed [8, 9, 10].

Automated tools have been developed to detect vulnerabilities in SCs, examples of which are Mythril [11], Slither [12], and Oyente [13]. But these tools still face limitations, such as scalability, high false positive rates, and difficulties in detecting new or complex vulnerabilities [10].

This paper presents a two-stage hybrid framework, DL-SCVDetect, that combines traditional machine learning (Random Forest) with deep learning techniques (Bi-LSTM and Attention) to detect and classify vulnerabilities in Solidity smart contracts. Unlike most previous work, which focuses on binary classification or single-type vulnerabilities, our model performs binary and multi-class classification. The key highlights of this paper are:

- Treating Solidity source code as a natural language sequence and applying symbolic normalization and tokenization to reduce semantic variation. While most previous research has dealt with contract bytecode, our paper uses source code, which provides more accurate semantic information that can be captured using deep learning models and leveraged in the vulnerability classification process.
- Introducing an attention-enhanced Bi-LSTM model to better capture bidirectional contextual dependencies in code semantics.
- Combining Conv1D and Global Max Pooling to efficiently reduce features and extract patterns before classification.
- Address class imbalance using SMOTE to avoid biased models for the majority classes.

These contributions collectively result in a more adaptive, accurate, and generalizable approach for smart contract vulnerability detection.

2. **Literature Reviews.** The paper [14] presents an approach to detect vulnerabilities in Solidity-based SC using an MLP-ANN model. The authors leverage Opcode and Control Flow Graph (CFG) features extracted from SC to train the proposed model. The authors rely on using an imbalanced dataset and then balancing it through a fault injection method. Although fault injection is good for dataset balancing, relying on artificial vulnerability injection may lead to gaps in capturing real-world vulnerability patterns. The paper used techniques such as 3D vectors and TFIDF to generate feature embeddings. This limitation can affect the robustness of the model in real-time scenarios. The validity of the research can be enhanced by relying on sampling techniques to balance the dataset, which is based on creating synthetic samples based on the analysis of existing samples. The research compares the proposed model with existing static and dynamic analysis tools, and no comparison was made with other ML or DL models, so including models such as transformer-based architectures could provide better performance. Considering more powerful feature extraction methods, such as BERT and RoBERTa, could capture complex relationships and patterns within SC code more accurately and effectively, providing richer feature sets for vulnerability detection.

The research [15] presented a tool called SliSE, designed to detect reentry vulnerabilities in SC. Program slicing is used to examine the I-PDG of a SC, generating warnings about potential vulnerabilities. SliSE reached an F1 score of 78.65%. The paper does not discuss how the proposed tool handles large or complex contracts. The symbolic execution can be very expensive in terms of resources and may have scaling issues, especially as the contract size increases. Also, the applicability of SliSE is limited to this single type of vulnerability, which reduces the generalizability of the model.

The research [16] presented the DA-GNN model for detecting vulnerabilities in SC. Traditional SC vulnerability detection methods rely heavily on predefined rules, which limit their accuracy and adaptability. DA-GNN seeks to address these limitations by implementing a dual attention mechanism within the graph attention network to enhance the extraction of relevant features for vulnerability detection. The model doesn't address other vulnerabilities or new vulnerabilities that may arise with the continuous development of attack mechanisms and techniques. Graph neural networks are computationally expensive and require high resources, especially when working on large and complex SC.

The research [17] presented a model called BiGAS, which is a model for detecting reentry vulnerabilities in SC. The authors replace the SoftMax classifier with an SVM classifier. The model is designed to detect reentry, which limits its applicability across a wider range of vulnerabilities that also pose risks to the security of SC.

The study [18] presents the HAM model that improves the detection accuracy of five types of vulnerabilities. The proposed approach for detecting vulnerabilities in SC is based on the HAM model. The approach consists of three main phases:

- Code Fragments Extraction: In this phase, the source code of SC is analyzed to extract code fragments that are likely to contain vulnerabilities.
- Training Phase: The model utilizes both single-head and multi-head attention encoders to capture different aspects of the code's semantic and contextual information.
- Finally, the model employs a fully connected network to classify the code fragments as vulnerable or not.

This paper did not take into account the balance of the data set and was limited to 5 types of vulnerabilities without taking into account other types of vulnerabilities or emerging vulnerabilities.

The study [19] introduces ASSBert. It uses a semi-supervised bi-encoder representation from a transformer (BERT) network for SC vulnerability classification. This approach ensures optimal performance with minimal labeled data and a large pool of unlabeled data. Within this framework, active learning identifies highly uncertain code samples from unlabelled Solidity (sol) files, which are then manually labeled and added to the training set. Meanwhile, semi-supervised learning continuously selects a set of high-confidence unlabelled code samples, assigns pseudo-labels to them, and incorporates them into the training dataset. Experimental results indicate that ASSBert outperforms baseline methods while relying on minimal labeled data and extensive unlabeled data. Table **??** summarizes the previous literature reviews.

Finally, inspired by hybrid quantum-classical approaches in other domains, such as the Hybrid Quantum Neural Network Classifier for spine image classification [20], our work demonstrates a similar fusion of advanced architectures (Bidirectional LSTM and Random Forest) to address the distinct challenges of smart contract vulnerability detection.

Table 1. Summary of literature reviews

Ref	**Year**	**Detection Method**	**Vulnerabilities Covered**	**Key Strengths**	**Gaps**
[14]	2024	Multi-Layer Perceptron (MLP)	Opcode Features, CFG-Based Detection	Standardized preprocessing, balanced dataset with synthetic errors	Synthetic errors may introduce bias
[15]	2024	Program Slicing + I-PDG Analysis	Reentrancy Vulnerabilities	Uses program slicing for inter-contract analysis	Limited to reentrancy, scaling issues for large contracts
[16]	2024	Graph Neural Networks (GNNs) + Dual Attention	SC Vulnerabilities	Improved feature extraction for contract analysis	High computational demand
[17]	2024	Bi-GRU + SVM	Reentrancy Vulnerabilities	Improved classification accuracy	Limited to reentrancy
[18]	2023	Hybrid Attention + Neural Network	5 SC Vulnerabilities	Captures both single and multi-head attention	No dataset balancing, limited to 5 vulnerabilities
[19]	2023	Bert	19 SC Vulnerabilities	Minimal labeled data and extensive unlabeled data	No dataset balancing may introduce bias

3. Methodology.

The proposed methodology addresses the problem of analyzing Solidity code as a Natural Language Processing (NLP) problem, and the working mechanism will be explained in detail in the subsequent sections. Figure 1 illustrates the proposed methodology's overall design, outlining the key components and processes involved in the approach.

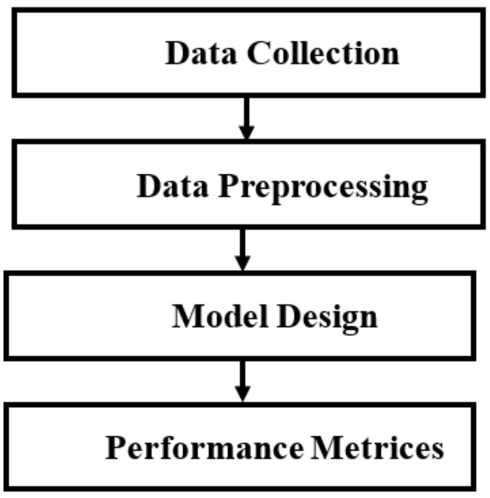


FIGURE 1. The proposed methodology's overall design

3.1. **Data Collection.** The dataset contains 2,217 Ethereum SC, covering four types of vulnerabilities, categorized as follows:

- Timestamp Dependency: 540 samples
- Reentrancy: 737 samples
- Integer Overflow: 630 samples
- Dangerous Delegatecall: 310 samples

These vulnerable contracts were sourced from Kaggle repositories. The dataset only contains vulnerabilities within SC. Therefore, valid SC were manually collected from Etherscan, a platform for creating and analyzing Ethereum contracts. Etherscan was created and launched in 2015 and is one of the oldest and longest-running independent projects created for Ethereum to provide fair access to BC data [21]. Approximately 366 valid SC written in Solidity were collected.

The final dataset is stored in a structured CSV file where each row contains:

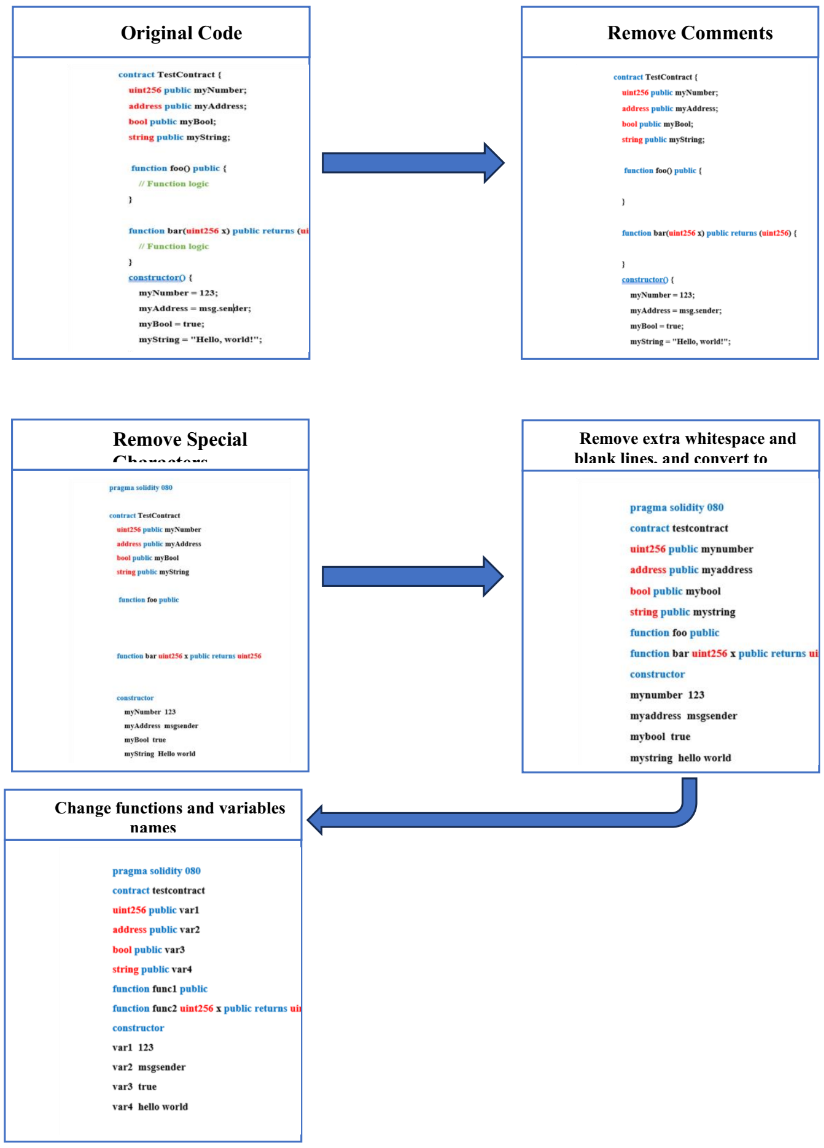
- A smart contract source code written in the Solidity programming language.
- A corresponding label representing the vulnerability type (or `Normal` for non-vulnerable samples).

3.2. **Data Preprocessing.** Preprocessing Solidity code before feeding it into a DL model involves converting the raw source code into a format that the model can understand and process effectively. To illustrate the changes that occur to the code after each preprocessing operation, a simple Solidity code is used and the changes after preprocessing operations are shown. The following are the preprocessing steps that will be performed:

3.2.1. *Code Cleaning.* In the cleaning phase, unnecessary characters, symbols, and special characters are removed, and punctuation marks, any non-text elements, and special characters are isolated. The following explains the operations performed in this phase:

- Remove comments (either single-line or multi-line).
- Remove special characters and punctuation.
- Remove extra white spaces and empty lines, and convert to lowercase.
- In code, most function identifiers and variables are usually given arbitrary names, where each programmer chooses the appropriate names for themselves. Therefore, these names are meaningless and have nothing to do with security vulnerabilities, and can affect the accuracy of classification performance because the model will consider

each of them as a new word, but rather they represent only a name. Therefore, perform a normalization process for the program code by assigning user-defined variables and functions to symbolic names (for example, “VAR1”, “FUNC1”) one by one.


Original Code
Remove Comments
Remove Special
Remove extra whitespace and
blank lines, and convert to
Change functions and variables names

```
contract TestContract {
  uint256 public myNumber;
  address public myAddress;
  bool public myBool;
  string public myString;

   function foo() public {
     // Function logic
  }

  function bar(uint256 x) public returns (ui
     // Function logic
  }
  constructor() {
     myNumber = 123;
     myAddress = msg.sender;
     myBool = true;
     myString = "Hello, world!";
```

```
contract TestContract {
  uint256 public myNumber;
  address public myAddress;
  bool public myBool;
  string public myString;

   function foo() public {


  }


  function bar(uint256 x) public returns (uint256) {


  }
  constructor() {
     myNumber = 123;
     myAddress = msg.sender;
     myBool = true;
     myString = "Hello, world!";
```

```
pragma solidity 080

contract TestContract
  uint256 public myNumber
  address public myAddress
  bool public myBool
  string public myString

   function foo public



  function bar uint256 x public returns uint256


  constructor
     myNumber  123
     myAddress  msgsender
     myBool  true
     myString  Hello world
```

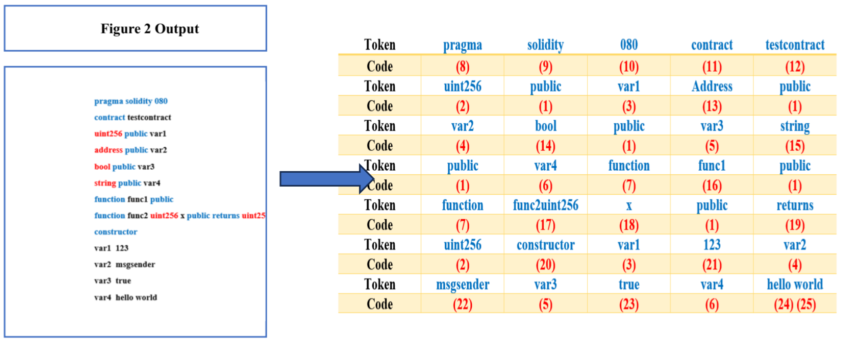
```
pragma solidity 080
contract testcontract
uint256 public mynumber
address public myaddress
bool public mybool
string public mystring
function foo public
function bar uint256 x public returns ui
constructor
mynumber  123
myaddress  msgsender
mybool  true
mystring  hello world
```

```
pragma solidity 080
contract testcontract
uint256 public var1
address public var2
bool public var3
string public var4
function func1 public
function func2 uint256 x public returns ui
constructor
var1  123
var2  msgsender
var3  true
var4  hello world
```

FIGURE 2. Code cleaning processes

3.2.2. *Tokenization.* Tokenization is the act of dividing a string or text into smaller units known as tokens, which can range from words to phrases or symbols, depending on the task. Each token is assigned to a number. Figure 3 shows the Solidity code after tokenization.

3.2.3. *Padding.* Padding is the practice of appending special characters, often zeros, to sequences so that they all have the same length. The goal of this process is to meet the requirement that many ML models expect input sequences of uniform size. Padding ensures consistency by making sure all sequences are of equal length.



```
pragma solidity 080
contract testcontract
uint256 public var1
address public var2
bool public var3
string public var4
function func1 public
function func2 uint256 x public returns uint25
constructor
var1  123
var2  msgsender
var3  true
var4  hello world
```

Token	pragma	solidity	080	contract	testcontract
Code	(8)	(9)	(10)	(11)	(12)
Token	uint256	public	var1	Address	public
Code	(2)	(1)	(3)	(13)	(1)
Token	var2	bool	public	var3	string
Code	(4)	(14)	(1)	(5)	(15)
Token	public	var4	function	func1	public
Code	(1)	(6)	(7)	(16)	(1)
Token	function	func2uint256	x	public	returns
Code	(7)	(17)	(18)	(1)	(19)
Token	uint256	constructor	var1	123	var2
Code	(2)	(20)	(3)	(21)	(4)
Token	msgsender	var3	true	var4	hello world
Code	(22)	(5)	(23)	(6)	(24) (25)

FIGURE 3. Tokenization process

3.3. **Model Design.** This research will employ two models. The first uses a random forest algorithm to detect whether a contract is healthy or vulnerable. If the contract is healthy, no action will be taken. However, if the contract is vulnerable, DL techniques will be utilized to classify 4 vulnerabilities kinds described in Section 1.3. Figure 4 illustrates the layered structure of the proposed model.

- Input layer: Input sequences of 128 length are the expected input for the model.
- Embedding layer: Every token is embedded into a vector of 128 elements through this layer.
- Bidirectional Long Short-Term Memory (Bi-LSTM) layer: By bidirectionally processing these sequences (both forwards and backwards), this layer generates a set of 256-dimensional vectors.
- Attention layer: It calculates attention scores based on the BI-LSTM outputs, highlighting key parts of the input sequence to enhance prediction accuracy.
- Convolutional layer (Conv1D): This component decreases the dimensions of the data on its input. In our model, the output after the Conv1D layer is (None, 126, 64), indicating that the sequence length has been reduced from 128 to 126, and the number of features has been reduced to 64.
- Global-Max-Pooling-1D: Apply max-pooling to reduce the dimensionality of the data and capture the most influential feature in the feature maps individually.

3.3.1. *Embedding Layer.* This layer transforms each word into a fixed-size dense vector composed of real-valued numbers. This transformation helps represent words more effectively while reducing dimensionality [22]. Essentially, it functions as a search table, where words serve as keys and their corresponding vectors act as values. Word embeddings are particularly useful in natural language processing (NLP) tasks where they capture contextual relationships, so the words with similar meanings have closely related vector representations.

3.3.2. *Bidirectional LSTM.* Before explaining the bidirectional LSTM architecture, it is important to first understand the unidirectional LSTM architecture. LSTM belongs to the recurrent neural networks (RNNs) family and is distinguished by memory cells that can preserve information for extended durations [23]. An internal state is maintained by the cell, allowing it to be modified according to both current inputs and previous outputs. An LSTM cell has three gates [24]:

- Forget gate: It detects what past info is no longer relevant and should be removed. Upon receiving input data, this gate identifies valuable information to retain and discards irrelevant data, enhancing the performance and training speed of the recurrent neural network. Its functioning depends on two inputs: the output of the previous

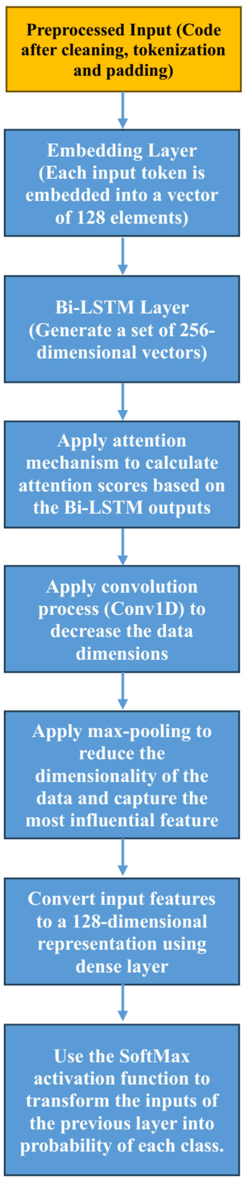

Preprocessed Input (Code after cleaning, tokenization and padding)
Embedding Layer (Each input token is embedded into a vector of 128 elements)
Bi-LSTM Layer (Generate a set of 256-dimensional vectors)
Apply attention mechanism to calculate attention scores based on the Bi-LSTM outputs
Apply convolution process (Conv1D) to decrease the data dimensions
Apply max-pooling to reduce the dimensionality of the data and capture the most influential feature
Convert input features to a 128-dimensional representation using dense layer
Use the SoftMax activation function to transform the inputs of the previous layer into probability of each class.

FIGURE 4. The layered structure of the proposed model

node and the current input. Once the matrix's weights and biases are established, each neuron is passed through the sigmoid activation function. Information is retained based on the gate's output values: a value of 0 prompts the forget gate to discard the data, while a value of 1 indicates important information to be kept.

- Input gate: This mechanism identifies the relevant new data that should be retained and included in the present state. This gate handles the incoming data, and the number of neurons it involves depends on how much data is provided. Using the tanh function, input values are constrained to the range [-1, 1], modifying the data before feeding it into the subsequent layer.
- Output gate: It produces the output according to the current state. The data of interest is passed through this cell from the previous one, where a matrix of values is created using the tanh function within the [-1, 1] range, and the sigmoid function generates the final output values.

Based on the above, bidirectional LSTM has two LSTMs, one that processes data sequentially (start-to-end) and another that processes data in reverse order. This approach

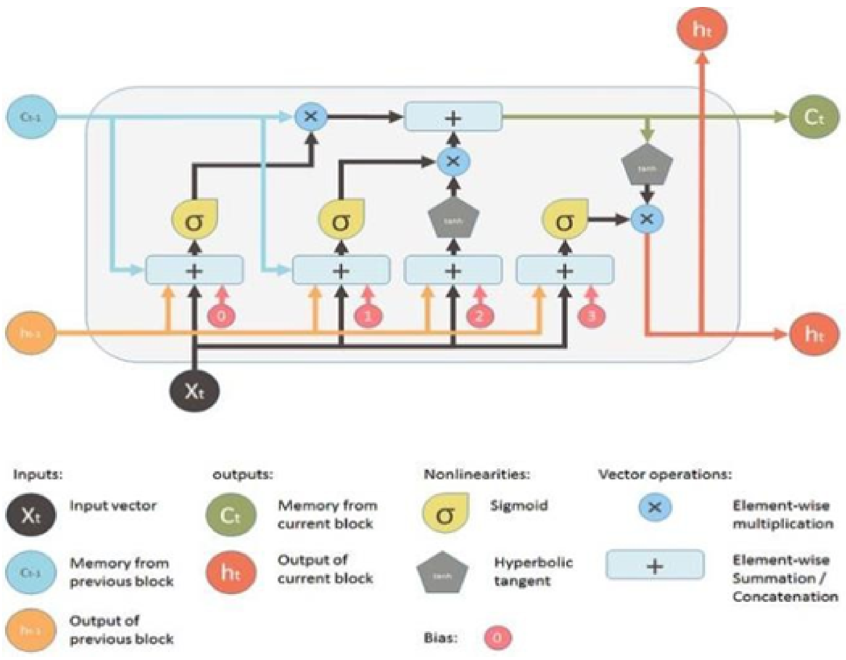


Figure 5. The structure of an LSTM cell

helps to better understand the context of the sequence by modeling, such as recognizing both preceding and following words in the sentence.

3.3.3. *Attention Layer.* It allows DL models to prioritize critical input elements, leading to more accurate predictions and making computations more efficient. By prioritizing key information, it highlights important features to boost the model's overall effectiveness. Three fundamental components make up this layer architecture: (1) the encoder, (2) the attention module, and (3) the decoder [25]. Figure 6 presents the attention mechanism's structure:

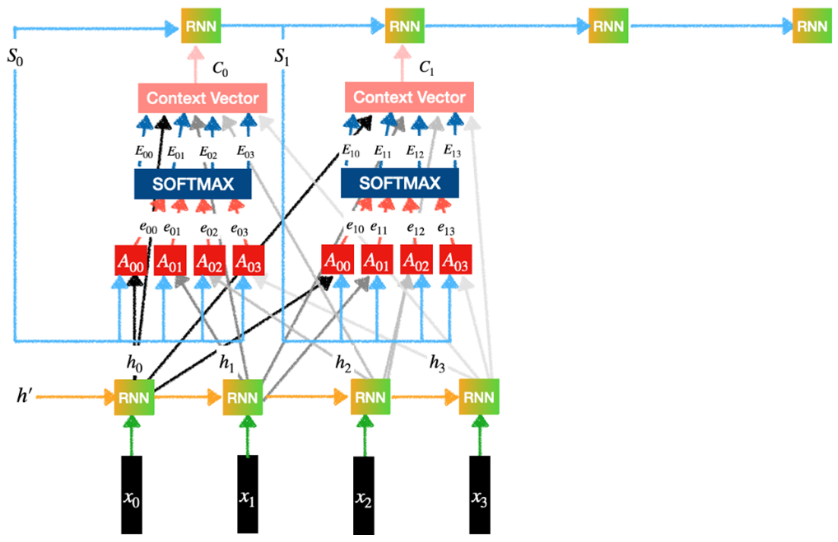


Figure 6. The attention mechanism's structure

By utilizing RNNs, the encoder processes the input sequence in a series of iterations. At each step, the encoder produces a hidden state that incorporates both the previous hidden state and the current input symbol, so to produce the full input sequence, the combination of these hidden states is captured [26]. The attention module contains: a feed-forward network, SoftMax, and a context vector, which is fed to the decoder with the current hidden state to predict the next symbol in the output sequence [26]. In this way, this process is repeated until the entire output sequence is generated [26].

3.3.4. *Convolution Layer (Conv1D).* In DL, the Conv1D layer is designed to operate on sequential convolutional operations along a single axis. In a Conv1D layer, the convolutional filter (or kernel) moves across the input sequence, carrying out element-wise multiplications with the values inside its receptive field [27]. The weighted values are summed to produce a single output, and this is done iteratively across all positions in the sequence to generate a transformed output sequence. Filters are composed of parameters that can be learned and fine-tuned during training to detect relevant patterns in the input data. Figure 7 illustrates an example of a Conv1D operation [27].

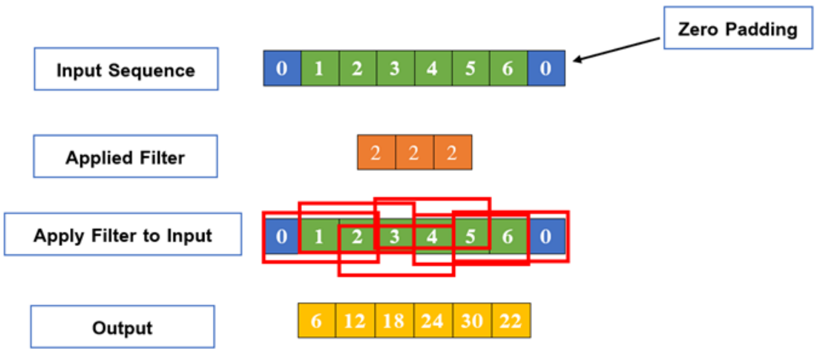


FIGURE 7. An example of a Conv1D operation

Each filter in the conv-layer performs the operation illustrated in Figure 7. This produces several outputs, commonly known as feature maps. Increasing the number of filters enhances the model's ability to extract diverse features, allowing it to identify patterns within the input data more effectively.

3.3.5. *Global Max Pooling Layer.* This layer serves to reduce feature maps by compressing each set of input samples of a specified size into a single sample. This size is a tiny window. The optimal window size is 1x2 when using a one-dimensional convolutional layer. Increasing the window size beyond this may result in the loss of important information [28]. The outcome of applying the described method to Figure 7's output is illustrated in Figure 8.

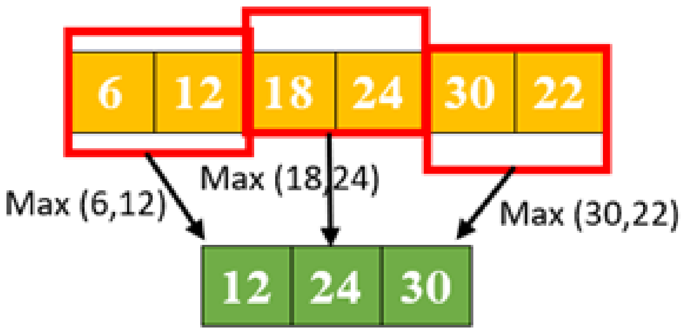


FIGURE 8. Max Pooling process

3.3.6. *Dense Layer.* In this layer, every neuron is connected to all neurons in the preceding layer, which is why it is referred to as "fully connected." As a result, inputs to each neuron come from all neurons in the earlier layer. The connections between neurons have associated weight parameters, which are learned during the training process. This layer computes the sum of input values multiplied by their corresponding weights, which are refined through learning [29].

4. Results and Discussions.

4.1. **Handling dataset imbalance.** The dataset used contains 2,217 vulnerable SC and 366 healthy SC, making the dataset unbalanced. Imbalanced datasets are a major challenge in the field of artificial intelligence in general. This may result in various issues [30], such as:

- Biased models: In this scenario, the model exhibits a bias toward the majority class, meaning it gives more weight to the dominant class while struggling to learn patterns from the minority classes. As a result, this imbalance leads to suboptimal model performance.
- Unreliable performance metrics (such as accuracy) can be misleading when dealing with imbalanced data. While they may yield high scores, these results do not accurately reflect the model's effectiveness. This is because the model tends to correctly predict the majority class more often, overshadowing its poor performance on the minority class.
- Overfitting: This means that the model may primarily learn the majority class patterns and features, leading to weak generalization and poor performance in the minority class.

Figure 9 shows the count of samples per class.

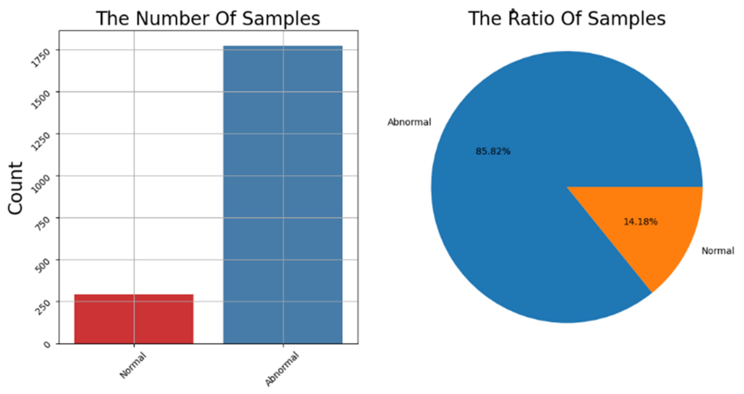


FIGURE 9. The count of samples per class

We employed the Synthetic Minority Oversampling Technique (SMOTE) to handle the imbalance in our dataset, generating synthetic samples for the minority class [31]. By creating artificial examples, SMOTE helps balance the dataset, enhancing the model's ability to learn from the minority class and improving overall performance on imbalanced data [31]. Figure 10 shows the number of samples for each class after applying SMOTE.

4.2. **The results of the Random Forest Algorithm (Binary Classification).** We employed the hold-out method to split the dataset into training and testing sets. Specifically, 70% of the data was allocated for training, while the remaining 30% was reserved for testing. Since the test data remains unseen by the model during training, it allows for

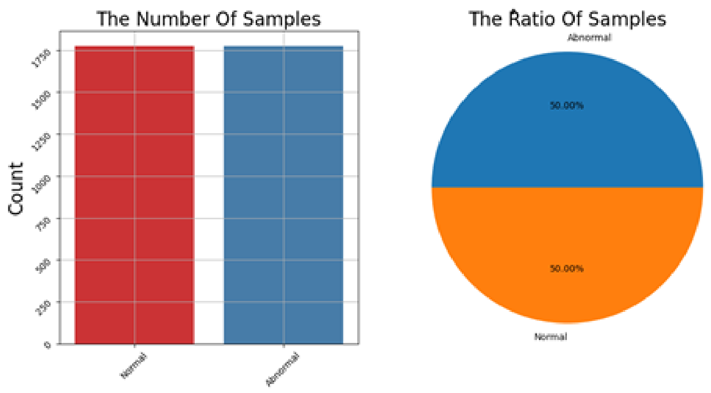


FIGURE 10. The number of samples for each class after applying SMOTE

an objective evaluation of the model's performance and provides valuable insights into its generalization ability to new, real-world data. Figure 11 presents the confusion matrix results for the Random Forest algorithm, obtained by applying the model to the test dataset.

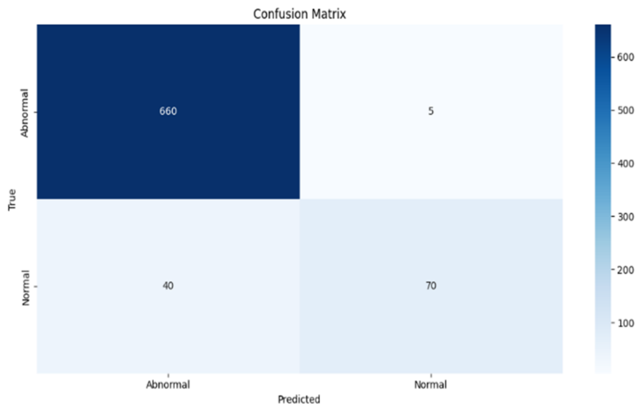


FIGURE 11. The confusion matrix results for the Random Forest algorithm

The confusion matrix provides a quick overview of the Random Forest algorithm's performance in predicting whether a SC is healthy or vulnerable. The elements on the main diagonal of the confusion matrix represent the correct predictions for each class. For example, the element in the top left (660) indicates that the model predicted 660 contracts as vulnerable, when in fact they are vulnerable. The element in the bottom right (70) indicates that the model predicted 70 contracts as unvulnerable, when in fact they are healthy. Elements outside the main diagonal represent misclassification. For example, the element in the bottom left (40) indicates that the model predicted 40 contracts as vulnerable, when in fact they are healthy, so the classification is incorrect. The element in the top right (5) indicates that the model predicted 5 contracts as unvulnerable, when in fact they are vulnerable, so the classification is incorrect. Table 2 shows the performance metrics for each class resulting from the Random Forest algorithm.

It can be seen from Table 2 that:

- The precision value for the Normal class is 0.93, meaning that of all samples predicted as Normal, 93% were Normal.

Table 2. The performance metrics for each class resulting from the Random Forest algorithm

	Accuracy	F1_Score	Recall	Precision
Normal	0.9419	0.76	0.64	0.93
Abnormal		0.97	0.99	0.94

- The Normal class has a recall of 0.64, which means that of all samples predicted as Normal, the model correctly predicted 64% of them.
- The F1_Score of 0.76 indicates that the model has a moderate balance between precision and recall for the Normal class.
- The precision value for the Abnormal class is 0.94, meaning that of all samples predicted as Abnormal, 94% were actually Abnormal.
- The Abnormal class has a recall of 0.99, which means that of all samples predicted as Abnormal, the model correctly predicted 99% of them.
- The F1_Score value of 0.97 indicates that the model has an excellent balance between precision and recall for the Abnormal class.
- An Accuracy value of 0.9419 indicates that the model correctly predicted 94.19% of the total number of samples in the test set.

4.3. **Training Configuration and Hyperparameters of the proposed model.** The proposed deep learning model was implemented using TensorFlow 2.12 and the Keras framework. All experiments were conducted on a system equipped with an Intel Core i5-13420H processor, 16 GB RAM, and an NVIDIA RTX 3050 GPU. The training details are summarized as follows:

- Batch size: 32.
- Epochs: 10.
- Optimizer: Adam.
- Initial learning rate: 0.0001.
- Loss function: Categorical Crossentropy (for multi-class classification).
- Evaluation metrics: Accuracy, Precision, Recall, and F1-Score.
- Early stopping: Applied with a patience of 3 epochs on validation loss to prevent overfitting.
- Validation split: 20% of the training data was used as the validation set.
- Random seed: Fixed to ensure reproducibility across runs.
- Training and validation performance were monitored using learning curves for both accuracy and loss.
- All hyperparameters were tuned experimentally using grid search to maximize generalization on unseen test data.

4.4. **The proposed model results.** During model training, a learning curve tracks the development of a chosen metric, plotting progress on the x-axis and error or performance on the y-axis. By tracking performance over time, the learning curve helps monitor progress and reveals any underlying issues during training. One of the most standard metrics plotted is the loss function curve, showing the progression of model error throughout training. As loss values decline, the model tends to perform more accurately. The accuracy curve is another common learning curve, reflecting the model's performance over time. Rising values suggest the model is learning effectively and becoming more accurate. Figure 12 presents the curves for accuracy and loss during both the training and validation stages.

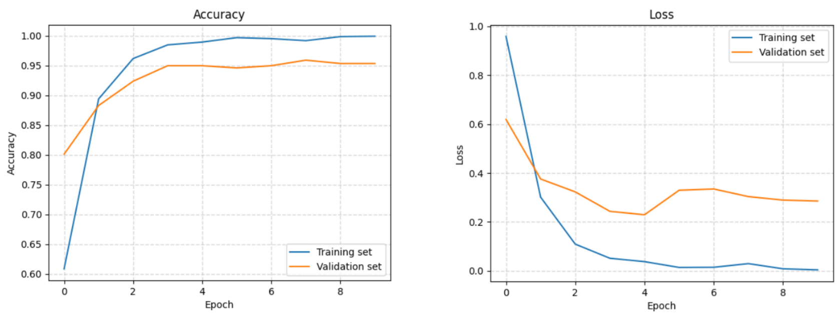


Figure 12. The curves for accuracy and loss during both the training and validation stages

Figure 12 shows that both the training and validation metrics show an upward trend, demonstrating that the model is effectively learning from the data and performing well on both known (training) and unseen (validation) examples. The decreasing loss and increasing accuracy in both datasets indicate successful convergence and improved generalization. Additionally, the model avoids overfitting, a common issue where a model excels on training data but struggles with new data. The proposed model achieved an accuracy of 99% on the training set and 96% on the validation set, confirming its strong performance. The confusion-matrix, obtained by applying the proposed model to the test data, is shown in Figure 13.

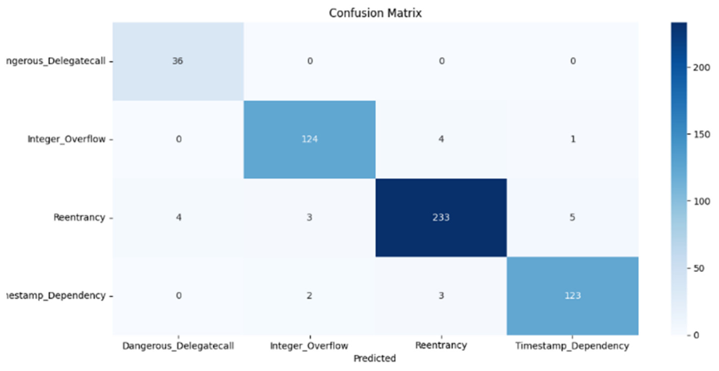


Figure 13. The results of the confusion matrix obtained from applying the proposed model to the test data

Based on the confusion matrix results, we note that:

- The proposed model correctly classified 36 samples containing a Dangerous Delegatecall vulnerability and did not misclassify any samples containing this vulnerability. The model classified four samples as containing a Dangerous Delegatecall vulnerability, but they contained a Reentrancy vulnerability.
- Regarding the samples containing an Integer Overflow vulnerability, we note that the model correctly classified 124 samples, while the model incorrectly classified three samples as containing an Integer Overflow vulnerability, when they actually contained a Reentrancy vulnerability. The model also incorrectly classified two samples as containing an Integer Overflow vulnerability when they contained a Timestamp Dependency vulnerability.

- For samples containing a Reentrancy vulnerability, we note that the model correctly classified 233 samples, while incorrectly classifying 4 samples as containing a Reentrancy vulnerability when they contained an Overflow Integer vulnerability. The model also incorrectly classified 3 samples as containing a Reentrancy vulnerability when they contained a Timestamp Dependency vulnerability.
- For samples containing a Dependency Timestamp vulnerability, we note that the model correctly classified 123 samples, while incorrectly classifying 1 sample as containing a Dependency Timestamp vulnerability when it contained an Overflow Integer vulnerability. The model also incorrectly classified 5 samples as containing a Timestamp Dependency vulnerability when they contained a Reentrancy vulnerability.

Table 3 shows the performance metrics for each class resulting from applying the proposal.

Table 3. The performance metrics for each class resulting from applying the proposal

	Accuracy	F1_Score	Recall	Precision
Dangerous Delegatecall	0.959	0.95	1.00	0.90
Integer Overflow		0.96	0.96	0.96
Reentrancy		0.96	0.95	0.97
Timestamp Dependency		0.96	0.96	0.95

It can be seen from Table 3 that:

- An overall accuracy of 95.9% indicates that the model correctly predicts the types of vulnerabilities in SC approximately 96% of the time.
- For the Dangerous Delegatecall vulnerability, a Precision value of 0.90 indicates that, of all samples predicted to contain a Dangerous Delegatecall vulnerability, 90% contained this vulnerability. A Recall value of 1 indicates that, of all samples that were Dangerous Delegatecall, the model correctly predicted them all 100% of the time. The high F1 score (0.95) implies that the model excels in balancing precision with recall.
- For the Integer Overflow vulnerability, a Precision value of 0.96 indicates that, of all samples predicted to contain an Integer Overflow vulnerability, 96% contained this vulnerability. A Recall value of 0.96 indicates that the model correctly predicted 96% of samples that were Integer Overflow category. The F1_score (0.96) shows that the model is very good at balancing precision with recall.
- For the Reentrancy vulnerability, a Precision value of 0.97 indicates that 97% of all samples which predicted as Reentrancy it was containing this vulnerability. A Recall value of 0.95 indicates that the model correctly predicted 95% of all samples that were Reentrancy category.
- For the Timestamp Dependency vulnerability, a Precision value of 0.95 indicates that, of all samples predicted as Timestamp Dependency, 95% of these contain this vulnerability. A Recall value of 0.96 indicates that, of all samples that were actually Reentrancy, the model correctly predicted 96%.

Therefore, it can be said that the proposed model shows exceptional performance in identifying all types of SC vulnerabilities. The high Recall values, especially for Dangerous Delegatecall, indicate that the model does not miss any cases containing this vulnerability, but the Precision value is slightly lower than others; therefore, improving this value would reduce false positives, which makes the model more accurate.

4.5. **Computational cost and model efficiency.** The proposed model architecture contains 12,703,428 trainable parameters (48.5 MB). The model architecture was chosen to balance contextual feature learning and model performance while maintaining acceptable complexity.

Training Details:

- Training time per epoch: ∼66 seconds on an NVIDIA RTX 3050 GPU
- Total training time: ∼11 minutes (for 10 epochs)

To evaluate the efficiency of our proposed model, we implemented a lightweight Multi-Layer Perceptron (MLP) model with ∼112K trainable parameters. The comparison is summarized in Table 4:

Table 4. Complexity comparison

Model	Parameters	Accuracy	Training Time	Notes
DL-SCVDetect	12.7M	95.9%	∼11 minutes	High accuracy
MLP	112K	87.4%	∼4 mins	Lightweight but lower accuracy

While the MLP model is computationally cheaper, it underperforms in classification accuracy. Our model's added computational cost is justified by its superior performance and generalizability.

5. **Conclusions and Future Works.** In this study, a deep learning approach was applied to detect vulnerabilities in SCs. The proposed approach involves using a Random Forest algorithm for binary classification to detect whether nodes contain vulnerabilities, achieving an accuracy of up to 94%. The Random Forest model follows a deep learning model based on multi-class classification to determine the type of vulnerability present. An embedding layer is used to transform the input sequences into fixed-size vectors. A Bi-LSTM layer is then used to concatenate the forward and backward sequences after each step. An attention layer is then added to help the model focus on the important parts of the input sequence. A Conv1D convolution layer is then added, applying a one-dimensional convolution to the output of the attention layer using 64 filters and a kernel size of 3 to capture local patterns in the data. A 1D Max pooling layer is then added to reduce the feature map. This is followed by two dense layers, the second of which produces predictions for four categories of vulnerabilities using a SoftMax activation function, whose output produces four probability values that determine the probability of belonging to each category.

The presented model demonstrates an overall accuracy of 95.9. The future efforts should focus on improving the accuracy of the Dangerous Delegate call through additional training data or feature engineering. Also, regular monitoring and validation of the model's performance on diverse datasets is recommended to ensure the model's robustness and reliability in different scenarios.

Although the proposed DL-SCVDetect framework shows promising results, several limitations should be addressed in future work:

- While the dataset covers four common types of vulnerabilities, it may not fully capture the diversity and complexity of smart contract code found in practical applications. Future work should consider larger and more diverse datasets to better generalize to different types of contracts.
- Although the model performs well on pre-defined vulnerabilities, it may struggle to detect new or emerging vulnerabilities that were not represented in the training set.

Future work should focus on developing the model to become more adaptive to new vulnerabilities through continuous learning.

- The deep learning model requires more computational resources, especially for large-scale contract analysis. The training time for the model could be a limiting factor when deployed in real-time systems. Efforts to optimize the model's architecture or utilize more efficient methods (e.g., transfer learning or pruning) would be beneficial for scalability. Furthermore, techniques like pruning, knowledge distillation, or quantization could be explored in future work to compress the proposed model for real-time use cases without significantly compromising accuracy.

REFERENCES

[1] M. Javaid, A. Haleem, R. Pratap Singh, S. Khan, and R. Suman, "Blockchain technology applications for Industry 4.0: A literature-based review," *Blockchain: Research and Applications*, vol. 2, no. 4, 2021, doi: 10.1016/j.bcra.2021.100027.

[2] M. Javaid, A. Haleem, R. P. Singh, R. Suman, and S. Khan, "A review of Blockchain Technology applications for financial services," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 3, 2022, doi: 10.1016/j.tbench.2022.100073.

[3] Z. Jian et al., "TSC-VEE: A TrustZone-Based Smart Contract Virtual Execution Environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1773–1788, Jun. 2023, doi: 10.1109/TPDS.2023.3263882.

[4] J. M. de A. Afonso, "Mechanisms for modeling and validation of smart contracts," MSc, NOVA University Lisbon, Almada, Portugal, 2023.

[5] A. Bakhshi, "Securing Smart Contracts: Strategies for Identifying and Mitigating Vulnerabilities in Blockchain Applications," University of Central Arkansas, Arkansas, USA, 2024.

[6] D. Kirli et al., "Smart contracts in energy systems: A systematic review of fundamental approaches and implementations," *Renewable and Sustainable Energy Reviews*, vol. 158, p. 112013, Apr. 2022, doi: 10.1016/j.rser.2021.112013.

[7] M. Hatami, Q. Qu, Y. Chen, H. Kholidy, E. Blasch, and E. Ardiles-Cruz, "A Survey of the Real-Time Metaverse: Challenges and Opportunities," *Future Internet*, vol. 16, no. 10, p. 379, Oct. 2024, doi: 10.3390/fi16100379.

[8] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive Mob Comput*, vol. 67, p. 101227, Sep. 2020, doi: 10.1016/j.pmcj.2020.101227.

[9] Y. Xu, T. Slaats, B. Düdder, T. Troels Hildebrandt, and T. Van Cutsem, "Safe design and evolution of smart contracts using dynamic condition response graphs to model generic role-based behaviors," *Journal of Software: Evolution and Process*, vol. 37, no. 1, Jan. 2025, doi: 10.1002/smr2730.

[10] A. Alshorman, F. Shannaq, and M. Shehab, "Machine learning approaches for enhancing smart contracts security: A systematic literature review," *International Journal of Data and Network Science*, vol. 8, no. 3, pp. 1349–1368, 2024, doi: 10.5267/j.ijdns.2024.4.007.

[11] N. Sharma and S. Sharma, "A Survey of Mythril, A Smart Contract Security Analysis Tool for EVM Bytecode," *Indian Journal of Natural Sciences*, vol. 13, no. 75, pp. 51003–51010, 2022. [Online]. Available: https://www.researchgate.net/publication/366391033

[12] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, May 2019, pp. 8–15. doi: 10.1109/WETSEB.2019.00008.

[13] J. J. Lohith and K. Singh, "Enhancing Oyente: four new vulnerability detections for improved smart contract security analysis," *International Journal of Information Technology (Singapore)*, vol. 16, no. 6, pp. 3389–3399, 2024, doi: 10.1007/s41870-024-01909-8.

[14] L. S. H. Colin, P. M. Mohan, J. Pan, and P. L. K. Keong, "An Integrated Smart Contract Vulnerability Detection Tool Using Multi-Layer Perceptron on Real-Time Solidity Smart Contracts," *IEEE Access*, vol. 12, pp. 23549–23567, 2024, doi: 10.1109/ACCESS.2024.3364351.

[15] Z. Wang, J. Chen, Y. Wang, Y. Zhang, W. Zhang, and Z. Zheng, "Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 161–181, Jul. 2024, doi: 10.1145/3643734.

[16] Z. Zhen, X. Zhao, J. Zhang, Y. Wang, and H. Chen, "DA-GNN: A smart contract vulnerability detection method based on Dual Attention Graph Neural Network," *Computer Networks*, vol. 242, p. 110238, Apr. 2024, doi: 10.1016/j.comnet.2024.110238.

[17] L. Zhang et al., “A Novel Smart Contract Reentrancy Vulnerability Detection Model based on BiGAS,” *J Signal Process Syst*, vol. 96, no. 3, pp. 215–237, 2024, doi: 10.1007/s11265-023-01859-7.

[18] H. Wu, H. Dong, Y. He, and Q. Duan, “Smart Contract Vulnerability Detection Based on Hybrid Attention Mechanism Model,” *Applied Sciences*, vol. 13, no. 2, p. 770, Jan. 2023, doi: 10.3390/app13020770.

[19] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, “ASSBert: Active and semi-supervised bert for smart contract vulnerability detection,” *Journal of Information Security and Applications*, vol. 73, no. January, p. 103423, 2023, doi: 10.1016/j.jisa.2023.103423.

[20] A. A. Hussein, A. M. Montaser, and H. A. Elsayed, “Classification of Spine Images using Hybrid Quantum Neural Network Classifier,” *Journal of Information Hiding and Multimedia Signal Processing*, vol. 15, no. 2, pp. 98–113, 2024. [Online]. Available: https://www.jihmsp.org/2024/vol15/N2/05.JIHMSP-240404%20(1).pdf

[21] W. Chan and A. Olmsted, “Ethereum transaction graph analysis,” in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, IEEE, Dec. 2017, pp. 498–500. doi: 10.23919/ICITST.2017.8356459.

[22] S. J. Johnson, M. R. Murty, and I. Navakanth, “A detailed review on word embedding techniques with emphasis on word2vec,” *Multimed Tools Appl*, vol. 83, no. 13, pp. 37979–38007, Oct. 2023, doi: 10.1007/s11042-023-17007-z.

[23] G. Van Houdt, C. Mosquera, and G. Nápoles, “A review on the long short-term memory model,” *Artif Intell Rev*, vol. 53, no. 8, pp. 5929–5955, 2020, doi: 10.1007/s10462-020-09838-1.

[24] S. Nosouhian, F. Nosouhian, and A. K. Khoshouei, “A review of recurrent neural network architecture for sequence learning: Comparison between LSTM and GRU,” 2021. [Online]. Available: https://www.preprints.org/manuscript/202107.0252/v1%0Ahttps://www.preprints.org/manuscript/202107.0252

[25] Z. Niu, G. Zhong, and H. Yu, “A review on the attention mechanism of deep learning,” *Neurocomputing*, vol. 452, pp. 48–62, 2021, doi: 10.1016/j.neucom.2021.03.091.

[26] G. Liu and J. Guo, “Bidirectional LSTM with attention mechanism and convolutional layer for text classification,” *Neurocomputing*, vol. 337, pp. 325–338, 2019, doi: 10.1016/j.neucom.2019.01.078.

[27] M. K, A. Ramesh, R. G, S. Prem, R. A A, and D. M. P. Gopinath, “1D Convolution approach to human activity recognition using sensor data and comparison with machine learning algorithms,” *International Journal of Cognitive Computing in Engineering*, vol. 2, pp. 130–143, 2021, doi: 10.1016/j.ijcce.2021.09.001.

[28] A. Zafar et al., “A Comparison of Pooling Methods for Convolutional Neural Networks,” *Applied Sciences (Switzerland)*, vol. 12, no. 17, p. 8643, 2022, doi: 10.3390/app12178643.

[29] X. He, Z. Zhou, and L. Thiele, “Multi-task zipping via layer-wise neuron sharing ,” *Adv Neural Inf Process Syst*,vol. 2018-Decem, no., pp. 6016–6026, 2018.

[30] B. Krawczyk, “Learning from imbalanced data: open challenges and future directions,” *Progress in Artificial Intelligence*, vol. 5, no. 4, pp. 221–232, Nov. 2016, doi: 10.1007/s13748-016-0094-0.

[31] D. Elreedy, A. F. Atiya, and F. Kamalov, “A theoretical distribution analysis of synthetic minority oversampling technique (SMOTE) for imbalanced learning,” *Mach Learn*, vol. 113, no. 7, pp. 4903–4923, Jul. 2024, doi: 10.1007/s10994-022-06296-4.