

SCADE: a deep learning ensemble for semantic flow analysis in smart contract vulnerability detection

Muralidhara Srirama¹, Usha Banavikal Ajay²

¹Department of Information Science and Engineering, BMS Institute of Technology and Management,
Affiliated to Visveswaraya Technological University, Belagavi, India

²Department of Computer Science and Engineering, BMS Institute of Technology and Management,
Affiliated to Visveswaraya Technological University, Belagavi, India

Article Info

Article history:

Received Nov 16, 2024

Revised Sep 9, 2025

Accepted Oct 15, 2025

Keywords:

Internet of things

Reliability

Security

Smart contract

Vulnerabilities

ABSTRACT

A vulnerability in smart contracts refers to weaknesses in the code that can be exploited by attackers, leading to security breaches and unintended behavior. With the growing use of smart contracts in decentralized blockchain systems, particularly in internet of things (IoT) environments, ensuring their security has become increasingly critical. Traditional vulnerability detection techniques, such as formal verification and symbolic execution, face significant limitations, including high rates of false positives and negatives, scalability issues, and difficulty in detecting complex vulnerabilities. To address these challenges, this paper proposes semantic contract flow analysis and deep learning ensemble (SCADE) for smart contract vulnerability detection. SCADE leverages semantic flow analysis combined with an ensemble of deep learning models, including convolutional neural networks (CNN), bidirectional sequence encoder (BSE), layered probabilistic neural network (LPNN), and adaptive context learning network (ACLN), to detect vulnerabilities effectively. The methodology breaks down the smart contract code into structured components through a contract structure mapper, followed by extracting semantic paths and converting them into sequential vector representations. These representations are then processed through a deep learning ensemble to identify potential vulnerabilities such as reentrancy, timestamp dependency, code injection, and hardcoded gas amounts.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Muralidhara Srirama

Department of Information Science and Engineering, BMS Institute of Technology and Management

Affiliated to Visveswaraya Technological University

Belagavi, India

Email: muralidhars_12@rediffmail.com

1. INTRODUCTION

Programs that can run on blockchain networks are known as smart contracts. Smart contracts' functionality and appearance are easily customizable to satisfy certain company needs. Like traditional apps, smart contracts go through a lifetime that includes development, deployment, execution, and completion. Smart contracts are being used quickly in several areas, including gambling, supply chains, voting, and the Internet of Things. They are vulnerable to security flaws because they operate in a decentralized network setting and frequently include financial assets. The security of blockchain platforms is seriously threatened by the increasing frequency of smart contract assaults, especially considering the quick growth of smart contracts and their indispensable function in the administration of priceless digital assets. Smart contracts

extend the use of blockchain technology into new industries, like banking, healthcare, and the Internet of Things, while also enhancing the decentralized features of blockchain platforms [1], [2].

There are two different methods for finding security holes in smart contracts. Traditional methods including formal verification, fuzzing, symbolic execution, and taint analysis are used in the first method of finding vulnerabilities in smart contracts. These technologies are more efficient than human audits, but they frequently have high false positive and false negative rates, and some systems might not be as good at detecting things as they could be. Most of these studies concentrate on specific vulnerability patterns and just assess keywords based on their surface features; for example, timestamp vulnerabilities are found by merely determining if the keywords contain the block timestamp. In one field of study, deep learning models are used to investigate vulnerabilities by means of feature extraction and model training. Even though these techniques have demonstrated increased accuracy, they frequently display a lack of interpretability. The training data sets have a major impact on the performance and quality of these models, which may limit their capacity to generalize to previously unknown vulnerabilities [3].

Deep learning and machine learning are widely used today and have made great strides in many different fields, despite their current limits. Both approaches require large amounts of data to train in order to accomplish the desired result; however, they frequently do not produce acceptable results in real-world scenarios with small sample numbers or annotated data [4], [5]. However, in the field of Ethereum smart contract vulnerability identification, the sample size of real-world vulnerable contracts is still small, which presents difficulties for security firms trying to quickly gather enough vulnerability samples. For this reason, it is crucial to investigate Ethereum smart contract vulnerabilities in the context of small sample sizes. The aim of this study is to detect weaknesses in ethereum smart contracts (ESC) and minimize the monetary damage linked to these agreements. The application of machine learning and deep learning techniques has significantly advanced the field of ethernet smart contract vulnerabilities, leading to increased processing speed and improved detection accuracy. The scarcity of real-world examples of Ethereum smart contract vulnerabilities makes it difficult for security companies to collect a large enough sample of these issues in a timely manner. As such, investigating small-sample learning techniques is an important first step in solving the issue and getting over other obstacles in the field of smart contract vulnerability detection [6].

An efficient vulnerability detection tool is offered to improve the security of smart contracts. The paper includes a method for graph extraction as well as a thorough process for identifying vulnerabilities. Graph extraction is the process of creating graphs and identifying patterns of vulnerability. To demonstrate susceptibility. To find sample SCGraphs from the data set, the SCGraph libraries first apply the approximation graph matching approach as the first stage in the vulnerability identification process. To find out if the contract has any defects, compute the degree of similarity between the SCGraphs created from the contracts that are being examined and the SCGraphs kept in the vulnerability library [7].

To improve detection capabilities and solve the shortcomings of current approaches, this research provides a phased methodology for identifying vulnerabilities in smart contracts. The suggested method makes use of graph neural networks and deep learning to utilize expert patterns [8]. The purpose of the second step of the checking mechanism is to provide error reports for further submission and to block contract transactions at the ethernet virtual machine (EVM) level that include potentially destructive operations. When using ContractWard to identify vulnerabilities in smart contracts, it is advised to use machine learning techniques. Bigram characteristics are initially retrieved from reduced smart contract operation codes. To create the models, we use five machine learning algorithms in addition to two sampling strategies. 49,502 actual Ethereum smart contracts are used to assess ContractWard [9].

A vulnerability detection tool is provided to enhance the security of smart contracts. The document presents a methodology for graph extraction alongside a comprehensive approach to vulnerability detection. The graph extraction technique involves the creation of a graph and the identification of vulnerability patterns. In the vulnerability detection process, the approximation graph matching method is utilized to identify representative SCGraphs from the dataset. This step is essential for the subsequent development of vulnerability SCGraph libraries. To determine the presence of defects in the contract, calculate the degree of similarity between the SCGraphs generated from the contracts under analysis and the SCGraphs stored in the vulnerability library [10]. The increasing adoption of smart contracts in sectors like finance, healthcare, and IoT has exposed significant security challenges, as these contracts are often vulnerable to attacks in decentralized networks. Traditional vulnerability detection methods, such as formal verification and symbolic execution, are inadequate due to high false positive/negative rates and inability to handle complex contract structures:

- a) Novel methodology for vulnerability detection: This paper introduces SCADE, a new methodology that combines semantic flow analysis with an ensemble of deep learning models to effectively detect vulnerabilities in smart contracts.
- b) Deep learning ensemble for enhanced detection: The proposed approach leverages a diverse ensemble of models, including convolutional neural networks (CNN), bidirectional sequence encoder (BSE),

layered probabilistic neural network (LPNN), and adaptive context learning network (ACLN), to improve the accuracy and robustness of vulnerability detection.

- c) Contract structure mapper for code analysis: SCADE utilizes a Contract Structure Mapper to break down smart contract code into structured components, enabling detailed semantic analysis and facilitating the detection of complex vulnerabilities.
- d) Fine-grained vulnerability identification: The methodology ensures fine-grained detection, providing specific insights into vulnerable segments of the contract code, which helps in pinpointing and addressing security issues effectively.
- e) Extensive evaluation on real-world datasets: The effectiveness of SCADE is demonstrated through comprehensive evaluations on real-world Ethereum smart contract datasets, showing its superiority over traditional methods in terms of accuracy and scalability.

2. RELATED WORK

The goal of finding vulnerabilities in smart contracts is accomplished by applying a deep learning approach. To train the deep learning model, the suggested technique entails detecting the unique features of vulnerabilities present in different smart contracts. The application of a deep learning technique enables the identification of vulnerabilities within smart contracts. The proposed methodology involves identifying the unique attributes of vulnerabilities present in different smart contracts to enhance the training efficiency of the deep learning model [11]. Recent developments in machine learning have led to the formulation of various algorithms aimed at identifying vulnerabilities within smart contracts. The prevalent methods utilized in this context include graph neural networks and natural language processing. The subsequent methods are employed to extract the characteristics of vulnerabilities from contracts and subsequently identify these issues. Considering the contract code as a language facilitates the analysis of its syntactic and semantic relationships, as well as control flow and data flow dependencies. Establishing specific work objectives and utilizing appropriate learning models enables the efficient detection and resolution of vulnerabilities and code similarities. A parser is utilized to construct a hierarchical code tree (HCT) for the identification of SmartEmbed vulnerabilities [12]. Each node in the HCT represents a distinct syntactic unit of the contract code. The nodes contain extensive information regarding the characteristics of the source code. The EVM bytecode undergoes analysis through Eth2Vec [13], which considers the distinctions between programming language and natural language. As a result of the analysis, multiple layers of JSON files are generated, which are subsequently utilized as input. The implementation of natural language processing in smart contracts is challenging due to the reliance on additional dependencies and the complexity of vocabulary sources that are difficult to obtain in the current environment [14]. The identified issues can be addressed through the implementation of source code abstraction, as depicted in a graphical representation. The generated graph enables the neural network to execute vulnerability detection tasks tailored to the characteristics of the programming language. This technique has the potential to optimize data flow collection and improve the management of dependent links. Instances of bidirectional graph neural networks utilized for the extraction of graph properties include BGNN4VD [15]. The graph convolutional network (GCN) serves as a widely utilized approach within the domain of graph embedding learning. Graph encoders improve the representation of graphs by effectively encoding the underlying graph structure along with node attribute data. The machine learning-based approach provides enhanced characterization of source code vulnerabilities when compared to traditional vulnerability detection methods, utilizing model training and improved generalization capabilities. The program demonstrates significant effectiveness in addressing complex logical reasoning challenges.

ReDefender is an application that has been developed recently, introducing a new approach to utilizing fuzz testing for the identification of reentrancy issues. The method consists of three primary phases. Initially, the source code of a contract requires preprocessing to generate a candidate pool suitable for fuzzing, along with a dependency graph that facilitates automatic contract deployment. The objective of fuzzing input creation is to produce transactions that are transmitted to an agent contract to initiate an attack. During each execution, runtime information is collected and documented in the execution log. Finally, vulnerability verification involves analyzing the execution log to determine the presence of a reentrancy process and to evaluate its potential risk [16].

To effectively identify vulnerabilities in smart contracts, developed a machine learning method known as SaferSC in 2018 [17]. The system employs a machine learning approach known as long short-term memory (LSTM). Durieux *et al.* [18] developed the TMP and DR-GCN technologies for smart contract detection. Three distinct vulnerabilities associated with Ethereum and the Viterbi chain have been identified in smart contracts. The system exhibits several limitations, including a dependency on timestamps related to reentrancy, the potential for infinite loops within graph neural networks, and additional unidentified issues.

The DR-GCN detection tool employs the smart contract graph to effectively depict the semantic structure and syntax of the smart contract function. The detection of vulnerabilities in smart contracts employs a dimensionless graph convolutional neural network. The temporal information propagation network (TMP) is utilized for the detection of vulnerabilities in smart contracts. The objective is achieved through the construction of a smart contract graph that accurately represents the syntax and semantic structure of the smart contract function. In the same year, Tikhomirov *et al.* [19] developed ContractWard, a software application aimed at identifying security vulnerabilities in smart contracts. This tool enables users to extract a segment of the reduced operational code from a binary-syntax smart contract. The second stage involves constructing models to identify vulnerabilities through various sampling and machine learning methodologies. The AME smart contract vulnerability detection tool was developed [20]. The application features an expert mode functionality and employs deep learning algorithms.

The provided array `<code>[]</code>` is devoid of elements. Tags are components that incorporate human input. This article will discuss three primary contributions. The subsequent methods are recommended to improve the technical components of the project: The smart contract weakness classification (SWC) registry aims to create a standardized methodology for identifying vulnerabilities. The hierarchical code tree (AST) approach is specifically engineered to intentionally induce faults, facilitating the generation of a diverse array of faulty files compatible with various Solidity versions. The pre-processing technique utilized for the smart contract training data [21] will undergo standardization.

This work aims to identify vulnerabilities in smart contracts through the application of multimodal fusion learning techniques. The main goal of this strategy is to enhance scalability through a focus on multimodal components while reducing the requirement for specialized knowledge. Three distinct datasets are utilized for training the models EfficientNet, BiLSTM, and Transformer: source code sequences pertaining to smart contracts, statistical characteristics of opcode bag frequency, and grayscale image quality [22]. This study aims to introduce a new method for the rapid and precise identification and analysis of smart contracts. The proposed method utilizes the BERT pre-training model to efficiently detect vulnerabilities in smart contracts. The preprocessing stage involves contract analysis and symbol modification to enhance the pre-training model's ability to extract contract attributes [23].

The TechyTech tool employs a distinctive dynamic analysis technique known as involuntary transfer to identify vulnerabilities related to tx.origin and reentrancy. The future identification of the two vulnerabilities and their respective versions will be accomplished through a tree-based classification string. This work addresses various software engineering issues, including the deployed owner, hijacked stacks, and the inability to provide transaction receipts for reentrant calls [24]-[29].

3. PROPOSED METHOD

The proposed model aims at focusing on the detection of vulnerabilities as well as the training procedure of the model. We observe in the training process, embedded code as well as semantic learning is utilized to conclude the training process of the model that aims at detection of vulnerabilities. During vulnerability evaluation, the focus is to finish coding at the fine-tuned phase used in detection. This is seen in the Figure 1. The focus lies on the development of a fine-tuned sample as well as the utilization of various neural networks to meet the requirement of ideal performance. Initially, a large amount of data is gathered of the vulnerabilities that are present in the real-world of the contract codes which are built by the semantic path retriever as well as the totally extracted paths from graphic function calls. Also, these are codes that are used to transform them as a contract structure that is termed as contract structure builder. During this procedure, we utilize a technique for the conversion of source codes of the smart contract into more refined contract structure flow segments by learning the structural information of the contract structure as well as the semantic paths built by the function calls represented graphically. Although some data that is unrelated to vulnerability is neglected. Furthermore, these contract structure flow segments are utilized as input in the next phase that is: sequential vector generator post normalization as well as labelling. Different techniques are adapted for retrieval of attribute vectors for tokens of word flow for building the embedded code to learn data. This data includes attributes contextually as well as based on information flow. In conclusion, during the detection phase of vulnerability, we develop a range of neural networks to train the vectors as well as utilized in detecting vulnerabilities. These detections are expressed in a detailed manner implying that we recognize the area that vulnerability is possible. Figure 1 shows the proposed architecture.

While we consider examining vulnerability, the contract check calls the semantic path retriever as well as the contract structure builder. One separate code of contract is split up into contract structure flow segments sets. These tokens of word flow have semantic data relating to vulnerability. The model proposed in this paper utilizes one-on-one word table of vectors for transforming the relating contract structure flow segments as vectors digitally as well as the outputs resulting in a predicted score of vulnerability of contract

structure flow segments via a classification method using neural networks. These contract structure flow components express the code having refined elements implying the report of vulnerability to be in detail.

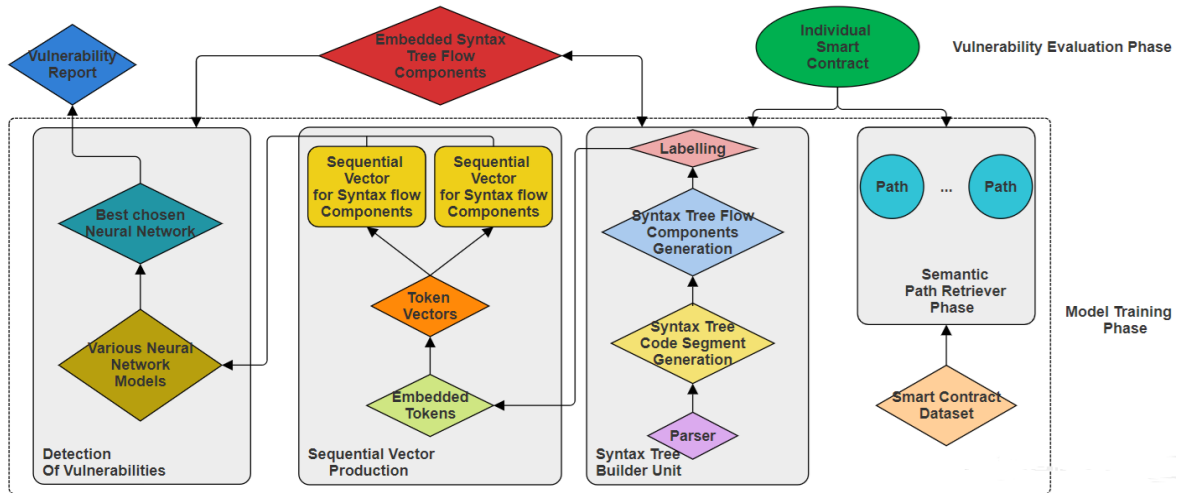


Figure 1. The architecture of the proposed model

3.1. Semantic path retriever phase

Here, we have a program being expressed as S that has a set of functions that are defined represented as i_1, \dots, i_ω , where $S = \{i_1, \dots, i_\omega\}$. A function i_l , in which 1 is lesser than or equal to ω , is an ordered sequence set of $v_{l,1}, \dots, v_{l,n}$ denoted as $i_l = \{v_{l,1}, \dots, v_{l,n}\}$. A statement $v_{l,m}$ where 1 is lesser than or equal to n and 1 is lesser than or equal to m is a sample of tokens that are in order $w_{l,m,1}, \dots, w_{l,m,\mu}$. These tokens are the smallest unit of the program that utilized as an operator, identifier, a key denotation and a constant, is retrieved being lexically analyzed.

While observing graphical call, a program $S = \{i_1, \dots, i_\omega\}$ is considered, the graphical call for S is represented as $J = (Y, H)$, in which $Y = \{q_1, \dots, q_u\}$ expresses a sample of nodes in which every node is used to depict a function i_l belongs to S and $H = \{h_1, \dots, h_y\}$ denotes direct edges in which every edge shows the control flow that happen possibility within two nodes. Consider program $S = \{i_1, \dots, i_\omega\}$, in which $i_l = \{v_{l,1}, \dots, v_{l,n}\}$ with $J = (Y, H)$. The graphic function call J is also denoted as the relational set of semantic paths, expressed as $J = \{P_1, \dots, P_\alpha\}$, in which the path P_α (1 less or equal to α) is a path given as $W_1 U_1 \Rightarrow W_2 U_2 \Rightarrow \dots U_s \Rightarrow W_{t+1}$.

3.2. Contract structure mapper

We notice from prior traditional techniques; the written source code is as solidity that could parse a contract structure mapper as well as conserve the structure correlations of the code. While another traditional study shows that parser generators are utilized grammatically for building a solidity parser for smart contract parsing as a contract structure. On merging the nodes that exclude roots as well as leaves of the contract structure and the semantic paths are collected by extraction unit, we introduce the contract structure mapper based code segments being used that expressed in detail in algorithm 1. Furthermore, this is split into refined detailed tokens utilized for unit flow of the contract structure flow units using the structure of an abstract structure that is described in Algorithm 2.

The data structure of a structure is utilized for a contract structure. In which, the root shows that the structure is free contextually, the leaves depict the terminal structure being free from contextual information. Initially, a program $S = \{i_1, \dots, i_\omega\}$, in which $i_l = \{v_{l,1}, \dots, v_{l,n}\}$ along with $v_{l,1} = \{w_{l,m,1}, \dots, w_{l,m,\gamma}\}$. specifically, an Contract Structure Mapper W is developed using S program. The fragment of code denoted as $I_{l,m,x,y}$ having one or more than one tokens being consecutive for a statement $v_{l,m}$ such that $I_{l,m,x,y} = (w_{l,m,x}, \dots, w_{l,m,y})$ in which 1 is lesser than or equal to x is lesser than or equal to y is lesser than or equal to γ . $w_{l,m,x}, \dots, w_{l,m,y}$ shows a sub-structure for W in which the elements of the leaf of structure having the root element as the non-leaf as well as non-root element original to contract structure W expressed as the contract structure based code segments. The contract structure based code segments which is proposed is described in the Algorithm 1.

Algorithm 1. Designing contract code segments

Input program $S=\{i_1, \dots, i_\omega\}$, Contract Structure W and graphic function call $J=\{P_1, \dots, P_\alpha\}$ generated by S .
Output Set F of Contract Structure Mapper code segments in program S

```

Step 1  $F \leftarrow \text{null}$ 
Step 2 A sample set utilized in storing root units built by algorithm expressed as  $TQ \leftarrow \text{null}$ 
Step 3 For every  $P_a$  belongs to  $J$  do
Step 4     For every  $w_{(a,b)}$  in  $P_a$  do
Step 5         For every  $i_N$  belongs to  $S$  do
Step 6             If  $w_{(a,b)}$  is equivalent to  $i_N$  then
Step 7                 The FuncDef node that relates to  $i_N$  is utilized as root
element  $\llbracket tq \rrbracket_\varphi$ ;
Step 8                                      $TQ \leftarrow TQ \cup \{\llbracket tq \rrbracket_\varphi\}$ 
Step 9             End if
Step 10        End for
Step 11    End for
Step 12 End for
Step 13 For every  $\llbracket tq \rrbracket_\varphi$  belongs to  $TQ$  do
Step 14     Generation of a sub-Structure  $Wv_\rho$  with  $\llbracket tq \rrbracket_\alpha$  as root unit node for  $W$ ;
Step 15      $F \leftarrow F \cup \{\llbracket tq \rrbracket_\alpha; [\text{leaf nodes of sub-tree } Wv_\rho]\}$ 
Step 16 End for
Step 17 Return the obtained result  $F$ ;

```

While considering a smart contract that has large amounts of code, the proposed system including contract structure mapper based code segments ensures the detection of vulnerabilities. The contract structure code segments are obtained because of the algorithm 1. The component flow of the contract structure is independent in relation to the vulnerabilities being detected using the code. While it is compared to leaf units used in processing of the initial structure, the contract structure code segment method can withhold contextual information having attributes that have a relation to vulnerability.

Firstly, a program $S = \{i_1, \dots, i_\omega\}$, in which $i_l = \{v_{l,1}, \dots, v_{l,\theta}\}$ along with $v_{l,1} = \{w_{l,m,1}, \dots, w_{l,m,\gamma}\}$. Where program S generates W as an contract structure mapper. We assume a sample set of leaf elements $OQ = \{oq_1, \dots, oq_\tau\}$, if these elements oq_1, \dots, oq_τ having a common root unit, then the leaf units combined to showcase the entire information that is expressed as contract structure flow components. The extraction of these components is described in Algorithm 2.

Algorithm 2. Extraction of contract execution elements from contract code segments

Input Consider program $S=\{i_1, \dots, i_\omega\}$, Contract Structure W introduced by S , Syntax Tree code segments $=\{tq, [w_1, \dots, w_2]\}$ resulting from algorithm 1 and a node set $C=\{c_1, \dots, c_\sigma\}$ having all the required types of nodes
Output Contract Execution Elements shown by a set X in a Contract Structure based code segments

```

Step 1 Function PriorTraverse(Structure U, node element P)
Step 2     If  $P == \text{null}$  then
Step 3         Return null
Step 4     End if
Step 5     SubStructure  $\leftarrow$  NewTree(node.value)
Step 6     For child, Structure.Children do (PriorOrder(Structure, child))
Step 7         Append subStructure.children
Step 8     End for
Step 9     Return subStructure
Step 10 End function
Step 11
Step 12  $X \leftarrow \text{null}$ 
Step 13 SubStructure shows Contract Structure Flow Components  $\leftarrow$  PriorOrder ( $W, tq$ );
Step 14 For every  $C_u$  belongs to  $C=\{c_1, \dots, c_\sigma\}$  do
Step 15     SubStructure  $\llbracket vw \rrbracket_u \leftarrow$  PriorOrder( $VW, c_u$ )
Step 16  $X \leftarrow X \cup (\text{leaf nodes of subtree } \llbracket vw \rrbracket_u)$ ;
Step 17 End for
Step 18 Return  $X$ 

```

3.3. Code vectorization module

Considering the utilization of neural network technique for the leaning purpose, an individual stream of sequential vectors is designed by mapping the contract structure flow component tokens as a vector space. This mapping for every individual token is done to a vector having constant size. Embedding algorithms, in this case three of them are implemented for the purpose of attaining the embedded vectors, these algorithms include contextual word embedding model (CWEM), fast word embedding generator (FWEG), contextualized word representation model (CWRM) and BERT. The vector prediction expression for words

in sentences is performed using the CWEM algorithm. Although, the word structure is not studied morphologically. The resolution of this issue is obtained by the fast text algorithm through two various kinds of attributes and separate individual words being embedded, represented as n-gram as well as n-char attributes. Both attributes are stored by hashing scores. The prior trained models include CWRM and BERT algorithms. A huge amount of information that is unlabeled as well as learning of attribute abstraction in the language is seen to be beneficial. Contextual data is utilized by both to produce word expressions. The BERT algorithm is proved to manage longer sequences. The generation of various expressions for very word is performed by the CWRM resulting in making the expression of the word better fine-tuned. Theoretically, it is observed that BERT as well as CWRM are proven to show better results. While considering the speed of embedding for words, the proposed study still utilizes FastText as the preferable method.

Assume a contract structure flow component generated using algorithm 2, for every individual token $w_{l,m,n}$ that belongs to contract structure flow component $= \{w_{l,m,1}, \dots, w_{l,m,\gamma}\}$ in which 1 is lesser than or equal to γ is lesser than or equal to γ . Hence, the sequence vector is expressed for contract structure flow component as given in the (1). Once the learning for embedding token vector is complete for a particular contract structure flow component, every component that is generated. The vector token flow is combined to result in the sequential vector.

$$Vector(Contract\ Structure\ flow\ component) = \sum_{n=1}^{\gamma} w_{l,m,n} \quad (1)$$

3.4. Deep ensemble network

While we consider the generated sequential vectors, various techniques of neural networks are implemented to resolve the challenge of detecting vulnerabilities. The major focus lies in providing a detailed phase of detecting vulnerabilities. The prior techniques of neural networks only account to vulnerabilities in the contract phase. The proposed study gives line- phase labels for every contract structure mapper flow component that is produced. Contract structure mapper flow components are recognized to possess vulnerabilities. These labels present in the contract structure mapper flow components aid in locations relating to areas in the code, permitting detection of vulnerabilities at the detailed phase of the source code in smart contract. The techniques of neural networks that utilize here are CNN utilize various layers of convolution in order to retrieve local attributes, which is further followed by completely linked layers used for classification or even for regression. BSE, bidirectional sequence gate (BSG) one being used for forward propagation and the other for backward propagation. This model has the capacity to process sequential information. LPNN: This network architecture consists of various layers in which there exists a concealed layer in every layer and these concealed layers are linked by functions that are non-linear. Random forest: This technique has resulted in positive results while considering classification as well as detection of vulnerabilities. Various structures can be trained through various samples as well as attributes to decrease the model variance and enhance the ability in generalizing the model. ACLN: This model is accountable for sequential tasks. It uses attention schemes to grasp long-ranging dependencies existing in between the sequences.

4. RESULTS AND DISCUSSION

The SCADE methodology is composed of several key stages to effectively detect vulnerabilities in smart contracts. The process begins with the contract structure mapper, which is responsible for breaking down the smart contract code into structured components. This mapping allows for the extraction of critical semantic relationships and provides a basis for subsequent analysis. The semantic path retriever then extracts meaningful paths from the contract structure, capturing relationships between different code components. These paths are converted into sequential vector representations through a code vectorization module, which applies embedding algorithms to capture contextual information at the token level. Embedding algorithms such as CWEM, FWEg, and CWRM are used to generate rich vector representations of the code components. Once the code has been vectorized, a deep learning ensemble is employed to process the sequential vectors and detect potential vulnerabilities. The ensemble includes models such as CNN for local feature extraction, BSE for capturing contextual information in both directions, LPNN for probabilistic modeling, and ACLN for long-range dependency learning. These models work collaboratively to provide a robust and comprehensive analysis of the smart contract code. The final stage involves vulnerability classification and detection, where the output from the deep learning ensemble is used to classify vulnerabilities, including reentrancy, timestamp dependency, code injection, and hardcoded gas amounts.

4.1. Dataset details

The 307,396 functions from 40,932 smart contracts are included in the (ESC) dataset. The 5,013 functions in this dataset include at least one call statement to call. Value, making them potentially vulnerable

to reentrant vulnerabilities. There are 4,833 routines with BLOCK.TIMESTAMP statements that could result in a timestamp dependence issue. The 6,896 routines employ the DELEGATECALL command at least once, indicating the presence of a code injection vulnerability.

4.2. Vulnerabilities evaluated

- Reentrancy: re-entrancy occurs when a smart contract calls an external contract and continues its execution before the external call completes, allowing attackers to exploit this by repeatedly calling the vulnerable function, manipulating the contract's state or balance.
- Timestamp dependency: Timestamp dependency refers to using block timestamps in smart contracts, which can be manipulated by miners. This vulnerability allows attackers to influence time-based logic, such as triggering specific actions at manipulated block times, potentially leading to unpredictable outcomes.
- Code injection: code injection happens when an attacker is able to insert malicious code into a smart contract, often through unsafe usage of commands like DELEGATECALL. This allows the attacker to modify the behavior of the contract, compromising its logic and security.
- Call with hardcoded gas amount: this vulnerability arises when a smart contract uses a fixed gas amount for external calls. If the specified gas is insufficient for the external function to execute fully, it can lead to failed operations or incomplete execution, creating potential points for exploitation.

4.3. Comparison with other network algorithms

We compare PS to various network algorithms for contract code reentrancy, timestamp dependency, and malicious code injection. This compares various methodologies for each vulnerability type, with acc, precision, recall, and F1 score as the primary metrics. Before understanding these metrics, we need to explain TP, FP, TN, and FN.

- True positive (TP): a correct positive example, where an instance is a positive class and is also determined to be a positive class.
- False positive (FP): a wrong positive example, false alarm, originally a false class but judged as the positive class.
- True negative (TN): for the correct counterexample, an instance is a false class and is also determined to be a false class.
- False negative (FN): a wrong counterexample, omission, positive class but decided as a false class.
- $Acc = (TP + TN)/(TP + TN + FP + FN)$ counts how many of all predicted results are predicted correctly,
- $Precision = TP/(TP + FP)$ counts how many of all predicted results are positive and correct, which means how many are true positives,
- $Recall = TP/(TP + FN)$ counting the number of correct predictions among all the actual categories that are positive,
- $F1 = (2 \times Precision \times Recall)/(Precision + Recall)$ is the summed mean value of precision and recall.

4.4. Results

The analysis for Reentrancy across different models (Vanilla-RNN, LSTM, GRU, GCN) and (DM [ES] and PS) highlights significant performance differences. GCN consistently outperforms Vanilla-RNN, LSTM, and GRU in all metrics—accuracy, precision, recall, and F1 score—indicating its superior capability for detecting reentrancy vulnerabilities. While the traditional models such as Vanilla-RNN, LSTM, and GRU show gradual improvement, GCN demonstrates a considerable leap in performance. Furthermore, DM [ES] and PS, likely to represent advanced detection mechanisms or post-processing steps, show even better results, achieving the highest accuracy (91.87%), precision (89.98%), recall (88.54%), and F1 score (89.65%). These enhancements suggest that employing post-processing techniques or ensemble methods can significantly refine vulnerability detection, reducing false positives and false negatives. Overall, GCN is a robust model for this task, but DM [ES] and PS offer even more reliable and accurate detection through advanced methodologies. Table 1 and Figure 2 shows the comparison of the Reentrancy metric.

Table 1. Comparison table of the Reentrancy metric

Reentrancy	Vanilla-RNN [25]	LSTM [26]	GRU [27]	GCN [28]	DM [ES] [29]	PS
Acc (%)	49.64	53.68	54.54	77.85	89.74	91.87
Precision (%)	49.82	51.65	53.1	70.02	85.35	89.98
Recall (%)	58.78	67.82	71.3	78.79	86.19	88.54
F1(%)	50.71	58.64	60.87	74.15	85.76	89.65

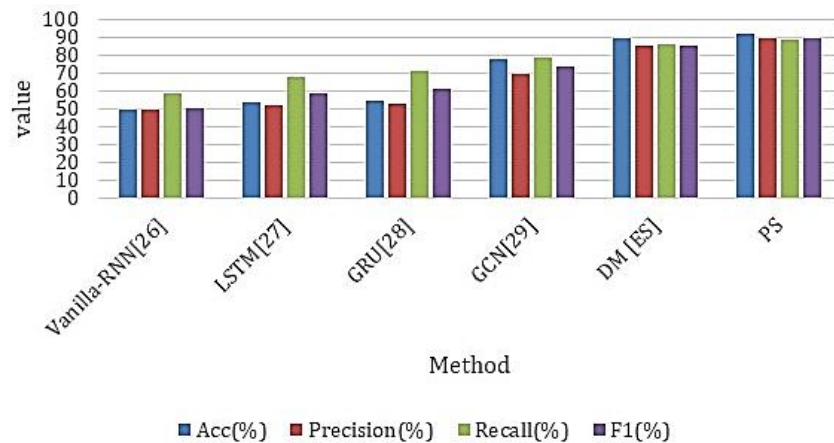


Figure 2. Reentrancy metric comparison with the proposed model

The analysis of the performance metrics for timestamp dependency across various models (Vanilla-RNN, LSTM, GRU, GCN) and (DM [ES], PS) reveals significant differences in their ability to detect timestamp vulnerabilities. GCN emerges as the top-performing machine learning model, achieving higher accuracy (74.21%), recall (75.97%), and F1 score (71.96%) compared to Vanilla-RNN, LSTM, and GRU, which show relatively modest improvements. While GCN delivers notable precision (68.35%) and is generally more balanced in terms of recall and F1 score, DM [ES] and PS metrics provide a significant boost across all measures, highlighting the importance of advanced post-processing techniques or ensemble detection methods. DM [ES] reaches 88.52% in accuracy and 84.1% in F1 score, while PS achieves the highest overall performance with 91.56% accuracy, 88.52% recall, and 88.09% F1 score. These results indicate that, while GCN is a robust model for detecting timestamp vulnerabilities, the use of ensemble techniques (DM [ES], PS) further enhances performance, especially in balancing precision and recall, making PS the most reliable in terms of overall detection effectiveness. Table 2 and Figure 3 shows the comparison with Timestamp dependency.

Table 2. Comparison table with Timestamp dependency

Timestamp dependency	Vanilla-RNN [25]	LSTM [26]	GRU [27]	GCN [28]	DM [ES] [29]	PS
Acc(%)	49.77	50.79	52.06	74.21	88.52	91.56
Precision(%)	51.91	50.32	49.41	68.35	82.07	85.65
Recall(%)	44.59	59.23	59.91	75.97	86.23	88.52
F1(%)	45.62	54.41	54.15	71.96	84.1	88.09

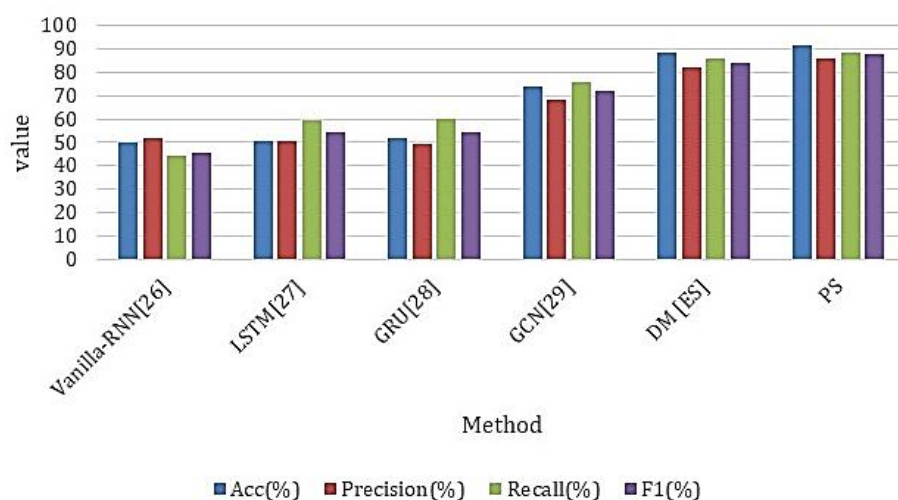


Figure 3. Timestamp dependency metric comparison with the proposed model

The analysis of the models' performance for detecting code injection vulnerabilities across Vanilla-RNN, LSTM, GRU, GCN, DM [ES], and PS shows clear differences in their effectiveness. GCN once again proves to be a superior machine learning model, with significantly higher accuracy (72.98%), recall (76.84%), and F1 score (73.16%) compared to the Vanilla-RNN, LSTM, and GRU models. Although the LSTM and GRU models show improvements over Vanilla-RNN, their precision and recall are still lower than that of GCN. On the other hand, DM [ES] and PS continue to demonstrate the highest performance across all metrics, with PS achieving the best results in accuracy (91.36%), recall (89.64%), and F1 score (87.32%). DM [ES] also performs strongly, especially in recall (87.57%) and precision (83.69%), indicating its ability to minimize false negatives while maintaining a good precision-recall balance. Overall, while GCN provides robust detection capabilities for code injection, advanced techniques such as DM [ES] and PS further improve the precision, recall, and F1 score, making PS the most effective model with a well-balanced performance across all metrics. Table 3 and Figure 4 shows the comparison of call with code injection.

Table 3. Comparison table of call with code injection

Code injection	Vanilla-RNN [25]	LSTM [26]	GRU [27]	GCN [28]	DM [ES] [29]	PS
Acc(%)	49.12	51.98	53.74	72.98	88.62	91.36
Precision(%)	42.64	50.64	52.01	69.82	83.69	85.76
Recall(%)	47.55	63.47	61.64	76.84	87.57	89.64
F1(%)	44.96	56.33	56.41	73.16	85.58	87.32

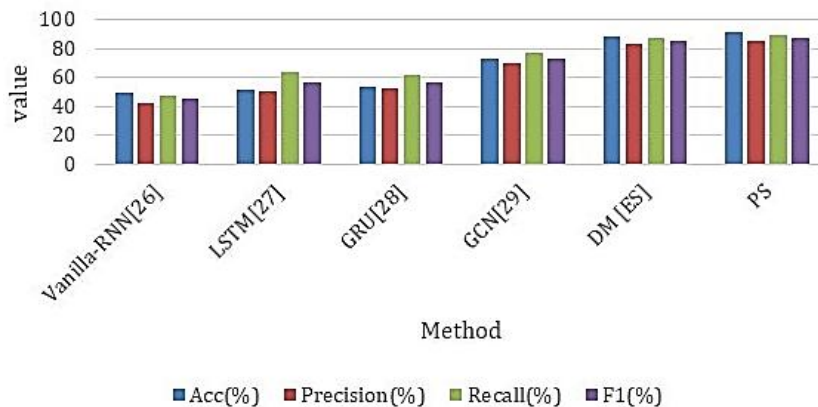


Figure 4. Code injection metric comparison with the proposed model

The analysis of the performance metrics for detecting "Call with Hardcode Gas Amount" vulnerabilities across different models (Vanilla-RNN, LSTM, GRU, GCN) and advanced detection techniques (DM [ES], PS) shows clear trends in improving detection accuracy and balance between precision and recall. GCN outperforms the traditional models (Vanilla-RNN, LSTM, GRU) with a significant jump in accuracy (74.92%), recall (77.97%), and F1 score (74.34%), indicating its stronger capability for detecting this type of vulnerability. However, DM [ES] and PS, representing advanced ensemble methods or post-processing techniques, push the performance further, with PS achieving the highest results across all metrics, including 93.54% accuracy, 91.65% recall, and 89.43% F1 score. These results reflect a more balanced detection capability with minimal false positives and negatives. DM [ES] also performs robustly, especially in terms of precision (86.69%) and recall (87.99%). Overall, while GCN is effective in detecting "Call with Hardcode Gas Amount" vulnerabilities, the advanced methods, particularly PS, demonstrate superior overall performance. Table 4 and Figure 5 shows the comparison of call with hardcode gas amount.

Table 4. Comparison table of call with hardcode gas amount

Call with hardcode gas amount	Vanilla-RNN [25]	LSTM [26]	GRU [27]	GCN [28]	DM [ES] [29]	PS
Acc(%)	52.12	55.28	57.74	74.92	90.62	93.54
Precision(%)	48.64	51.91	54.01	71.03	86.69	88.97
Recall(%)	49.55	68.47	63.64	77.97	87.99	91.65
F1(%)	49.09	59.05	58.43	74.34	87.34	89.43

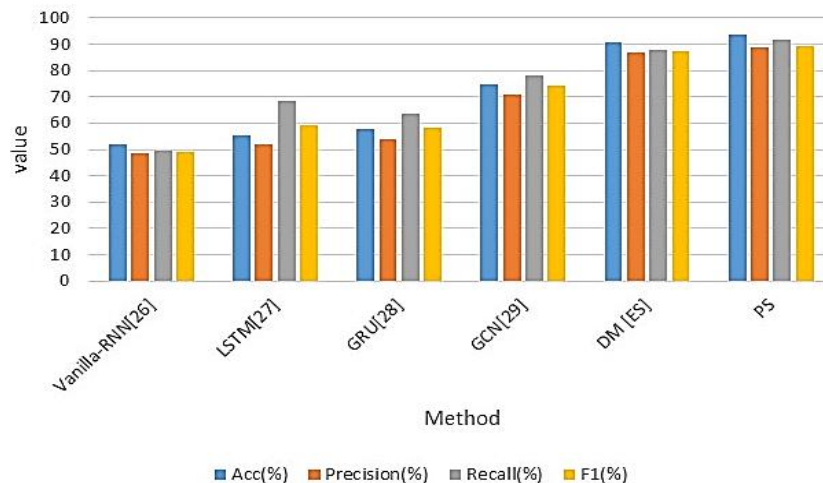


Figure 5. Call with Hardcode gas amount metric comparison with the proposed model

5. CONCLUSION

SCADE offers a comprehensive and effective approach for detecting vulnerabilities in smart contracts, addressing the limitations of traditional methods such as high false positive/negative rates and inability to handle complex vulnerabilities. By combining semantic flow analysis with an ensemble of advanced deep learning models, SCADE provides a fine-grained and robust solution for identifying vulnerabilities, including reentrancy, timestamp dependency, code injection, and hardcoded gas amounts. The use of Contract Structure Mapper and semantic path extraction enables a detailed understanding of contract code, improving the accuracy of vulnerability detection. Extensive evaluations on real-world Ethereum smart contract datasets have shown that SCADE outperforms traditional detection methods in terms of accuracy, scalability, and robustness. The methodology's fine-grained detection capabilities allow for precise identification of vulnerable code segments, ultimately enhancing the security of blockchain systems. SCADE provides a scalable solution for the growing security challenges in decentralized environments, contributing significantly to the field of smart contract security.

ACKNOWLEDGEMENT

I would like to express our sincere gratitude to all those who have supported and contributed to this research project. Primarily, I extend our heartfelt thanks to our guide for her unwavering guidance, invaluable insights, and encouragement throughout the research process. No funding is raised for this research.

FUNDING INFORMATION

No funding is raised for this research.

AUTHOR CONTRIBUTION

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Muralidhara Srirama	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	
Usha Banavikal Ajay		✓				✓		✓	✓	✓	✓	✓		

C : **C**onceptualization

M : **M**ethodology

So : **S**oftware

Va : **V**alidation

Fo : **F**ormal analysis

I : **I**nterpretation

R : **R**esources

D : **D**ata Curation

O : **O**riginal Draft

E : **E**diting

Vi : **V**isualization

Su : **S**upervision

P : **P**roject administration

Fu : **F**unding acquisition

CONFLICT OF INTEREST

Author declares no conflict of interest.

DATA AVAILABILITY

Dataset is utilized in this research.




REFERENCES

- [1] P. Praitheeshan, L. Pan, and R. Doss, "Security evaluation of smart contract-based on-chain ethereum wallets," in *Network and System Security*, Springer International Publishing, 2020, pp. 22–41.
- [2] K. Ramana, R. M. Mohana, C. K. Kumar Reddy, G. Srivastava, and T. R. Gadekallu, "A blockchain-based data-sharing framework for cloud based internet of things systems with efficient smart contracts," in *2023 IEEE International Conference on Communications Workshops (ICC Workshops)*, May 2023, pp. 452–457, doi: 10.1109/iccworkshops57953.2023.10283747.
- [3] W. Wang, Z. Han, T. R. Gadekallu, S. Raza, J. Tanveer, and C. Su, "Lightweight blockchain-enhanced mutual authentication protocol for UAVs," *IEEE Internet of Things Journal*, vol. 11, no. 6, pp. 9547–9557, Mar. 2024, doi: 10.1109/jiot.2023.3324543.
- [4] X. Tang, K. Zhou, J. Cheng, H. Li, and Y. Yuan, "The vulnerabilities in smart contracts: a survey," in *Advances in Artificial Intelligence and Security*, Springer International Publishing, 2021, pp. 177–190.
- [5] P. Momeni, Y. Wang, and R. Samavi, "Machine learning model for smart contracts security analysis," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, Aug. 2019, pp. 1–6, doi: 10.1109/pst47121.2019.8949045.
- [6] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: a tool for clone and bug detection in smart contracts through structural code embedding," Sep. 2019, doi: 10.1109/icsme.2019.00067.
- [7] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, May 2021, pp. 47–59, doi: 10.1145/3457337.3457841.
- [8] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, Aug. 2021, doi: 10.1016/j.infsof.2021.106576.
- [9] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, Jul. 2020, pp. 3283–3290, doi: 10.24963/ijcai.2020/454.
- [10] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities," *arXiv:1811.06632*, 2018.
- [11] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, Apr. 2021, doi: 10.1109/tNSE.2020.2968505.
- [12] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, Aug. 2021, pp. 2751–2759, doi: 10.24963/ijcai.2021/379.
- [13] 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.
- [14] W. Lian, Z. Bao, X. Zhang, B. Jia, and Y. Zhang, "A universal and efficient multi-modal smart contract vulnerability detection framework for big data," *IEEE Transactions on Big Data*, vol. 11, no. 1, pp. 190–207, Feb. 2025, doi: 10.1109/tbdata.2024.3403376.
- [15] F. Yiting, M. Zhao Feng, D. Pengfei, and L. Shoushan, "Automated vulnerability detection of blockchain smart contracts based on BERT artificial intelligent model," *China Communications*, vol. 21, no. 7, pp. 237–251, Jul. 2024, doi: 10.23919/jcc.ja.2023-0189.
- [16] Z. Ali Khan and A. Siami Namin, "Involuntary transfer: a vulnerability pattern in smart contracts," *IEEE Access*, vol. 12, pp. 62459–62479, 2024, doi: 10.1109/access.2024.3351736.
- [17] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021, doi: 10.1109/tkde.2021.3095196.
- [18] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020, pp. 530–541, doi: 10.1145/3377811.3380364.
- [19] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, May 2018, pp. 9–16, doi: 10.1145/3194113.3194115.
- [20] M. Mossberg et al., "Manticore: a user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 1186–1189, doi: 10.1109/ase.2019.00133.
- [21] C. F. Torres, J. Schütte, and R. State, "Osiris: hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, Dec. 2018, pp. 664–676, doi: 10.1145/3274694.3274737.
- [22] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016, pp. 254–269, doi: 10.1145/2976749.2978309.
- [23] Y. Zhang et al., "An efficient smart contract vulnerability detector based on semantic contract graphs using approximate graph matching," *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 21431–21442, Dec. 2023, doi: 10.1109/jiot.2023.3294496.
- [24] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018, pp. 67–82, doi: 10.1145/3243734.3243780.
- [25] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, pp. 347–352, doi: 10.1109/icnn.1996.548916.




- [26] T. Zia and U. Zahid, "Long short-term memory recurrent neural network architectures for Urdu acoustic modeling," *International Journal of Speech Technology*, vol. 22, no. 1, pp. 21–30, Nov. 2018, doi: 10.1007/s10772-018-09573-7.
- [27] A. A. Ballakur and A. Arya, "Empirical evaluation of gated recurrent neural network architectures in aviation delay prediction," in *2020 5th International Conference on Computing, Communication and Security (ICCCS)*, Oct. 2020, pp. 1–7, doi: 10.1109/icccs49678.2020.9276855.
- [28] S. Fu, W. Liu, D. Tao, Y. Zhou, and L. Nie, "HesGCN: Hessian graph convolutional networks for semi-supervised classification," *Information Sciences*, vol. 514, pp. 484–498, Apr. 2020, doi: 10.1016/j.ins.2019.11.019.
- [29] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, "A smart contract vulnerability detection mechanism based on deep learning and expert rules," *IEEE Access*, vol. 11, pp. 77990–77999, 2023, doi: 10.1109/access.2023.3298048.

BIOGRAPHIES OF AUTHOR



Muralidhara Srirama    received his B.E. degree in Computer Science and Engineering from Visvesvaraya Technological University, Belagavi, in 2008 and his M.E. degree in Computer Science and Engineering from University Visvesvaraya College of Engineering, Bangalore University, Bengaluru, in 2012. Currently, he is a Tech Lead at Life9 Systems Private Limited, Bengaluru. He has 10 years of teaching experience and more than 3 years of industry experience. His research interests include blockchain, machine learning, and artificial intelligence. His work encompasses machine learning, IoT, and application development. He can be contacted at email: muralidhars_12@rediffmail.com.



Dr. Usha Banavikal Ajay    is a Dedicated Motivational Individual Committed to Maximizing Learning Opportunities in Diverse Academic Settings Using Consistent and Organized Practices. Energetic and Ambitious Professional With 18 Years of Experience in Teaching. She has completed her research in the area of Information Security. She Obtained her Ph.D. from Visvesvaraya Technological University, Belagavi, Karnataka, India in the Year 2016. She has published more than 35 research papers in reputed International Journal and Conferences where some of them are Scopus indexed and also has good impact factors. She can be contacted at this email: ushaba@bmsit.in.