



OPEN An elegant intellectual engine towards automation of blockchain smart contract vulnerability detection

Balachandar Raju^{1✉} & Gayathri Devi K²

To prevent vulnerabilities and ensure app security, smart contract vulnerability detection identifies flaws in blockchain code. To overcome the limitations of traditional detection methods, this study introduces a novel approach that combines Explainable Artificial Intelligence (XAI) with Deep Learning (DL) to detect vulnerabilities in smart contracts. The proposed intellectual engine operates in multiple stages. First, a smart contract is created, and the user provides a value during the runtime phase. XAI and DL then analyze the opcodes in high-value contracts to detect potentially risky processes. If violations such as security protocol failures, insufficient funds, or account restrictions are found, the engine halts the transaction and generates an error report. If the contract passes this vulnerability assessment, it continues executing without interruption. This ensures flagged transactions remain functional while being assessed. Our proposed Hybrid Boot Branch and Bound Long Short-Term Memory (HB³LSTM) approach achieves outstanding performance, with an accuracy of 99.68%, precision of 99.43%, recall of 99.54%, and an F1-score of 99.40%, which surpasses the performance of existing methods.

Keywords Blockchain, Branch and bound optimization, Deep learning, Opcode, Smart contract, Vulnerability detection

Blockchains are distributed databases with immutability and tamper-resistance¹. In smart contract systems, code is law, and conditions in smart contracts cannot be modified². Smart contracts offer multiple³. However, it can be difficult to address exploitable bugs in smart contracts written in languages like Solidity⁴. Ethereum's open network and control of financial assets make smart contracts easy targets for attackers⁵. Bugs in smart contracts can have global effects, leading to hard forks and significant financial losses⁶. Existing methods for preventing financial losses due to flaw exploitation in smart contracts include data flow analysis, runtime monitoring, fuzzing, symbolic execution, and Satisfiability Modulo Theories (SMT) solutions⁷.

These techniques require human expertise, have long detection times, and do not cover all vulnerabilities⁸. Machine Learning (ML) has shown promise in detecting smart contract vulnerabilities, but existing solutions have limitations, requiring access to source code and being unable to identify specific vulnerability types, only classifying them as binary⁹. Smart contract vulnerabilities can be divided into four groups. Reentrancy attacks, where malicious contracts recursively call the original contract; integer overflow/underflow, causing incorrect calculations; improper access control, allowing unauthorized actions; and unchecked external calls, leading to failed or exploited interactions. Mitigations include using safe math, access controls, and handling external call errors¹⁰.

Additional risks, such as greedy contract designs and unprotected self-destruct instructions¹¹, can lead to Denial of Service (DoS) attacks. Miners can also influence transaction order, causing transaction order dependence¹². Mitigations include using safe math libraries, strong access control, error handling for external calls, and the "Checks-Effects-Interactions" pattern¹³. The use of Deep Neural Network (DNN) detectors is also gaining traction in automatically identifying and mitigating smart contract vulnerabilities¹⁴. These practices emphasize the importance of secure coding and thorough audits to prevent exploitation¹⁵.

¹Department of Computer Science and Engineering, Pollachi Institute of Engineering and Technology, Pollachi, India. ²Department of Electronics and Communication Engineering, Dr. N.G.P Institute of Technology, Coimbatore, India. ✉email: balachandarraju.j@gmail.com

DNN detectors can detect a variety of problems by keeping track of historical vulnerabilities¹⁶. With little data, transfer learning allows for rapid adaptability to new vulnerabilities¹⁷. It can be difficult to analyze Artificial Intelligence (AI) models and feature selection in closed systems¹⁸.

Learning-based vulnerability detection techniques have benefited from DL success in IoT security¹⁹. Effective vulnerability detection is becoming more and more necessary as smart contracts in blockchain systems proliferate in order to stop exploitation²⁰. Conventional approaches lack coverage and are labor-intensive and slow. Although machine learning, especially DL, has potential, it finds it difficult to adjust to new weaknesses. Models must retain prior knowledge while rapidly picking up new information through transfer learning. XAI integration improves smart contract security and guarantees transparency. In this study, a DL model known as HB³LSTM uses Long Short-Term Memory (LSTM) networks to identify smart contract vulnerabilities. For clear and understandable vulnerability detection, it incorporates XAI. The principal findings of this study are as follows:

- The goal is to improve finding vulnerabilities in smart contracts by using a phased approach that integrates DL and XAI to identify violations and ensure the safe execution of high-value contracts.
- HB³LSTM intellectual engine detects smart contract vulnerabilities by combining Branch and Bound Optimization algorithms (BBO) with LSTM networks. BBO is used for optimization, and LSTMs are used for sequence prediction. The model's main goals are to spot potentially dangerous transactions and stop them.
- Improved Quantum online Portfolio Optimization (IQPO) to solve complex problems more quickly and shorten the time it takes to find vulnerabilities. This improves scalability and makes it possible to react to changes in the economy in real time by adjusting the portfolio.
- Smart contracts can be vulnerable to opcodes, causing issues like reentrancy, overflow, or access control flaws. XAI's SHAP (SHapley Additive exPlanations) values are employed; contracts with positive values are considered invulnerable, while those with negative values are considered vulnerable.

This study is divided into five components. A review of the literature is given in “Literature Survey” Section, and the suggested model is explained in “Proposed methodology” Section. The outcomes of the suggested procedures are displayed in “Results and discussions” Section, along with a comparison of this model with a few other modern methods. The discussion is presented in “Conclusion” Section, and recommendations for additional research are made in Sect. 6.

Literature survey

In 2021, He et al.²¹ introduced a model for smart contract vulnerability detection, leveraging BERT for semantic feature extraction, BiLSTM for sequence learning, and an attention mechanism to prioritize critical features. This approach enhances detection accuracy and generalization, outperforming traditional methods in identifying security flaws in smart contracts.

In 2022, Jingya Dong et al.²² Proposed a feasible energy trading method to achieve a self-sufficient energy consumption. It provides the accurate energy transfer signals in the blockchain to attain the better self-sufficiency of transactions.

In 2022, Jingya Dong et al.²³ suggested a novel transaction processing method to protect private data using mapping algorithm to solve the privacy attacks. It also improves the performances of efficient executions of transactions using the privacy preserving decentralized energy trading scheme.

In 2022, Jingya Dong et al.²⁴ introduced a method light weight data fusion to reduce the network congestions and wastage of bandwidth with the secure analysis for IOT. Furthermore, proposed an improved hierarchical fuzzy based hashing technique to local the anomalies in the machine learning models to ensure the security of the sensitive data.

In 2023, Wu et al.²⁵ investigated the techniques for locating smart contract vulnerabilities, highlighting the shortcomings of conventional methods. They highlighted the use of machine learning, especially DL and attention processing, for improved accuracy. The paper's conclusion noted the need for detecting methods that are more dependable and scalable.

In 2023, Yazdinejad et al.²⁶ introduced a secure, intelligent fuzzy blockchain framework to enhance threat detection in blockchain-based IoT networks, addressing uncertainty and ambiguity in IoT data. It combines a fuzzy DL model, optimized ANFIS, fuzzy matching, and a fuzzy control system to detect and mitigate network attacks.

In 2023, Chen et al.²⁷ suggested a novel method for smart contract vulnerability detection by constructing a Semantic Graph (SG) for each function, capturing both syntax and semantic relationships. It then utilizes an Edge-Attention Residual Graph Convolutional Network (EA-RGCN) to extract content and semantic features.

In 2023, Ma et al.²⁸ introduced a Hierarchical Graph Attention Network (HGAT) for smart contract vulnerability detection, addressing the inefficiencies of existing methods. It Constructs Code Graphs (CFG) from Abstract Syntax Tree (AST) and Control Flow Graph (CFG) to extract node features and applies Graph Attention Network (GAT) for feature learning.

In 2023, Jie et al.²⁹ introduced static analysis and multimodal AI techniques that have improved the detection of smart contract vulnerabilities. To define 84 efficient vulnerability-uncovering techniques, strategies make use of code and graph embeddings from word2vec, Transformer-Based Bidirectional Encoder Representation (BERT), and Graph Convolutional Network (GCN) models. In intermodal, intermodal, and multimodal contexts, this encompasses feature selection, fusion, training, and decision-making units. Bidirectional Long Short-Term Memory (BiLSTM) models, thick layers, Random Forest (RF), Max Pooling (MP), Spatial Pyramid Pooling (SPP), and text Convolutional Neural Networks (CNNs) are examples of high-accuracy jobs.

In 2023, Dong et al.³⁰ proposed a network called decentralized autonomous oracle network combined with consensus protocol and non-interactive reputation maintenance to secure the smart contract to be more reliable and tamper-proof inputs and output.

In 2024, Osei et al.³¹ suggested WIDENNET, a DL-based method using WDE and DNN to detect vulnerabilities in smart contracts. It focuses on identifying reentrancy and timestamp dependence issues by extracting byte codes and converting them into operational codes.

In 2024, Sharma et al.³² introduced two detection techniques used by the Intrusion Detection System (IDS): anomaly-based IDS, which detects known hostile activities, and signature-based IDS, which detects abnormal system behavior. Signature-based IDS is inefficient due to its reliance on pre-existing signatures and limitations in storage and computation, making it unable to detect new attacks. Anomaly-based IDS can detect novel attacks but may also produce false positives.

In 2024, Zhen et al.³³ introduced a Dual Attention Graph Neural Network (DA-GNN) for smart contract vulnerability detection. It converts Control Flow Graph (CFG) opcode sequences into feature matrices and leverages a dual attention mechanism for improved node embedding updates.

In 2024, Wu et al.³⁴ introduced a two-pronged approach to smart contract vulnerability detection. It improved detection efficiency and accuracy by combining static and dynamic analysis. The approach provided a more reliable security solution and performed better than conventional methods.

In 2024, Mothukuri et al.³⁵ suggested a novel AI-driven solution to address the DeFi credibility issue by introducing a Trust Score system that evaluates DeFi projects using four key risk factors. These include smart contract vulnerabilities, suspicious transactions, anomalous price changes, and scam sentiment from social media, two of which are novel in DeFi fraud detection.

In 2025, Dong et al.³⁶ introduced a P2P energy efficient trading system to permit the users to conduct an energy efficient transaction without the having the contribution of third party. It helps to improve the high performance and efficiency to meet the energy efficient tractions in the trading system. A comparison of recent works is presented in Table 1.

Problem statement

One of the main challenges to smart contract analysis is the dearth of open-source resources, the inability to discover vulnerabilities in bytecode, and a high false rate in detection techniques. Complexity is increased by scalability problems, lengthy vulnerability discovery times, and the requirement for ongoing IoT network monitoring. Furthermore, issues with large data quality, computational cost, and the lack of transparency of AI models undermine confidence in automated analysis, which makes it more difficult to properly protect these systems. To address these drawbacks, the proposed method, “HB³LSTM: Smart contract-based vulnerability detection using deep explainable AI,” uses an HB³LSTM intellectual engine framework to overcome these challenges. The IQPO is used to reduce vulnerability detection processing time and improve scalability. The HB³LSTM intellectual engine can improve predictive accuracy and reduce risks by efficiently resolving vulnerabilities. Furthermore, SHAP helps to improve the accuracy. Our ultimate objective is to improve the detection methods’ capacities in order to more accurately detect smart contract flaws.

Authors	Focus	Technique	Advantages	Limitations
Chen et al. ²⁷	Semantic-based vulnerability detection	Residual Graph Convolutional Networks (GCN) with Edge Attention	Captures contextual information in smart contract code effectively	Edge attention mechanism increases computational complexity
Ma et al. ²⁸	Graph-based vulnerability detection	Hierarchical Graph Attention Network (HGAT)	Improves detection accuracy by considering hierarchical relationships	Requires high-quality labeled data for optimal performance
Jie et al. ²⁹	Blockchain programs for IoT use contract-oriented languages, executed automatically	Blockchain technology	Enhancing IoT solutions with technical capabilities	Traditional methods rely on rules, struggle with false positives, low accuracy
Osei et al. ³¹	Wide and DL for vulnerability detection	Wide and Deep Neural Network (WDNN)	Combines memorization and generalization capabilities	May struggle with unseen vulnerability patterns
Sharma et al. ³²	IoT security layers discussed, DL model proposed for intrusion detection, explainable AI used for high accuracy	Intrusion Detection System (IDS)	Detect intrusions in IoT networks, prevent malicious activities efficiently	IoT systems face daily attacks, identification and mitigation needed for network protection
Zhen et al. ³³	Smart contract vulnerability detection	Dual Attention Graph Neural Network (DA-GNN)	Improves accuracy by leveraging dual attention for better feature extraction	Computationally intensive due to complex graph processing
He et al. ²¹	Enhancing smart contract security	Pre-trained language models for vulnerability detection	Leverages NLP advancements for precise vulnerability identification	Requires extensive training data and high computational resources
Wu et al. ²⁵	Smart contract vulnerability detection using hybrid attention	Combine self-attention and convolutional layers	Improved accuracy, handles complex patterns well	Computationally expensive, struggles with large datasets
Wu et al. ³⁴	Improved dual-channel technique for identifying vulnerabilities in smart contracts	Dual-channel approach for syntactic and semantic analysis	Enhanced detection performance and faster processing	Increased computational cost and scalability issues
Mothukuri et al. ³⁵	Trust scoring and vulnerability detection in DeFi projects	XGBoost	Multi-perspective evaluation with Enhanced transparency and trust	May not scale efficiently across all DeFi platforms
Yazdinejad et al. ²⁶	Threat detection in IoT networks using blockchain	DNN	Decentralized architecture enhances security	High resource requirements for fuzzy logic and blockchain computation

Table 1. Comparisons for recent works.

Proposed methodology

The proposed methodology for identifying smart contract vulnerabilities includes several important phases. First, data from Smart Bugs Wild Dataset uses Natural Language Processing (NLP) techniques for preprocessing, such as word segmentation, lexical analysis, word-to-vector conversion, and TF-IDF for feature extraction. To maximize processing time and scalability, feature selection is then carried out using IQPO. A new hybrid approach called the HB³LSTM intellectual engine finds and blocks risky transactions and extracts code fragments that highlight vulnerabilities. To improve vulnerability detection, the model collaborates with expert systems. The next step involves interpreting model outputs using XAI approaches like SHAP values. Negative values draw attention to aspects associated with vulnerabilities, making it easier to identify crucial opcodes influencing contract security, while positive values show characteristics that contribute to invulnerable contracts. Figure 1 describes the overall proposed methodology.

Dataset collection

Our study leverages the Kaggle Smart Bugs Wild Dataset³⁷, which includes over 1250 Solidity-written smart contracts, vulnerable and non-vulnerable, annotated with specific vulnerability types such as reentrancy, integer overflow/underflow, and access control issues.

Opcode extraction

Smart contracts are self-executing agreements with terms encoded in opcode. Simplifying opcode enhances efficiency and reduces transaction costs by minimizing complexity. This ensures correct operation across scenarios like caller authorization and transaction validity while preserving functionality [38]. Table 2 describes the opcode extraction from the SmartBug wild dataset.

The input–output mechanism of SOLC is used to extract opcodes, guaranteeing consistency between compiler versions. The function name can be compared to the Application Binary Interface (ABI); a cross-check is carried out to retrieve opcodes solely from injected contracts. Algorithm (1) describes the opcode extraction algorithm in detail. An ABI and an ordered set of opcodes are both included in every contract.

$$C = \{(ABI_1, O_1), (ABI_2, O_2), \dots, (ABI_n, O_n)\} \tag{1}$$

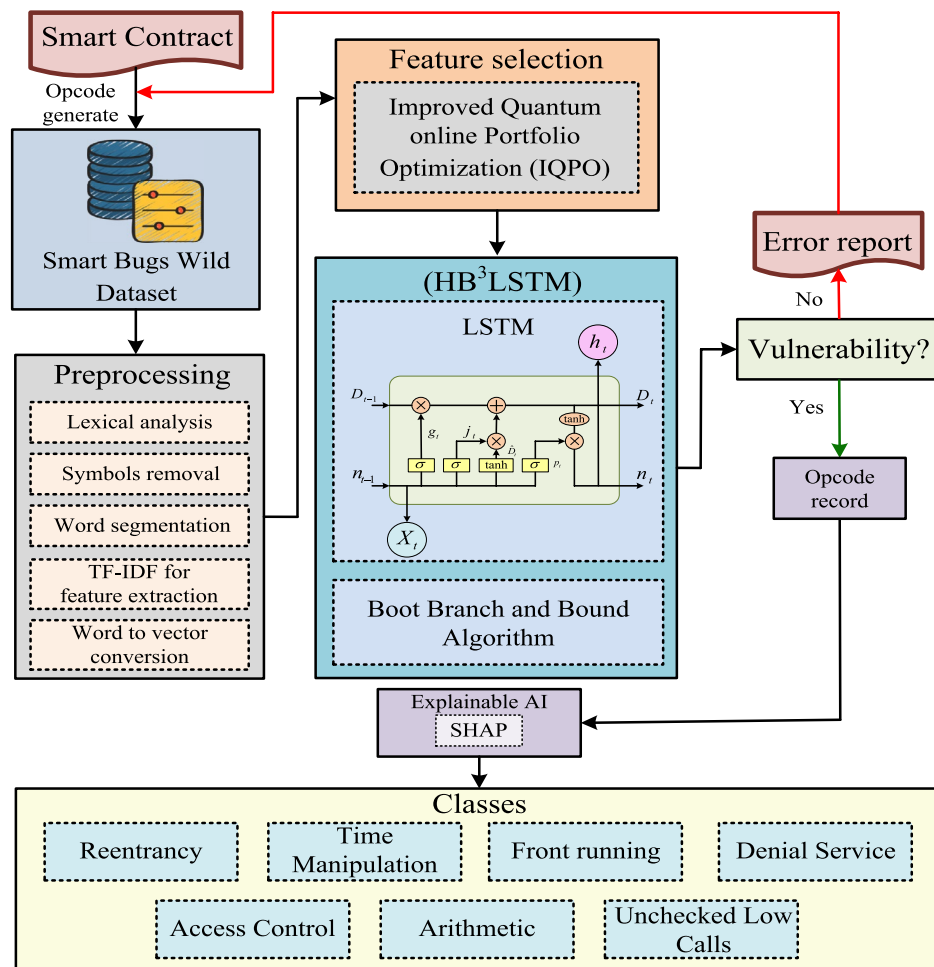


Fig. 1. Overall proposed methodology.

Algorithm 1 Extract Opcodes From Bugged Files

```

1: procedure SimplifyOpcode( $g_c, B_{used}^*$ )
2:   Initialize  $L_{opcode} \leftarrow \phi$ 
3:    $C \leftarrow SOLC(g_c)$ 
4:   for  $(ABI, O) \in C$  do
5:     found_match  $\leftarrow$  False
6:     for  $bug \in B_{used}^*$  do
7:       for  $F_j \in ABI$  do
8:         if  $name_j = bug.name$  then
9:           found_match  $\leftarrow$  True
10:           $O^* \leftarrow \text{Simplify}(O)$ 
11:           $L_{opcodes} \leftarrow L_{opcodes} \cup \{O^*\}$ 
12:         end if
13:       end for
14:     if found_match then
15:       Break
16:     end if
17:   end for
18: end for
19: return  $L_{opcodes}$ 
20: end procedure

```

Table 2. Opcode extraction from SmartBug wild dataset.

where C is the collection of contracts that SOLC returned in order to explain the sifting procedure, ABI_n indicates the ABI for the n^{th} contract and O_n indicates the ordered list of opcode for the n^{th} contract. Opcodes like PUSH, which has 32 possibilities, and other operations like MSTORE, CALL VALUE, ISZERO, and JUMPI are simplified to minimize variations.

Preprocessing

In data preprocessing, smart contract code undergoes lexical analysis, symbol removal, and word segmentation; TF-IDF is applied for feature and word-to-vector conversion maps²⁵.

Lexical analysis in smart contract vulnerability identification tokenizes raw Solidity code into meaningful units like keywords, operators, identifiers, and delimiters while removing comments and whitespace. Symbol removal eliminates non-informative components like comments, blank lines, and useless code, reducing noise and computational complexity. This helps the model focus on relevant code patterns, improving accuracy and speeding up training and inference. Word segmentation divides code into discrete tokens, ensuring each component is processed independently while preserving structure. This enhances the model's ability to detect vulnerabilities by focusing on important patterns and relationships. TF-IDF is a text vectorization method for feature extraction. It combines Term Frequency (TF), which weights words based on their frequency in a document, with Inverse Document Frequency (IDF), which reduces the weight of common terms across documents.

$$TF(t, d) = \frac{b}{B} \quad (2)$$

where b represents the number of times t appears in the document d and B represents the total number of terms in a document d .

```

# Define a function for preprocessing
def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()
    # Remove special characters and digits
    text = re.sub(r'^a-zA-Z\s]', '', text)
    # Tokenize the text
    tokens = word_tokenize(text)
    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]
    # Join the tokens back into a string
    text = ' '.join(tokens)
    return text

```

Fig. 2. Preprocessing code.

Parameters	Values	Description
max_features	10,000	Captures the most relevant tokens
N-grams	(1,2)	Uses unigram and bigram for context preservation
Stop words	Removed (NLTK)	Removes common words using the NLTK stopword list
TF-IDF Norm	L ₂	Normalizes feature weights using L2 norm
smooth_IDF	True	Prevents zero division errors in IDF computation
Sublinear TF Scaling	True	Applies to dampen the effect of frequent terms

Table 3. TF-IDF parameter settings.

$$IDF(t, D) = \log \left(\frac{a}{A} \right) \quad (3)$$

where a denotes the total number of documents in the corpus and A denotes the number of documents containing term t . The TF-IDF score for a term in a document is obtained by multiplying its TF and IDF scores.

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D) \quad (4)$$

TF-IDF transformation can be as a

$$z = \text{encoder}(X) \quad (5)$$

where X denotes the input feature matrix extracted from smart contract opcode and z denotes the resulting TF-IDF vector representation of the input data. This map is to the actual application of converting tokens to vector form using the TF-IDF encoder. Figure 2 shows the code for the preprocessing.

The parameter settings for the TF-IDF were selected based on a systematic hyperparameter tuning procedure. Each setting was evaluated through repeated experiments using a validation split to measure its effect on downstream model performance. The max_features parameter was set to 10,000 after testing various thresholds; this value best preserved relevant vocabulary while preventing overfitting. The choice of n-grams (1, 2) was validated through comparisons with unigram- and trigram-only setups, where (1, 2) provided optimal contextual richness. Stop word removal via NLTK improved classification accuracy by reducing noise. L₂ normalization outperformed L₁ in maintaining consistent feature scaling and stable model behavior. Smooth_IDF was enabled to handle unseen terms without destabilizing inverse frequency values. Sublinear TF scaling was included after observing improvements in model generalization and robustness. This tuning process involved grid-like evaluation and iterative refinement across these parameters, as shown in Table 3.

Word-to-vector conversion captures the semantic linkages between code parts by converting tokens into numerical vectors using embedding approaches. This improves the model's capacity to identify vulnerabilities like inappropriate variable use or dangerous dependencies by helping it comprehend how the contract interacts.

Improved quantum online portfolio optimization (IQPO) for feature selection

IQPO³⁹ is a quantum computing-based framework that enhances online decision-making in financial markets by leveraging quantum algorithms such as quantum state preparation, norm estimation, and inner product estimation. It is chosen for feature selection because of its ability to process high-dimensional data efficiently, dynamically adapt to evolving datasets, and achieve a quadratic speedup in computation compared to classical methods. The key advantage of IQPO is its ability to use quantum oracles to encode feature relevance scores, perform fast probabilistic sampling using multi-sampling algorithms, and estimate importance measures through quantum inner product computations, making it ideal for selecting the most informative features in real-time machine learning tasks.

Quantum representation of feature data

Instead of classical access to feature vectors, assume quantum access through a set of unitaries $P_{\rho^{(u)}}$, representing the transformed feature importance values at time u . The input data is encoded as

$$P_{\rho^{(u)}} |j\rangle |\vec{0}\rangle = |k\rangle \left| \rho_j^{(u)} \right\rangle \quad (6)$$

where $\rho^{(u)} = (\rho_1^{(u)}, \rho_2^{(u)}, \dots, \rho_n^{(u)})$ is the feature vector at the time u , $\max_{j \in [o]} \rho_j^{(u)} = 1$ ensures feature values are normalized and $\rho_j^{(u)} \geq s_{\min} > 0$ guarantees all features have a minimum significance level. This encoding allows efficient quantum computations on the feature space, preparing for importance ranking.

Quantum portfolio weight update rule

The weight update for each feature follows a softmax-like transformation, capturing cumulative importance over multiple time steps:

$$\omega_j^{(u+1)} = \frac{\exp\left(\eta \sum_{u'=1}^u \frac{\rho_j^{(u')}}{\omega^{(u')} \cdot \rho^{(u')}}\right)}{\sum_{j=1}^o \exp\left(\eta \sum_{u'=1}^u \frac{\rho_j^{(u')}}{\omega^{(u')} \cdot \rho^{(u')}}\right)} \quad (7)$$

where $\omega(1) = (\frac{1}{n}, \dots, \frac{1}{n})$ initializes all features with equal importance, η denotes a learning rate that controls the impact of past observations, and $\rho_j^{(u)}$ denotes the importance score of feature j at time u . This update ensures that the most relevant features gain higher importance while less significant features are gradually suppressed.

Quantum computations for feature selection

To efficiently compute cumulative feature importance in superposition, define the following transformations:

$$|j\rangle |\vec{0}\rangle \rightarrow |j\rangle \left| \sum_{u'=1}^{u-1} \frac{\rho_j^{(u')}}{\bar{j}^{(u')}} \right\rangle \quad (8)$$

This transformation allows encoding past feature gains into a quantum state for efficient parallel computation. Furthermore, a quantum unitary to extract feature selection scores:

$$|j\rangle |0\rangle \rightarrow |j\rangle r_j^{(u)}, \quad r_j^{(u)} = \exp\left(\eta \sum_{u'=1}^{u-1} \frac{\rho_j^{(u')}}{\bar{j}^{(u')}}\right) \quad (9)$$

This representation allows rapid ranking of features based on their cumulative contribution.

Quantum norm estimation for feature importance

To measure the total contribution of each feature vector w , estimate its norm using quantum queries:

$$\left| \|w\|_1 - \tilde{a} \right| \leq \varepsilon \|w\|_1 \quad (10)$$

The required query complexity for this estimation is

$$P\left(\frac{\bar{z}}{\varepsilon} \log\left(\frac{1}{\delta}\right)\right) \quad (11)$$

where ε controls the approximation accuracy, δ denotes the confidence level of the estimation, and z denotes the total number of features. This step ensures that feature importance scores are precisely quantified, helping to refine the selection process.

Quantum inner product estimation for feature ranking

To evaluate the correlation between feature vectors, compute the inner product using quantum estimation:

$$|\tilde{J}\tilde{Q} - v \cdot w| \leq \varepsilon v \cdot w \quad (12)$$

where $\tilde{J}\tilde{Q}$ denotes the quantum estimate of $v \cdot w$. This estimation is performed efficiently with a quantum complexity of

$$P \left(\frac{\sqrt{m}}{\varepsilon \sqrt{v_{\min}}} \log \left(\frac{1}{\delta} \right) \right) \quad (13)$$

where δ denotes the failure probability, P denotes the number of quantum gates, m denotes the feature vectors, v_{\min} ensures that no feature has a negligible contribution. By applying inner product estimation, features undergo ranking based on their relevance.

Quantum multi-sampling for feature selection

To finalize feature selection, quantum multi-sampling identifies the top p features by ensuring:

$$U \geq \|qx\|_1 \quad (14)$$

$$|U - \|q\|_1| \leq \min \{1/\sqrt{p}, \varepsilon\} \|q\|_1 \quad (15)$$

where U denotes the inner product sum over selected features, $\|q\|_1$ denotes the expected outcome, \sqrt{p} denotes the number of quantum gates root values, and $\|qx\|_1$ denotes the length of the vector. The expected run-time complexity for multi-sampling is

$$P \left(\sqrt{tz} \log \left(\frac{1}{\delta} \right) \right) \quad (16)$$

where P denotes the cumulative importance of selected features, tz denotes the number of features to select. This step ensures that only the most significant features are retained for vulnerability detection. Once IQPO selects the most informative features, they are used as input to the HB³LSTM model.

$$X' = QOPO(X) \quad (17)$$

where X denotes the original feature matrix before selection, and X' denotes the optimized and selected subset of features most relevant to vulnerability detection. This represents IQPO transforming the original feature set into a selected, optimized set. HB³LSTM processes the feature vectors and applies DL techniques to detect software vulnerabilities. This integration enhances both accuracy and computational efficiency, enabling real-time detection of potential security threats. The IQPO algorithm was tested primarily on Qiskit's statevector simulator to emulate noiseless quantum computations. The simulation utilized 8 qubits to represent the feature space encoding and enable quantum operations such as state preparation, norm estimation, and inner product estimation. The choice of 8 qubits balances computational tractability with sufficient feature dimensionality representation for the datasets in use.

Table 4 shows the parameter settings for the IQPO. The QLR was set to 0.001 to ensure gradual convergence while maintaining stability, particularly in complex search spaces. The decay rate of 0.95 was chosen to enable adaptive learning rate reduction, helping the algorithm fine-tune its learning behavior over time. A regularization factor of 0.01 was incorporated to mitigate overfitting by penalizing overly complex solutions, which is essential for generalization. The quantum walk probability was fixed at 0.5 to strike a balance between global exploration and local exploitation. Although a systematic hyperparameter tuning procedure such as grid search or Bayesian optimization was not exhaustively applied due to computational constraints, the selected values were empirically validated through sensitivity analysis across benchmark datasets. This approach ensured that the chosen parameters provided consistently good performance, even if not globally optimal.

Hybrid boot branch and bound long short-term memory (HB³LSTM) for vulnerability detection

The HB³LSTM is a combination of Boot LSTM⁴⁰ and BBO. It is a novel strategy that combines the advantages of several approaches to enhance smart contract vulnerability identification. This method combines an LSTM

Parameter	Values	Description
Quantum learning rate (QLR)	0.001	Controls convergence speed
Decay rate	0.95	Adjusts learning rate adaptively
Regularization factor	0.01	Prevents overfitting by adding a penalty term
Quantum walk probability	0.5	Balances exploration and exploitation

Table 4. Parameter settings for IQPO.

network, which functions well for a series of prediction tasks, with algorithms that are commonly employed in optimization problems. With an emphasis on identifying potentially hazardous transactions and preventing risky behavior in smart contracts, the HB³LSTM model is especially made to extract code fragments that expose vulnerabilities.

LSTM

A specialized kind of Recurrent Neural Network (RNN) called an LSTM is employed in our study as a model to capture temporal dependencies in sequential data. LSTM learns sequence patterns by retaining knowledge across time steps through gates (input, forget, output) and a tanh layer. The cell state carries important information, enabling selective updates. This architecture is ideal for sequence prediction tasks. The typical LSTM cell structure is depicted graphically in Fig. 3.

Which data from earlier time steps should be ignored is decided by the first layer, also referred to as the forget layer. Equation (18) provides the mathematical expression of the forget gate's output (g_t).

$$g_t = \sigma(W_g \cdot [n_{t-1}, X_t] + b_f) \quad (18)$$

where σ denotes the sigmoid role of activation, W_g denotes the forget gate's weight, b_f indicates the forget gate's bias, X_t denotes the time t and n_{t-1} is the hidden layer of time. In the LSTM, the input gate (j_t) is the second layer that decides whether the cell state receives fresh data. This choice, determined by applying the subsequent formula, is shown in Eq. (19).

$$j_t = \sigma(W_j \cdot [n_{t-1}, X_t] + b_j) \quad (19)$$

where W_j indicates input gate weight and b_j indicates input gate bias. The tanh layer, sometimes referred to as the cell state layer (\hat{D}_t), is the third layer. Equation (20) defines the vector of new candidate values that are produced by this layer:

$$\hat{D}_t = \phi(W_D \cdot [n_{t-1}, X_t] + b_D) \quad (20)$$

where ϕ denotes the function of tanh, W_D denotes the weight of the cell, and b_D denotes the bias cell. The old cell state \hat{D}_t is changed into the new cell state D_t following the first three layers. The interplay between the input gate and the forget gate produces this update. According to Eq. (21), new data is added by the input gate, while old data is discarded by the forget gate.

$$D_t = g_t D_{t-1} + j_t \hat{D}_t \quad (21)$$

where D_t new cell, J_t denotes the input gate, and g_t denotes the output gate. The last layer, the output gate, is in charge of generating the final output in accordance with the modified cell state. The output gate operates in the following manner:

$$p_t = \sigma(W_p \cdot [n_{t-1}, X_t] + b_p) \cdot \phi(D_t) \quad (22)$$

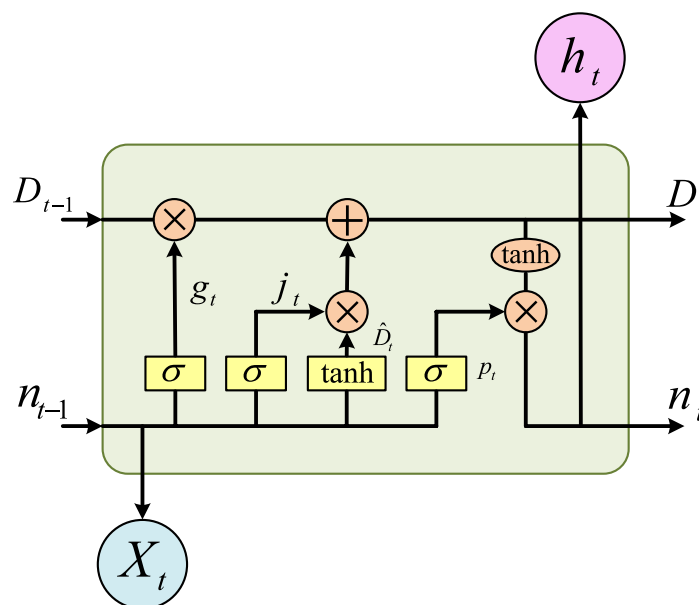


Fig. 3. Typical LSTM cell structure.

where W_p denotes output gate weight and the b_p indicates output gate bias.

Bootstrap

Bootstrap is a general statistical inference approach that builds a sampling distribution by uniformly sampling with replacements from the original data. It is widely used as a robust alternative to parametric statistical inference, which may be unreliable due to complexities in computing standard errors. Bootstrap methods are particularly useful when parametric assumptions fail or are difficult to verify. Three bootstrap techniques exist for regression analysis: pairs bootstrap, standard residuals bootstrap, and wild residuals bootstrap. Among these, pairs bootstrap is preferred for problems like SPF, where observations are correlated. This method helps preserve the dependence structure between observations, ensuring more reliable statistical inference. By resampling from the original data, it provides better estimations without relying on strict distributional assumptions. For vulnerability detection, bootstrap and LSTM integrate by leveraging bootstrap resampling to enhance training data variability and model robustness. Bootstrap helps generate diverse datasets, reducing overfitting, while LSTM captures temporal dependencies for accurate predictions. This combination improves the reliability and generalization of the vulnerability detection model.

Integrated boot LSTM

In the proposed Boot-LSTM framework for vulnerability detection, bootstrapping enhances the robustness of LSTM models by training them on multiple bootstrapped datasets. This approach is particularly beneficial in handling the variability and uncertainty in cybersecurity threat patterns caused by evolving attack techniques. Furthermore, bootstrapping facilitates the exploration of LSTM behavior on different resampled sequences of vulnerability datasets, each potentially representing diverse cyber threat scenarios. By continuously generating diverse detection outcomes during resampling and leveraging temporal correlations identified by LSTM, the model produces high-quality predictive indicators that account for the inherent uncertainty of threat evolution. To improve detection accuracy, the model's weights are refined through the BBO algorithm, enhancing the reliability of vulnerability assessments. This hybrid approach strengthens adaptability and reduces false positives, leading to more effective cyber threat detection in dynamic environments. The optimized feature set is passed through an encoding module of HB³LSTM to generate latent representations z , which are further used by a classifier layer to produce vulnerability predictions y_{pred} .

$$y_{pred} = \text{classifier}(z) \quad (23)$$

Where y_{pred} denotes the predicted output indicating software vulnerability status. The detection stage identifies various classes of software vulnerabilities, including time manipulation, front running, DoS, reentrancy, access control issues, arithmetic errors, unchecked low-level calls, and other types. The architecture of HB³LSTM has been shown in Fig. 4.

Table 5 shows the hyperparameter settings for HB³LSTM. The hyperparameters were chosen based on a combination of empirical testing, and domain-specific considerations to ensure robust performance. The three-layer structure allows HB³LSTM to effectively capture hierarchical temporal features, a common practice in deep sequence models. A hidden unit size of 256 provides a balance between expressive power and computational tractability. The dropout rate of 0.3 was selected based on preliminary experiments that indicated it effectively reduced overfitting without compromising learning. A batch size of 64 was used to balance memory constraints with training stability. The BBO optimizer was chosen for its proven adaptability and fast convergence in DL tasks. The learning rate of 0.0005 was fine-tuned empirically to ensure smooth training dynamics. Overall, the selected hyperparameters reflect a pragmatic balance between experimental insight and best practices from the existing performance.

Enhancing weight update with branch and bound algorithm

An optimization technique called BBO⁴¹ has been effectively used for smart contract vulnerability detection. BBO is an optimization algorithm that systematically explores solution spaces by computing bounds and eliminating suboptimal solutions, making it effective for constrained problems. It is chosen over heuristic or greedy methods because it guarantees global optimality while efficiently navigating large solution spaces through intelligent pruning. Unlike heuristic or greedy methods, BBO avoids unnecessary computation by bounding suboptimal regions, making it scalable and practical for high-dimensional problems. Its flexibility supports complex constraints like sparsity, and it can be tailored to exploit specific problem structures. It balances optimality, efficiency, and adaptability, making it ideal for structured optimization tasks. In the optimization, the weight parameters of the Boot LSTM network are trained using BBO to enhance convergence and prevent getting stuck in local minima. By applying BBO, the model efficiently refines weight updates, prioritizing critical features related to smart contract vulnerabilities.

Initialization

For the initialization of the first and second blocks, it can be represented as

$$M^{(0)} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, v^{(0)} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (24)$$

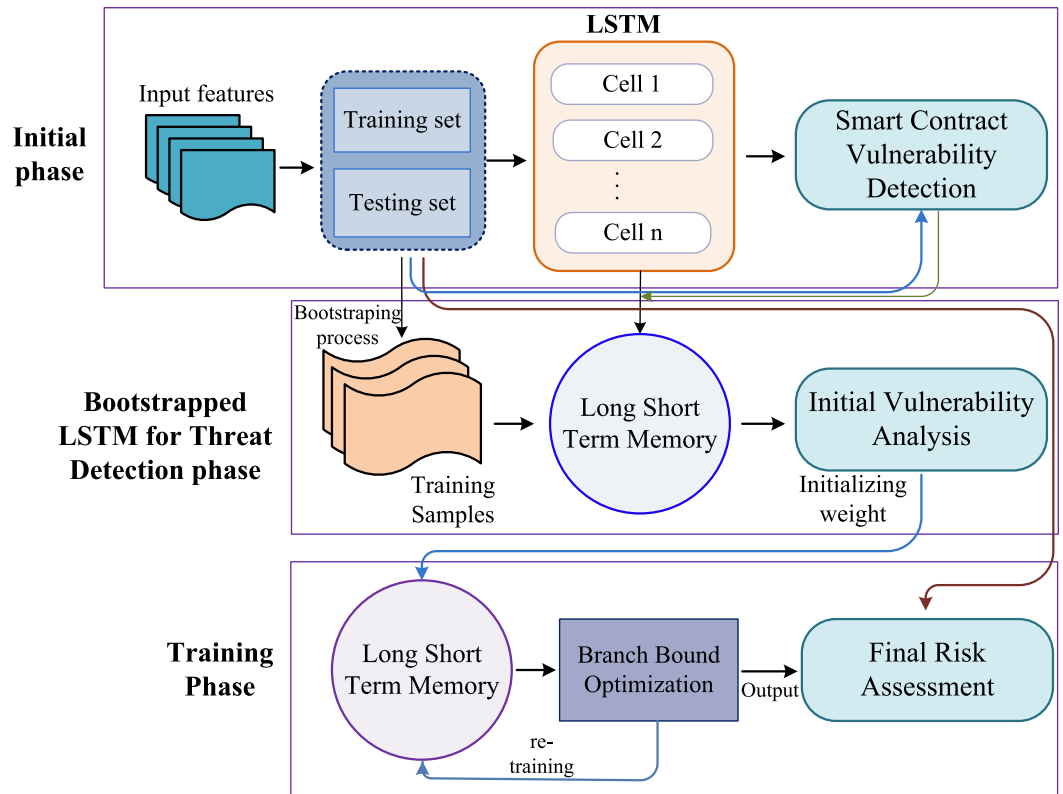


Fig. 4. Architecture of HB³LSTM.

Parameter	Values	Description
Number of layers	3	Defines a deep hierarchical structure
Hidden UNITS	256	Optimal for learning contextual dependencies
Dropout rate	0.3	Prevents overfitting by randomly dropping units
Batch size	64	Balances computational efficiency and performance
Optimizer	BBO	Uses an adaptive learning optimization algorithm
Learning rate	0.0005	Fine-tuned for stability

Table 5. Hyperparameter settings for HB³LSTM.

where $M^{(0)}$ denotes the initialization phase of the upper bound, and $v^{(0)}$ denotes the initialization phase of the lower bound.

Fitness function

In the fitness function, LSTM weight parameters are optimized to minimize detection error and enhance model accuracy. It can be denoted as

$$Fitness\ function = Min(W_D) \tag{25}$$

where $Min(W_D)$ denotes the minimizing the weight from the Boot LSTM. By optimizing this function, BBO ensures that the model reduces false positives, improves convergence speed, and focuses on learning from critical opcode sequences related to smart contract vulnerabilities. The fitness evaluation continues iteratively until optimal or near-optimal values are found.

Branching process

The branching process is central to the BBO algorithm. It systematically explores the search space using a binary enumeration tree, where each node corresponds to a candidate subproblem. At each node, a pair of binary vectors is maintained. For the lower bound vector:

$$M := (M_1, M_2, \dots, M_{q+r})^T \in \{0, 1\}^{q+r} \tag{26}$$

For the upper bound vector:

$$v := (v_1, v_2, \dots, v_{q+r})^T \in \{0, 1\}^{q+r} \quad (27)$$

Each binary variable $a_k \in \{0, 1\}$ is bounded by the corresponding entries $M_k \leq a_k \leq v_k$ for all $k \in \{1, \dots, p\}$. At the root node, starts with $M = 0$ and $v = 1$, representing the unconstrained space. The process of branching involves selecting an unfixed variable a_k and creating two child nodes as $M_k = v_k = 0$ and $M_k = v_k = 1$. This recursive process continues until all variables are uniquely determines a . The feasibility of a node is ensured by a constraint that the number of non-zero entries in a subset of a should lay within a desired target:

$$\sum_{k=1}^q M_k \leq \theta_y \leq \sum_{k=1}^q v_k, \quad \sum_{k=1}^r M_{q+k} \leq \theta_z \leq \sum_{k=1}^r v_{q+k} \quad (28)$$

where q denotes the number of variables in the first block, r denotes the number of variables in the second block, and θ_y, θ_z denotes the target sparsity level for each block. This branching structure allows for efficient pruning and ensures only feasible and promising paths are explored.

Terminal node

A terminal node is one where no further branching is necessary because the solution is either fully determined or satisfies the stopping condition. The terminal node function can be represented as:

$$\text{terminal}((M, v), (t, u), \theta) := \begin{cases} \text{true} & \text{if } \sum_{k=t}^u M_k = \theta, \\ \text{true} & \text{if } \sum_{k=t}^u v_k = \theta, \\ \text{false} & \text{otherwise} \end{cases} \quad (29)$$

For the first block, the node is terminal if:

$$\begin{aligned} \text{terminal}((M, v), (1, q), \theta_y) &= \text{true} \\ \sum_{k=1}^q M_k &= \theta_y \quad (\text{or}) \quad \sum_{k=1}^q v_k = \theta_y \end{aligned} \quad (30)$$

For the second block, the node is terminal if:

$$\begin{aligned} \text{terminal}((M, v), (q+1, q+r), \theta_z) &= \text{true} \\ \sum_{k=1}^r M_{q+k} &= \theta_z \quad (\text{or}) \quad \sum_{k=1}^r v_{q+k} = \theta_z \end{aligned} \quad (31)$$

If both conditions are met, the solution vector xx is complete and marks the terminal node. Terminal nodes represent either optimal or bounded suboptimal solutions and are essential in concluding viable paths in the solution tree.

Lower and upper bounds

To avoid unnecessary exploration, each feasible node is evaluated by computing bounds on the objective value. An upper bound is computed by relaxing the sparsity constraint and setting all unfixed variables to their upper limit.

$$\text{upper}(M, v) = \lambda_{\max}^*(v) \quad (32)$$

where $a = v$ denotes all the remaining variables active and λ_{\max}^* denotes the objective value. A lower bound is derived by greedily building a feasible solution within the sparsity constraint. Variables are selected based on descending importance measured from the first block and the second block until the sparsity limits θ_y and θ_z are met. The resulting vector a^{LB} is used to compute

$$\text{lower}(M, v) = \lambda_{\max}^*(a^{LB}) \quad (33)$$

This provides a valid bound from a feasible configuration. These bounds help eliminate suboptimal paths early and prioritize exploration of the most promising branches. Evaluating bounds is critical for maintaining computational efficiency while ensuring the search remains on track toward optimal solutions.

Re-evaluating the fitness

The optimization process repeatedly re-evaluates the LSTM weights and feature selections to minimize detection error and refine accuracy. This cycle continues until the model converges to the most optimal and globally accurate configuration.

```

Function BBO_Optimize(Weights):
  Initialize:
  Create root node with:
    Lower bound vector  $M = [0, \dots, 0]$ 
    Upper bound vector  $v = [1, \dots, 1]$ 
    Current solution vector  $X = \text{unset}$ 
  Set best_solution = None
  Set best_fitness =  $\infty$ 
  Initialize priority queue  $Q \leftarrow [\text{root node}]$ 
  While  $Q$  is not empty:
    # Pop node with the lowest lower bound
    node  $\leftarrow Q.\text{pop}()$ 
    If is_terminal(node):
      candidate_solution  $\leftarrow \text{generate\_solution}(\text{node}.X)$ 
      fitness  $\leftarrow \text{Fitness}(\text{candidate\_solution})$ 
      If fitness < best_fitness:
        best_fitness  $\leftarrow$  fitness
        best_solution  $\leftarrow$  candidate_solution
      Continue
    # Branching
    i  $\leftarrow \text{select\_unfixed\_variable}(\text{node})$ 
    for val in  $[0, 1]$ :
      child  $\leftarrow \text{copy}(\text{node})$ 
       $[0, 1] = X$ 
      child.X[i]  $\leftarrow$  val
      If is_feasible(child):
        lower  $\leftarrow \text{compute\_lower\_bound}(\text{child})$ 
        upper  $\leftarrow \text{compute\_upper\_bound}(\text{child})$ 

```

Table 6. Pseudocode for BBO.

Termination

The process terminates once the best solution is found that satisfies all constraints with minimal error. At this point, no further branching is needed, and the optimized weights are finalized for deployment. The optimized weights from the Boot LSTM model, refined using the BBO, are fed into SHAP for interpretability. SHAP analyzes the model predictions by attributing contributions to each input feature. This integration enhances transparency and trust in smart contract vulnerability detection. Table 6 shows the pseudocode for the BBO.

Theoretical convergence proof

The convergence of the BBO algorithm is theoretically guaranteed under standard assumptions. Specifically, BBO ensures global convergence when:

- The objective function is bounded below.
- The branching process partitions the feasible space exhaustively.

<p>If lower < best_fitness:</p> <p> child.lower_bound ← lower</p> <p> child.upper_bound ← upper</p> <p> Q.push(child)</p> <p>Return best_solution</p> <p>Function is_feasible(node):</p> <p> block1_count ← count_nonzero(node.X[block1])</p> <p> block2_count ← count_nonzero(node.X[block2])</p> <p> Return (block1_count ≤ S1) AND (block2_count ≤ S2)</p> <p>Function is_terminal(node):</p> <p> Return all variables in node.X are fixed OR</p> <p> node satisfies stopping condition</p> <p>Function compute_lower_bound(node):</p> <p> X_greedy ← greedily_select_variables(node)</p> <p> Return Fitness(X_greedy)</p> <p>Function compute_upper_bound(node):</p> <p> X_relaxed ← set_unfixed_to_1(node.X)</p> <p> Return Fitness(X_relaxed)</p> <p>Function generate_solution(X):</p> <p> Apply mask X to the original variable space</p> <p> Return resulting candidate solution</p> <p>Function Fitness(X):</p> <p> Evaluate the objective function for solution X</p> <p> Return the objective value</p> <p>End Function</p>
--

Table 6. (continued)

- The bounding functions compute valid lower and upper bounds for all subproblem.

The objective function is continuous and bounded. The pruning strategy in BBO eliminates suboptimal nodes using reliable bounds, and the branching process ensures exhaustive exploration of feasible regions. According to established theory, ensures that the optimal solution will eventually be found:

$$\lim_{u \rightarrow \infty} g_u^* = g^{opt} \quad (34)$$

where g_u^* denotes the best solution at iteration u , and g^{opt} denotes the global optimum. The stopping criterion $|VC - MC| \leq \epsilon$ ensures termination when a near-optimal solution is reached within acceptable tolerance. This theoretical foundation, together with the empirical evidence in Fig. 5, validates BBO's convergence properties for high-dimensional, constrained optimization tasks.

On the x-axis, generations are plotted, while the y-axis shows the corresponding objective function values. Initially, at generation 0, the objective function starts at approximately -9.5 , indicating a suboptimal configuration. By generation 3, the value improves to -10 , reflecting early-stage optimization progress. At generation 5, a sharper drop is observed as the value reaches -13.08 , suggesting that BBO is effectively pruning suboptimal branches and focusing on more promising solutions. By generation 10, the objective value stabilizes

between -13.05 and -12 , demonstrating strong convergence characteristics. This smooth and monotonic decline without significant oscillations highlights BBO's ability to avoid local minima and maintain a stable optimization path. The convergence trend also indicates that the BBO bounding mechanism effectively eliminates non-viable candidate's early, accelerating convergence. Compared to heuristic algorithms, BBO structured search ensures global optimality by thoroughly exploring feasible solutions. The absence of erratic spikes in the graph affirms the method's efficiency and stability. This behavior validates BBO's suitability for high-dimensional, constrained optimization tasks.

Explainable AI interpretation of SHAP

HB³LSTM intellectual engines, highly effective, are often hard to interpret because of their black-box nature, which makes it challenging to rely on their results for vulnerability detection. To enhance trust and usability in vulnerability detection, XAI techniques such as SHAP⁴² provide insights into feature contributions, improving transparency and model reliability. For complex models like DNN, Kernel SHAP approximates each feature's contribution using weighted linear regression. It employs a surrogate linear model that closely aligns with the original model's predictions, ensuring interpretability. The Shapley value for a feature quantifies its influence on the final prediction:

$$g(z') = \varphi_0 + \sum_{i=1}^N \varphi_i z'_i, \quad (35)$$

where z'_i indicates whether the feature i is present (1) or not (0); φ_i represents relative feature contribution using the Shapley value and φ_0 is the starting value in the event that no input features are present (0). By systematically analyzing feature combinations, SHAP assigns values to each input, highlighting their impact on predictions. Summing Shapley values across all instances:

$$I_i = \frac{1}{m} \sum_{j=1}^m |\phi_i^{(j)}| \quad (36)$$

where m denotes how many occurrences there are in the dataset. This method ensures model-agnostic explanations, maintaining consistency across different DL architectures.

To evaluate the interpretability of the HB³LSTM model, SHAP values were analyzed across multiple predictions. A consistent pattern emerged, showing that certain opcodes such as add, push1, call, and delegate call regularly ranked among the top contributors. These opcodes are closely associated with critical operations like arithmetic computations and external contract interactions, which are often targeted in common smart contract vulnerabilities. The repeated prominence of these opcodes in the SHAP rankings strongly indicates their significance in model decisions. Additionally, cross-referencing these findings with known vulnerability patterns from curated datasets confirmed that opcodes with higher Shapley values are indeed aligned with commonly exploited code fragments. This alignment reinforces the conclusion that SHAP not only facilitates interpretability but also effectively highlights security-relevant features that correlate with actual vulnerabilities.

Figure 6 shows the code for vulnerability identification logic of SHAP. Figure 7 illustrates the SHAP flowchart of proposed methodology, where feature-level explanations pinpoint the most significant attributes influencing

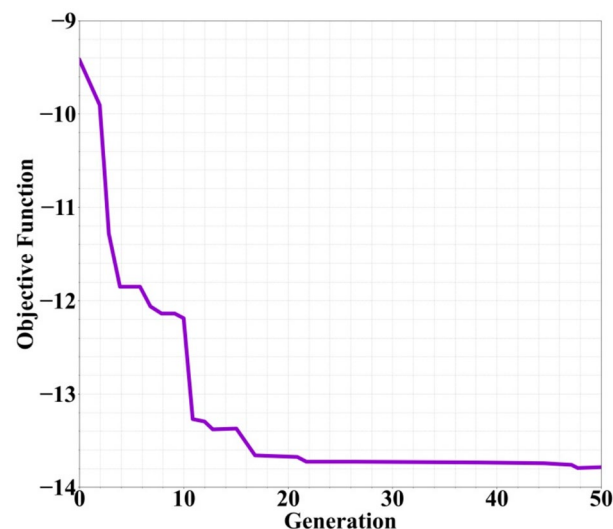


Fig. 5. Convergence characteristics of BBO.

vulnerability detection. This aids security teams in identifying root causes of attacks and prioritizing key metrics for faster detection with reduced false positives.

The methodology uses IQPO to optimize feature selection and NLP techniques to preprocess data in order to identify smart contract vulnerabilities. In addition to identifying dangerous transactions, a hybrid HB³LSTM intellectual engine extracts vulnerable code fragments. In order to evaluate results and identify certain vulnerability types, XAI approaches such as SHAP values are employed. TF-IDF and opcode extraction help with feature extraction, and LSTM manages sequence prediction. This method increases blockchain security and strengthens vulnerability detection. Table 7 shows the code presentation of the overall proposed approach.

Results and discussions

To address the generalizability of the proposed model, additional experiments were conducted using smart contracts developed for the Hyperledger fabric platform. These contracts, sourced from open-access repositories and executed within a simulated Fabric chain code environment, enabled comprehensive testing within a permissioned blockchain framework. To address key limitations of Ethereum, such as potential overfitting, the model was adapted and implemented in the Hyperledger environment. The experiments conducted on Hyperledger analyzed platform-specific performance metrics, including success rate, throughput, latency, and resource consumption. The results demonstrate that the proposed method maintains strong performance across these metrics, indicating effective generalization and adaptability for vulnerability detection on diverse blockchain platforms. The approach classifies vulnerability types with high accuracy by leveraging advanced DL techniques, implemented using Python's robust libraries and frameworks, and optimized Hyperledger ecosystems. Python's complex DL libraries and frameworks are utilized in the method, and the implementations are done on the Hyperledger platform. The quantum feature selection algorithm IQPO was implemented on IBM Qiskit's simulator with 8 qubits to represent feature vectors. Quantum operations such as norm and inner product estimations were executed in a noiseless simulation environment, enabling precise evaluation of feature importance rankings without noise interference. This simulation-based approach allowed us to verify IQPO's computational advantages and scalability before deployment on physical quantum hardware. The implementation runs on a Windows 10 operating system with Python 3.12.7, utilizing a 2.15 GHz processor and 1267 GB of RAM. Visual Studio Code is used as the development environment for executing the code. The experimental findings are presented in this section, along with an examination of how the method generates classifications that are highly accurate.

Dataset description

The SmartBugs Wild dataset is a large-scale collection of 47,331 Ethereum smart contracts, curated for the purpose of analyzing and detecting real-world security vulnerabilities. It is designed for use in training and testing machine learning and DL models aimed at smart contract vulnerability detection. The dataset includes both vulnerable and non-vulnerable contracts, providing balanced data for binary and multi-class classification tasks. A subset of the dataset, which includes over 1250 Solidity-written smart contracts, is annotated with specific vulnerability types such as reentrancy, integer overflow/underflow, and access control issues. These labeled contracts allow for fine-grained vulnerability classification and supervised learning. The dataset supports empirical evaluation of security analysis tools by offering realistic and diverse samples from the Ethereum

```
import shap
import numpy as np
# Assume 'model' is the trained HB3LSTM model and 'X' is the input data
# Prepare a prediction function for SHAP
def model_predict(x):
    return model.predict(x)
# Create the SHAP explainer
explainer = shap.KernelExplainer(model_predict, X)
# Compute SHAP values
shap_values = explainer.shap_values(X)
# Identify vulnerabilities
vulnerable_features = np.where(shap_values < 0)
# Print vulnerable features
print("Vulnerable features:")
for feature in vulnerable_features:
    print(feature)
# Visualize SHAP values
shap.summary_plot(shap_values, X)
```

Fig. 6. Code for vulnerability identification logic of SHAP.

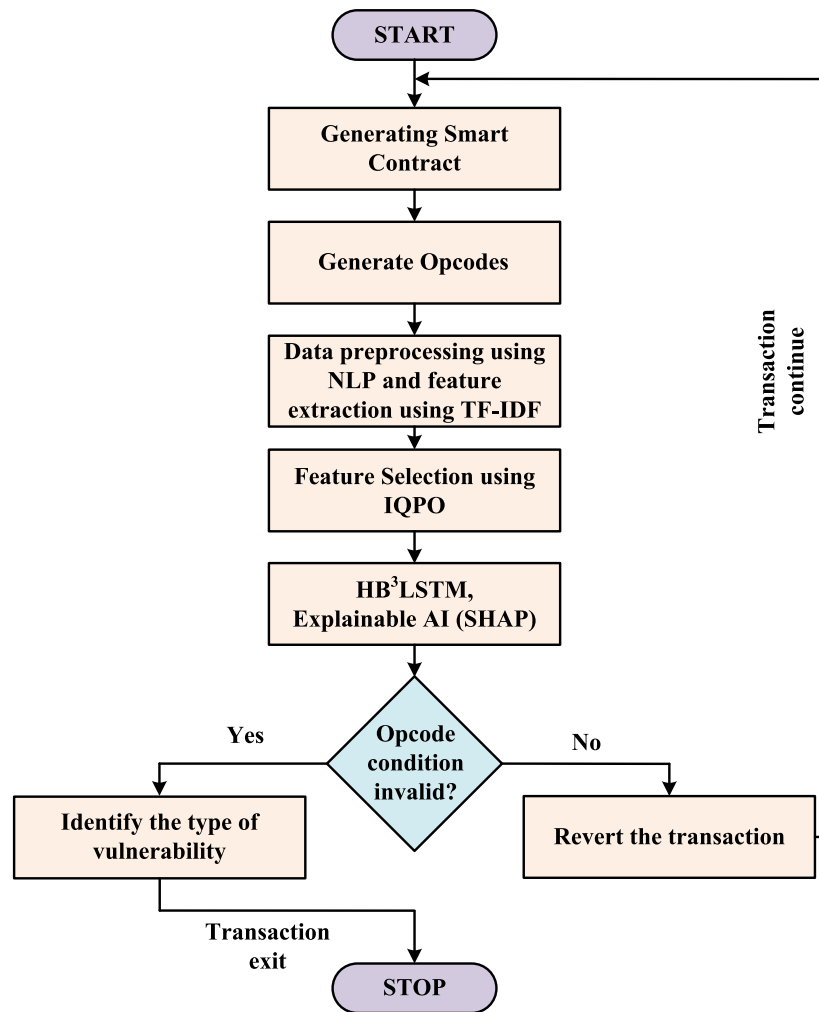


Fig. 7. Flowchart for proposed methodology.

network. It enables the training of robust models capable of identifying subtle and complex flaws in contract logic. The diversity and scale of the data also help reduce model overfitting and improve generalization. Researchers can use the dataset to benchmark detection techniques across various types of smart contract vulnerabilities. Ultimately, SmartBugs Wild aims to advance the security and reliability of blockchain applications through improved vulnerability detection methodologies.

Performance metrics

Key metrics for categorization models are compiled in Table 8. While memory measures the ability to recognize good instances, precision demonstrates the consistency of favorable predictions. While accuracy denotes complete correctness, the F1-score finds a balance between recall and precision. A higher AUROC rating indicates better class distinction. TP (True Positives), TN (True Negatives), FP (False Positives), FN (False Negatives), TPR (True Positive Rate), and FPR (False Positive Rate) are key terms that help compare model performances.

Exploratory data analysis

Table 9 contrasts different security tools based on how well they can identify smart contract vulnerabilities using the Smart Bugs Wild Dataset. It evaluates the accuracy, recall, F1-score, and FPR of every instrument for a variety of vulnerabilities. No single tool performs better than the others, even though Mythril and Oyente are excellent in terms of recall and precision for certain vulnerabilities. While Mythril frequently has superior precision but a higher FPR, Maian and Securify exhibit balanced performance with differing strengths in recall and F1-score.

Taxonomical analysis of vulnerabilities in hyperledger fabric using HB³LSTM

To enhance vulnerability detection in Hyperledger Fabric, a refined taxonomy based on the HB³LSTM model is proposed, focusing on actual exploit types rather than architectural layers. The detection process identifies key vulnerability classes, including reentrancy, where repeated function calls compromise contract state, and access control issues due to improper identity verification. DoS vulnerabilities are flagged when resource exhaustion or execution blocking is detected. Arithmetic errors, such as overflows or underflows, are also recognized as

```

import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Bidirectional
import shap
from sklearn.metrics import accuracy_score
from sklearn.feature_selection import SelectKBest, mutual_info_classif
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
# Load Smart Bugs Wild Dataset
# Assume 'data' is a pandas DataFrame with contract code and labels
# Define a function for preprocessing
def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()
    # Remove special characters and digits
    text = re.sub(r'^[a-zA-Z\s]!', '', text)
    # Tokenize the text
    tokens = word_tokenize(text)
    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]
    # Join the tokens back into a string
    text = ' '.join(tokens)
    return text
# Apply the preprocessing function to the dataset
data['code'] = data['code'].apply(preprocess_text)
# Define a TF-IDF vectorizer
vectorizer = TfidfVectorizer()
# Fit the vectorizer to the preprocessed data and transform it into a matrix

```

Table 7. Open code Presentation of Proposed Approach.

critical risks. The model detects unchecked low-level calls, which may lead to unexpected behaviors and time manipulation vulnerabilities arising from misuse of temporal data. Front-running is identified when transaction ordering is exploited for gain. These categories are supplemented by other miscellaneous vulnerabilities representing less common yet dangerous flaws. This taxonomy enhances threat classification and interpretability, ensuring HB³LSTM alignment with practical security challenges in Fabric-based smart contracts.

```

X = vectorizer.fit_transform(data['code'])
y = data['label']
# Feature selection using IQPO (simplified example)
selector = SelectKBest(mutual_info_classif, k=1000)
X_selected = selector.fit_transform(X, y)
# Split data
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2,
random_state=42)
# Reshape data for LSTM
X_train = X_train.toarray().reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.toarray().reshape(X_test.shape[0], X_test.shape[1], 1)
# HB3LSTM model
model = Sequential()
model.add(Bidirectional(LSTM(64, return_sequences=True),
input_shape=(X_train.shape[1], 1)))
model.add(Bidirectional(LSTM(32)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Train model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))
# Evaluate model
y_pred = model.predict(X_test)
y_pred_class = (y_pred > 0.5).astype('int32')
print("Accuracy:", accuracy_score(y_test, y_pred_class))
# XAI using SHAP
explainer = shap.DeepExplainer(model, X_train)
shap_values = explainer.shap_values(X_test)
# Visualize SHAP values
shap.force_plot(explainer.expected_value[0], shap_values[0], X_test[0])

```

Table 7.. (continued)

Parameter	Formula
Accuracy	$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \times 100$
Precision	$Precision = \frac{TP}{TP+FP} \times 100$
Recall	$Recall = \frac{TP}{TP+FN} \times 100$
F1-score	$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall}$
Area_Under_ROC (AUROC)	$TPR = \frac{TP}{FP+FN}$ $FPR = \frac{FP}{FP+TN}$

Table 8. Binary classification and performance metrics.

Vulnerabilities	Tools	Precision	Recall	F1-score	FPR
Time manipulation	Mythril	0.94	0.79	0.90	0.61
	Slither	0.86	0.91	0.78	0.86
	Oyente	0.93	0.92	0.92	0.54
	Osiris	0.86	0.76	0.97	0.77
	Smartcheck	0.72	0.86	0.87	0.65
	Manticore	0.85	0.91	0.79	0.76
	Maian	0.92	0.89	0.91	0.24
	Securify	0.78	0.80	0.89	0.67
	Honeybadger	0.81	0.92	0.78	0.45
Front running	Mythril	0.79	0.93	0.95	0.64
	Slither	0.94	0.89	0.78	0.82
	Oyente	0.90	0.91	0.89	0.56
	Osiris	0.72	0.89	0.75	0.62
	Smartcheck	0.89	0.75	0.90	0.54
	Manticore	0.83	0.79	0.73	0.76
	Maian	0.76	0.82	0.79	0.65
	Securify	0.79	0.90	0.93	0.84
	Honeybadger	0.91	0.82	0.91	0.76
DoS	Mythril	0.90	0.89	0.74	0.64
	Slither	0.78	0.75	0.91	0.65
	Oyente	0.79	0.90	0.88	0.87
	Osiris	0.90	0.82	0.80	0.68
	Smartcheck	0.82	0.79	0.75	0.76
	Manticore	0.90	0.72	0.79	0.87
	Maian	0.92	0.94	0.83	0.43
	Securify	0.89	0.92	0.92	0.73
	Honeybadger	0.93	0.83	0.88	0.84
Reentrancy	Mythril	0.73	0.76	0.87	0.56
	Slither	0.89	0.83	0.94	0.69
	Oyente	0.81	0.92	0.70	0.76
	Osiris	0.87	0.90	0.85	0.54
	Smartcheck	0.92	0.76	0.89	0.65
	Manticore	0.86	0.89	0.90	0.84
	Maian	0.84	0.92	0.92	0.54
	Securify	0.93	0.80	0.76	0.74
	Honeybadger	0.80	0.82	0.72	0.85
Access control	Mythril	0.89	0.76	0.78	0.56
	Slither	0.90	0.82	0.90	0.74
	Oyente	0.74	0.88	0.92	0.81
	Osiris	0.79	0.76	0.76	0.55
	Smartcheck	0.83	0.90	0.87	0.82
	Manticore	0.90	0.89	0.73	0.88
	Maian	0.79	0.72	0.79	0.75
	Securify	0.87	0.80	0.80	0.77
	Honeybadger	0.88	0.76	0.88	0.84
Arithmetic	Mythril	0.78	0.91	0.87	0.80
	Slither	0.90	0.78	0.88	0.63
	Oyente	0.86	0.85	0.90	0.87
	Osiris	0.82	0.91	0.78	0.76
	Smartcheck	0.92	0.70	0.94	0.65
	Manticore	0.79	0.89	0.76	0.80
	Maian	0.91	0.72	0.92	0.56
	Securify	0.92	0.80	0.80	0.52
	Honeybadger	0.76	0.91	0.82	0.76
Continued					

Vulnerabilities	Tools	Precision	Recall	F1-score	FPR
Unchecked low calls	Mythril	0.87	0.92	0.77	0.89
	Slither	0.92	0.79	0.80	0.80
	Oyente	0.76	0.89	0.93	0.67
	Osiris	0.79	0.76	0.74	0.46
	Smartcheck	0.90	0.70	0.90	0.88
	Manticore	0.82	0.88	0.87	0.67
	Maian	0.89	0.86	0.73	0.49
	Securify	0.78	0.92	0.92	0.81
	Honeybadger	0.91	0.81	0.80	0.69
Others	Mythril	0.91	0.80	0.89	0.84
	Slither	0.78	0.85	0.70	0.87
	Oyente	0.72	0.92	0.76	0.49
	Osiris	0.79	0.87	0.88	0.77
	Smartcheck	0.93	0.75	0.92	0.67
	Manticore	0.87	0.80	0.74	0.80
	Maian	0.82	0.73	0.83	0.66
	Securify	0.80	0.77	0.89	0.67
	Honeybadger	0.73	0.89	0.84	0.59

Table 9. Comparison of different tools in smart bugs wild dataset.

Case study

Protect Decentralized Finance (DeFi) platforms by scanning their deployed smart contracts for vulnerabilities using the dataset.

Real-Time Application: Analyze the blockchain mempool (where unconfirmed transactions reside) to prevent attacks like front running.

Confirmed Example:

A mempool monitor detects a front running attack where an attacker tries to outbid a trade by submitting a transaction with a higher gas fee.

Steps to Apply Dataset:

- *Train a model* Use the dataset to train a model that identifies malicious transaction patterns, such as excessive gas usage in attacks.
- *Deploy on nodes* Run the model on Ethereum nodes to monitor transactions in the mempool.
- *Block malicious transactions* Flag and prevent suspicious transactions from being confirmed.

Dataset analysis

Figure 8 presents the distribution of vulnerability classes in the SmartBugs Wild Dataset, visualized using a pie chart. With 30.4% of the total, “arithmetic” is the largest sector, followed by “other” with 22.9%. “Unchecked_low_calls” and “reentrancy” are about equal, at about 11.8% and 11.9%, respectively. “Time manipulation” is 3.3%, “front running” is 6.6%, “access control” is 3.1%, and “denial service” is 10.0%. This graph helps prioritize security efforts by highlighting the frequency of various vulnerabilities.

The feature importance graph for the Smart Bugs Wild Dataset illustrates the significance of various features in the dataset. The x-axis represents the feature importance, ranging from 0.00 to 0.12, while the y-axis lists the features being evaluated. The features, from highest to lowest importance, are add, and, 0×0, dup2, push1, dup1, jumpdest, pop, swap1, and push2. Each feature has a corresponding horizontal bar indicating its importance, with error bars showing the uncertainty or variability in the measurement. The feature “add” has the highest importance, followed by “and” and “0×0,” while “push2” has the lowest importance among the listed features. This Fig. 9 helps in understanding which features are most significant in the Smart Bug Wild Dataset, potentially guiding further analysis or model development.

The features in Smart Bugs Wild Dataset’s ROC curve illustrates how well classifiers work in identifying smart contract vulnerabilities, as described in Fig. 10. For different classifiers, it compares the true positive rate against the false positive rate. Effectiveness is indicated by the Area Under the Curve (AUC) values; greater values signify superior performance. The capacity of classifiers to differentiate between smart contracts that are vulnerable and those that are not is compared using this curve.

Figure 11 shows the confusion matrix for the SmartBugs. Wild Dataset shows the model’s performance in identifying smart contract vulnerabilities. The actual class is represented by each row, while the anticipated class is represented by each column. The model accurately detected instances of arithmetic, front-running, denial-of-service, time-manipulation, unchecked-low-calls, reentrancy, and access-control vulnerabilities. Better performance is indicated by higher diagonal values, which also reveal areas for improvement and areas of strength.

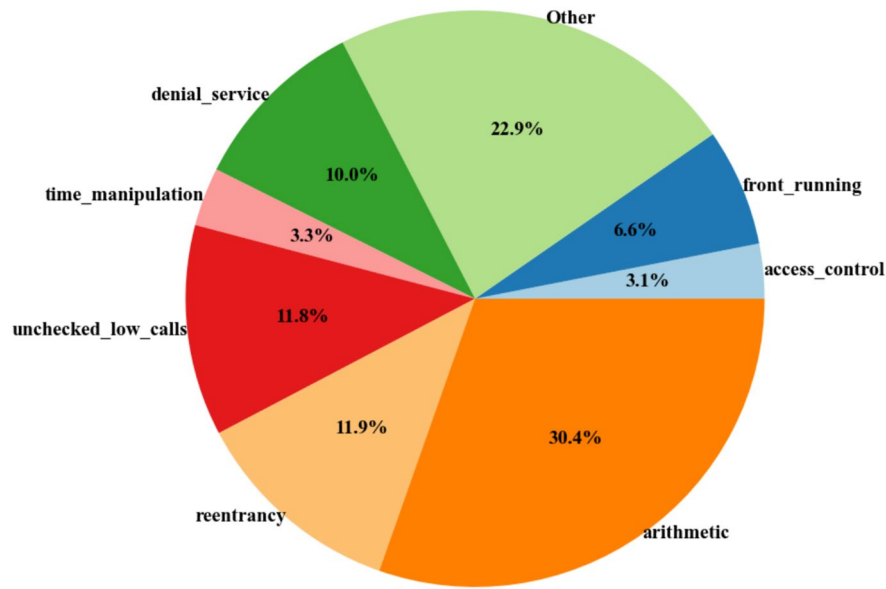


Fig. 8. Class distribution in the smart bugs wild dataset.

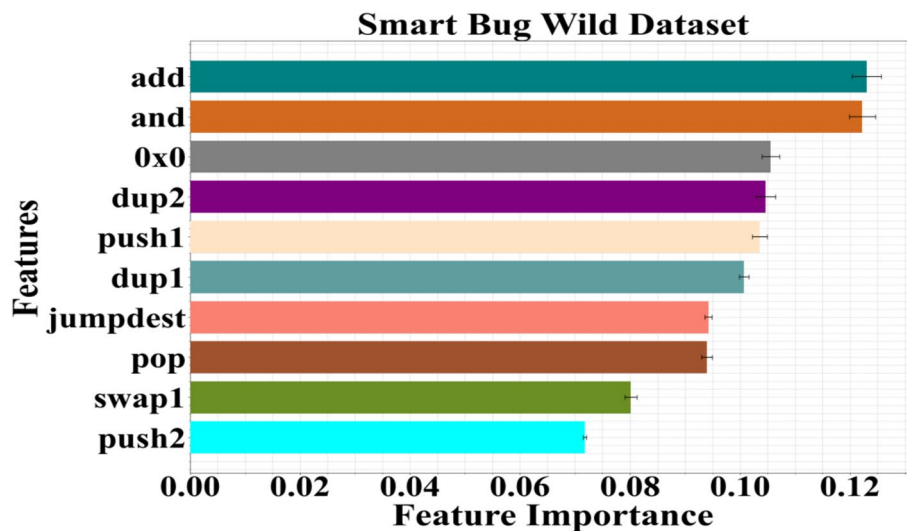


Fig. 9. Feature importance graph for smart bugs wild dataset.

Figure 12 shows the performance of the model in terms of accuracy and loss over 100 training epochs for the task of vulnerability detection. In (a) presents the accuracy trends, where the training accuracy reaches approximately 0.98, the testing accuracy stabilizes around 0.86, and the validation accuracy achieves about 0.78. (b) Shows the corresponding loss values, with the training loss at approximately 0.38, testing loss around 0.28, and validation loss reaching as low as 0.17. These results indicate good generalization ability of the model, with a slight performance drop in the validation phase suggesting potential areas for further optimization.

Figure 13 shows the computational efficiency of the proposed HB³LSTM model compared to existing methods in terms of the number of nodes processed and their corresponding running time. The existing methods like EA-RGCN²⁷, WIDENET³¹, and DA-GNN³³ demonstrate a running time of approximately 10⁴ units over 500 nodes and higher. While the proposed HB³LSTM model significantly outperforms these methods with a substantially reduced running time of approximately 10⁻¹, indicating its superior computational efficiency and scalability in handling large-scale data. This improvement is primarily attributed to the integration of BBO into the training phase of the Boot LSTM network. Within the computational graph, BBO replaces traditional gradient-based optimization by systematically evaluating candidate weight configurations to minimize detection error. By computing upper and lower bounds on the objective function, BBO prunes suboptimal paths and focuses computation on promising regions of the search space. This intelligent pruning strategy not only

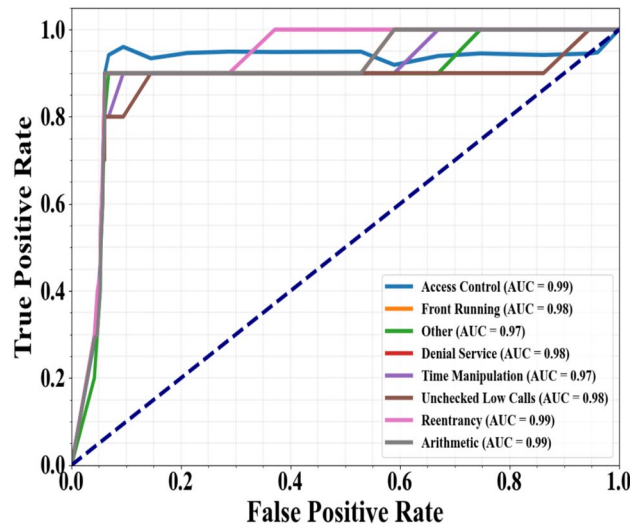


Fig. 10. ROC curve of classes in smart bugs wild dataset.

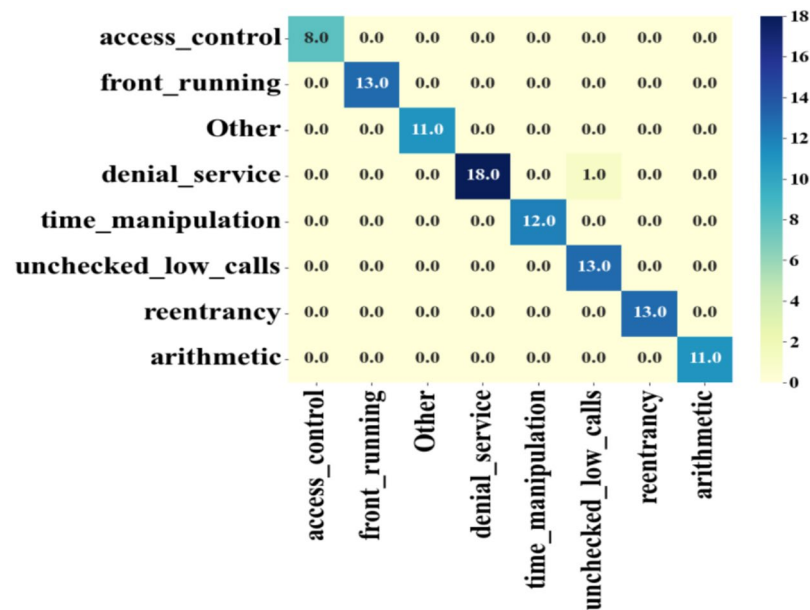


Fig. 11. Confusion matrix of smart bugs wild dataset.

improves convergence and accuracy but also drastically reduces computational overhead, contributing to the superior performance.

Figure 14 presents a SHAP summary plot illustrating the most influential TF-IDF features in predicting vulnerabilities in smart contracts from the Smart Bugs Wild dataset. Features such as “push1,” “and,” “jumpdest,” and “swap1” exhibit the highest mean SHAP values, indicating strong contributions to the model’s classification outcomes. Specifically, high SHAP values for “jumpdest” correlate with control-flow vulnerabilities like reentrancy, while “push1” is tied to arithmetic flaws such as overflows. The x-axis represents the mean SHAP values, and the color gradient indicates the original feature values, aiding in interpretability. “dup2” shows a consistent positive impact on vulnerability detection, whereas features like “add” and “and” demonstrate mixed influence. The plot provides a quantitative lens into not just feature importance but also their directional impact. SHAP thus enhances the transparency of complex models, which ensures explainable predictions. Overall, the analysis validates SHAP’s role in highlighting security-critical patterns in smart contract code.

Table 10 shows the statistical validation of SHAP feature contributions for key opcodes in HB³LSTM based vulnerability detection. The mean SHAP values for each opcode, indicating their average contribution to the model’s predictions. The opcode call has a mean SHAP value of 0.084 with a t-value of 6.41 and a highly significant p-value (< 0.001), alongside a large effect size d = 1.25, highlighting its strong influence. The opcode delegatecall has the mean of 0.079, t = 5.87, p < 0.001, and d of 1.15, and selfdestruct has the mean value of 0.075, t = 5.42,

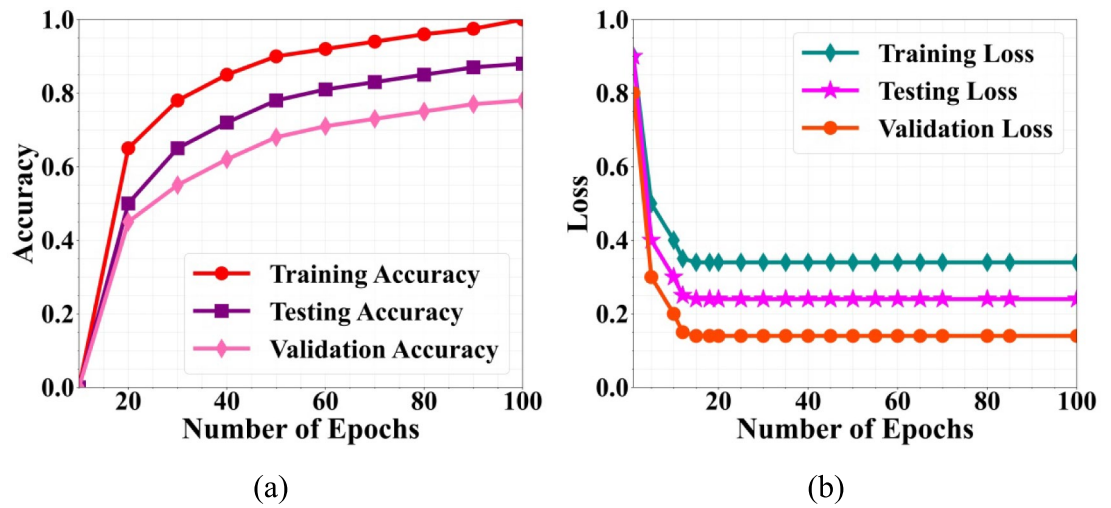


Fig. 12. Performance evaluation of training, testing, and validation phases. (a) accuracy versus epochs (b) loss versus epochs.

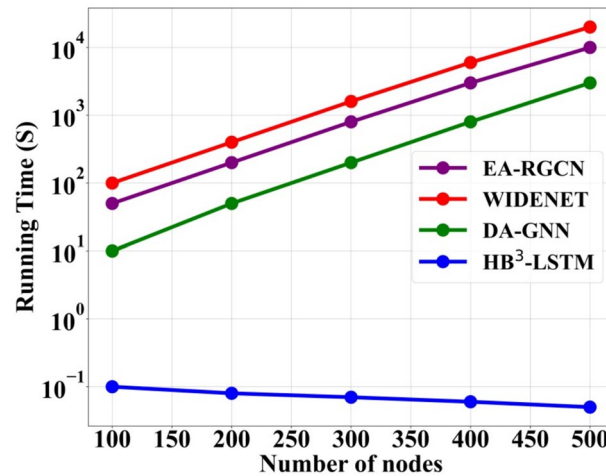


Fig. 13. Computational efficiency comparison of the proposed model.

$p < 0.001$, and $d = 1.05$, also showing significant positive contributions. The add opcode contributes moderately with a mean SHAP value of 0.067, $t = 4.22$, $p = 0.0002$, and effect size $d = 0.89$. In contrast, opcodes like jumpdest have the mean of 0.014, $t = 1.13$, $p = 0.26$, and $d = 0.21$; log0 has the mean of 0.011, $t = 0.97$, $p = 0.33$, and $d = 0.18$; and revert has the mean of 0.008, $t = 0.42$, $p = 0.68$, and $d = 0.10$ have lower mean values and non-significant p -values, indicating weaker or inconsistent influence. The statistical tests confirm that most key opcodes contribute meaningfully to vulnerability detection, reinforcing the model's interpretability via SHAP. Overall, this analysis validates the importance of these features and supports the reliability of the HB³LSTM model in identifying smart contract vulnerabilities.

Figure 15 shows the performance comparison of the proposed HB³LSTM model on original and adversarial smart contracts using key evaluation metrics. The model achieved high values on the original dataset, with an accuracy of 0.97, precision of 0.98, recall of 0.99, and F1-score of 0.98, indicating excellent predictive capabilities. In contrast, the performance significantly declined on adversarial contracts, where accuracy dropped to 0.85, precision to 0.75, recall to 0.7, and F1-score to 0.65. This stark performance gap highlights the robustness of the original model against data perturbations. It also confirms the effectiveness of the BBO algorithm in optimizing the model's weights.

Comparison analysis

The accuracy of three distinct approaches, like Hierarchical Attention Network (HAN), Dual-Channel Convolutional Neural Network (DC-CNN), and the suggested HB³LSTM across tenfold cross-validation, is shown in the graph in Fig. 16. The accuracy is shown on the y-axis, which ranges from 0.60 to 1.00, and the number of folds is shown on the x-axis, which ranges from 1 to 10. The accuracy of the suggested HB³LSTM approach

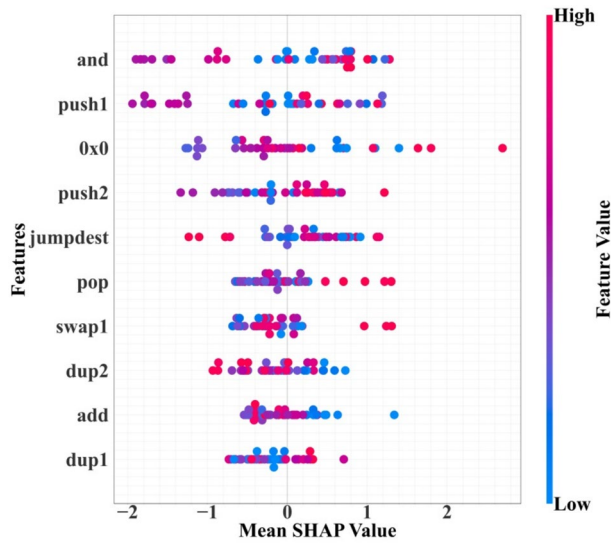


Fig. 14. SHAP summary plot for smart bugs wild dataset.

Opcode	Mean SHAP value	t-value	p-value	Effect Size (Cohen's d)
Call	0.084	6.41	<0.001	1.25
Delegatecall	0.079	5.87	<0.001	1.15
Add	0.067	4.22	0.0002	0.89
Push1	0.0059	3.78	0.001	0.73
Jumpdest	0.014	1.13	0.26	0.21
Log0	0.011	0.97	0.33	0.18
Revert	0.008	0.42	0.68	0.10
Self_destruct	0.075	5.42	<0.001	1.05

Table 10. Statistical validation of SHAP feature contributions for key opcodes in HB³LSTM based vulnerability detection.

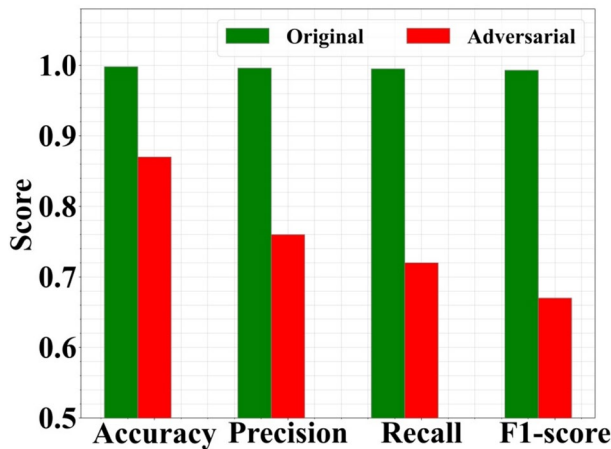


Fig. 15. Comparative performance metrics of HB³LSTM on original vs. adversarial smart contracts.

is higher across all folds, consistently outperforming the other two techniques. This superior performance of HB³LSTM can be attributed to its hybrid architecture, which combines the strengths of Boot-LSTM with BBO. The LSTM component enables effective sequence learning and temporal pattern recognition, which is particularly beneficial for modeling contextual dependencies in sequential data. BBO adaptively optimizes model parameters, enhancing convergence and generalization. Furthermore, the hierarchical structure of

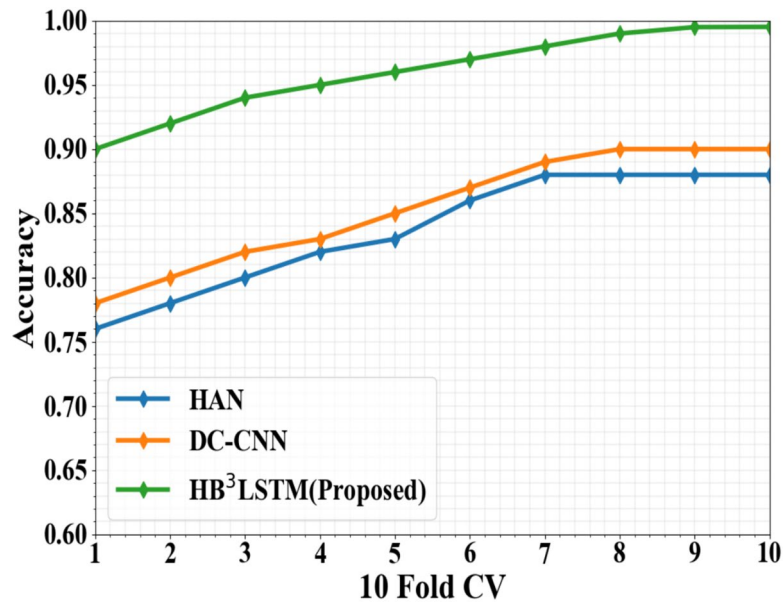


Fig. 16. Cross-validation of existing and proposed methods.

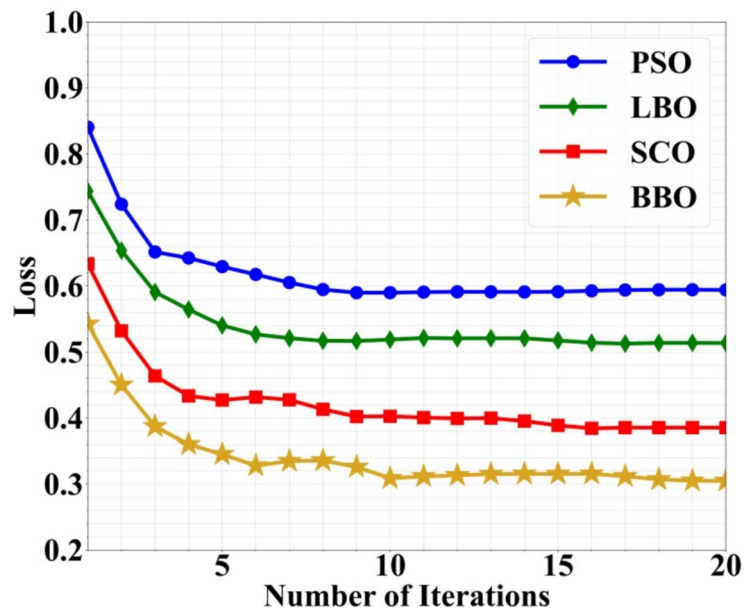


Fig. 17. Comparative convergence analysis of optimization algorithms.

HB³LSTM allows it to capture both local and global features more effectively than the flat architectures of HAN and DC-CNN. This integrated and adaptive design leads to more robust learning and explains the consistently higher accuracy observed across all folds.

Figure 17 shows the comparative convergence behavior analysis of four optimization algorithms, like Particle Swarm Optimization (PSO), Lyrebird Optimization (LBO), Single Candidate Optimization (SCO), and the BBO, over 20 iterations. The y-axis denotes the loss values, while the x-axis represents the number of iterations. Among the existing methods, PSO has the loss of 0.85, LBO has the loss of 0.75, and SCO has the loss of 0.65, respectively. While the proposed BBO achieves the lowest loss of 0.55, indicating faster and more stable convergence. This superior performance is primarily due to BBO's structured approach to optimizing the weight parameters of the Boot LSTM network. Unlike population-based methods like PSO and LBO, which often suffer from premature convergence due to limited global coordination. But BBO employs bounding and pruning techniques. These methods eliminate suboptimal regions of the solution space early in the search, improving both convergence speed and accuracy while reducing the risk of getting trapped in local minima.

Compared to SCO, which explores a single solution path at a time, BBO maintains a strategic balance between exploration and exploitation. This balance is particularly valuable in high-dimensional spaces, such as those found in smart contract analysis, where solution landscapes are often complex and filled with deceptive local optima. Overall, BBO's systematic and scalable search strategy enhances both the robustness and precision of the optimization process, making it highly effective in minimizing loss and improving the detection of smart contract vulnerabilities.

Figure 18 shows the computational analysis of BBO over the existing algorithms, addressing concerns related to computational placement and efficiency. The x-axis reflects the number of function evaluations, while the y-axis presents the function error value on a logarithmic scale. The proposed BBO algorithm maintains a consistent error around 10^0 , indicating it rapidly converges and stabilizes early in the optimization process. This stability demonstrates where BBO fits in the computational graph as an efficient optimizer with bounded weight updates. Unlike PSO, which continues to decrease error to below 10^{-2} , and LBO, which reaches around 10^{-1} , BBO emphasizes early convergence and conservative updates. These bounded behaviors are a result of adaptive limits placed on neural weight changes, implicitly acting as a pruning mechanism. Therefore, BBO contributes to computational efficiency by enforcing stability and reducing unnecessary exploration. This reflects its role in constraining weight space evolution and potentially aiding in implicit network pruning.

Table 11 presents a comparative analysis of existing methods against the proposed HB3LSTM model using key performance metrics. Among static-based tools, Mythril shows the lowest accuracy at 39.48% and F1-score at 37.04%, while Oyente has 70.04% accuracy and a 59.41% F1-score. DL models demonstrate superior performance, with HAN achieving 94.76% accuracy and a 96.29% F1-score and a DC-CNN further improving to 96.89% accuracy and a 97.64% of F1-score. The LLM, like CodeBERT, has an accuracy of 85.42%, a precision of 75.23%, recall of 69.45, and an F1-score of 7.32%. FinBERT has the accuracy of 82.45%, precision of 79.73%, recall of 69.06%, and F1-score of 70.26%. The proposed HB3LSTM outperforms all with 99.34% accuracy, 99.52% precision, 99.28% recall, and a 99.13% F1-score. This clearly highlights the effectiveness of HB3LSTM in vulnerability detection.

Figure 19 shows the comparative performance of the proposed HB³LSTM model against existing LLMs based on the accuracy metric. The existing models like the N-Gram²⁷ model achieved an accuracy of 40.87%, CodeBERT with 56.89%, and FinBERT with 78.35%. In contrast, the HB³LSTM model demonstrated a remarkable accuracy of 98.79%, significantly outperforming all baseline models. This superior performance is attributed to the HB³LSTM hybrid architecture that combines the optimization strength of the BBO algorithm with the sequential learning capabilities of LSTM networks. The BBO component efficiently navigates complex search spaces to enhance model precision, while LSTM excels at capturing temporal dependencies in transaction sequences. Together, they enable HB³LSTM to detect smart contract vulnerabilities more effectively. Overall, HB³LSTM offers a robust, scalable solution for identifying potentially malicious blockchain transactions.

Figure 20 shows a performance comparison between Ethereum²⁷ and Hyperledger across four key metrics. In (a), the success rate shows that Ethereum achieved 40% in Test Case (TC) 1, 20% in TC 2, 65% in TC 3, and 60% in TC 4. In contrast, Hyperledger showed much higher success rates of 96% in TC 1, 93% in TC 2, 97% in TC 3, and 96% in TC 4 in the respective test cases. In (b), the throughput shows. Ethereum recorded 50 transactions per second in Scenario (S) 1, 70 in S2, 62 in S3, and 78 in S4. Hyperledger outperformed with 65 in S1, 95 in S2, 90 in S3, and 91 transactions per second, respectively. In (c), the latency shows that Ethereum had values

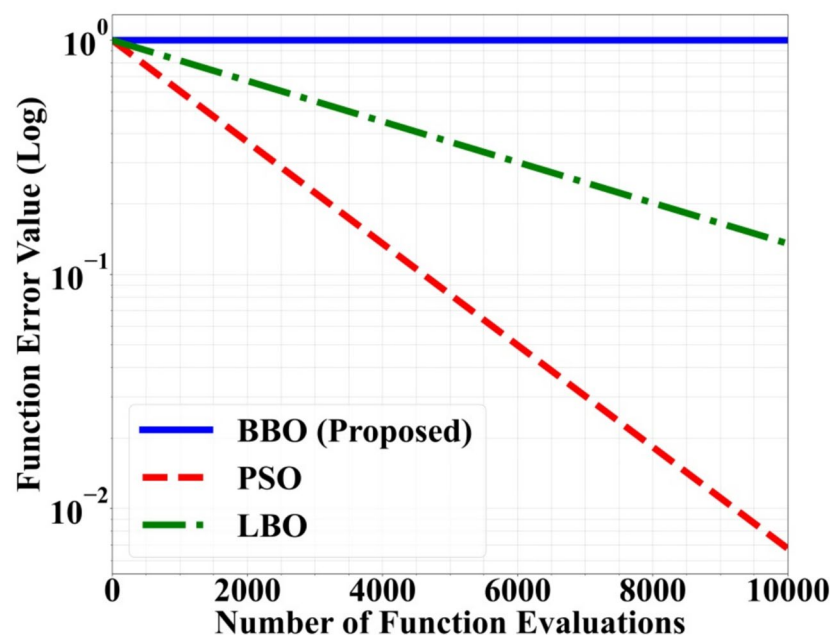


Fig. 18. Computational analysis of BBO.

	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Statics-based tools				
Mythril ³⁴	39.48	55.58	42.25	37.04
Oyente ²⁷	70.04	45.62	61.23	59.41
DL models				
HAN ²⁵	94.76	95.65	96.43	96.29
DC- CNN ³⁴	96.89	96.45	97.23	97.64
Large language models (LLM)				
CodeBERT ³⁴	85.42	75.23	69.45	71.32
FinBERT ³⁵	82.45	79.73	69.06	70.26
HB ³ LSTM	99.34	99.52	99.28	99.13

Table 11. Dataset comparison of existing methods with the proposed model.

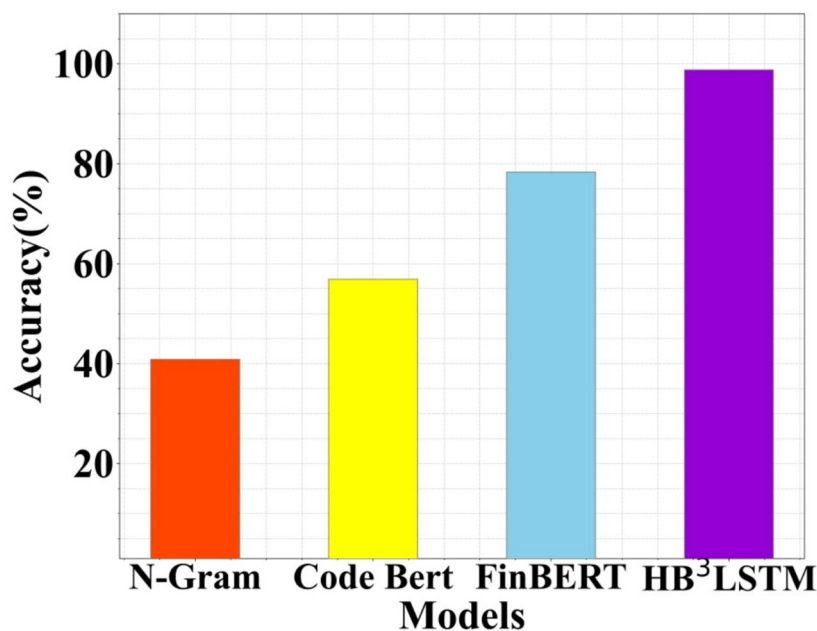


Fig. 19. Performance comparison of the proposed HB³LSTM model with existing LLM based on accuracy.

of 75 ms, 60, 68, and 70 across the scenarios, while Hyperledger maintained lower latency at 92, 90, 90, and 87 ms. In (d), it shows the resource utilization; Ethereum used 200% CPU and 900 Mega Bytes (MB) of memory, whereas Hyperledger consumed only 120% CPU and 550 MB of memory, showcasing its higher efficiency. Table 12 shows the example code for SHAP.

Figure 21 illustrates the cross-platform performance evaluation of the proposed HB³LSTM model across four operating systems: Windows, Linux, Ubuntu, and macOS. The model achieved the highest accuracy on Ubuntu at 99.8%, followed closely by macOS at 99.7%, Linux at 99.2%, and Windows at 98.5%. In terms of memory usage, Ubuntu again performed best, consuming only 260 MB, followed by Linux with 280 MB, macOS at 300 MB, and Windows using the most at 320 MB. Throughput analysis revealed that Ubuntu also led in request handling, achieving 1750 requests/sec, with Linux at 1700, macOS at 1600, and Windows trailing at 1500. These results underscore Ubuntu's superior efficiency across all three metrics. Linux and macOS also showed competitive performance, while Windows lagged slightly in comparison. This cross-platform consistency highlights the robustness and portability of the proposed model.

Ablation study

Table 13 presents an ablation study highlighting the performance of various existing methods. EA-RGCN achieved 90.47% accuracy, 91.16% precision, 89% recall, and a 90.03% F1-score. BERT showed 92.53% accuracy, 94.21% precision, 95.77% recall, and 89.47% F1-score. ANFIS reached 96.4% accuracy, 99.4% precision, 96% recall, and a 99.1% F1-score. CNN-LSTM obtained 91% accuracy, 92% precision, 99% recall, and 95% F1-score. While these methods deliver competitive results, our proposed HB³LSTM with Explainable AI significantly outperforms them all, achieving 99.68% accuracy, 99.43% precision, 99.54% recall, and 99.40% F1-score. The proposed method outperforms established baselines like CNN-LSTM and BERT. The proposed HB³LSTM + Explainable

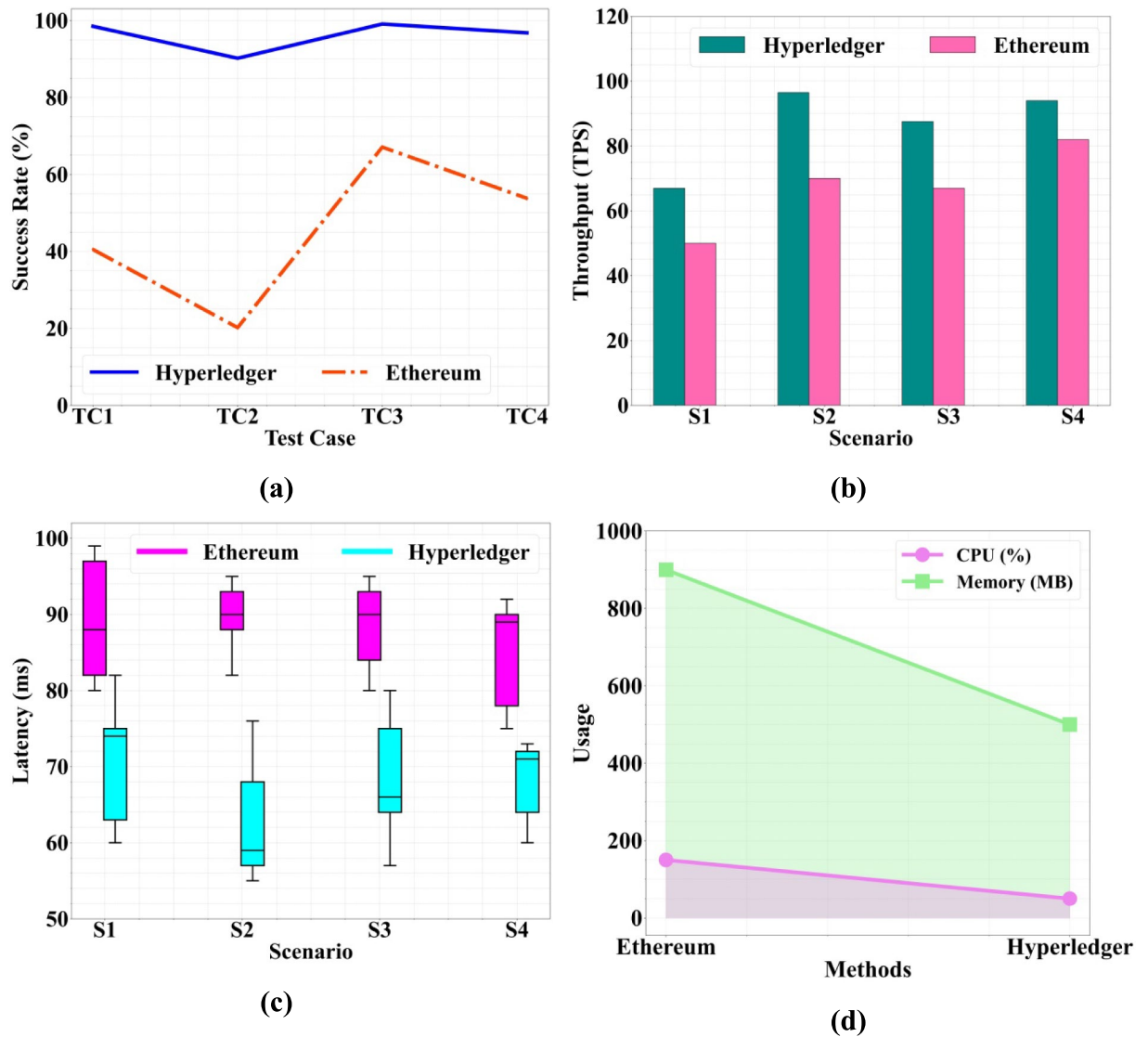


Fig. 20. Comparative metrics analysis between ethereum and hyperledger platforms. (a) success rate, (b) throughput, (c) latency, (d) resource utilization.

AI method demonstrates superior performance compared to other methods, highlighting its potential for effective vulnerability detection. The results suggest that the combination of HB³LSTM and Explainable AI provides a robust and accurate approach for identifying vulnerabilities.

Discussion

Our suggested approach improves the detection of smart contract vulnerabilities in several steps. First, NLP methods are used to preprocess the input. The significance of important terms in the smart contract is then captured by applying TF-IDF for feature extraction. Improved Quantum Online Portfolio scalability is used for feature selection. In order to find vulnerabilities and stop dangerous transactions, the hybrid HB³LSTM then extracts important code fragments. Lastly, the model's decisions are interpreted using XAI approaches, such as SHAP values, which show how particular aspects affect the vulnerability or invulnerability of a contract. This method increases detection accuracy while offering insightful information about the security of smart contracts.

To further enhance the robustness of smart contract vulnerability detection systems, future work can explore adversarial training strategies. Adversarial examples are inputs deliberately crafted to deceive the model, which poses significant threats to the reliability of security classifiers. Incorporating adversarial training could increase model resilience by exposing it to potential attack vectors during the learning phase. The integrating modular defense mechanisms, such as runtime behavior analyzers or transaction simulation engines, could serve as an added layer of protection. These modules can act in tandem with the static analysis approach, offering both proactive and reactive security coverage. Another promising direction is the fusion of semantic-aware graph representations with HB³LSTM to better model code dependencies and interactions. Combining this with adversarial robust training regimes may yield systems that are both interpretable and resistant to manipulation.

```
# Assume 'X' is the input data and 'shap_values' are the SHAP values computed for
the HB3LSTM model

# Identify the contribution of the "add" opcode to vulnerability
add_opcode_index = X.columns.get_loc('add')

add_opcode_shap_value = shap_values[:, add_opcode_index]

# Check if the "add" opcode contributes to vulnerability
if np.any(add_opcode_shap_value < 0):

    print("The 'add' opcode contributes to vulnerability in some samples")

# Further analysis can be done to identify specific samples or contracts
else:

    print("The 'add' opcode does not contribute to vulnerability")
```

Table 12. Example code for SHAP.

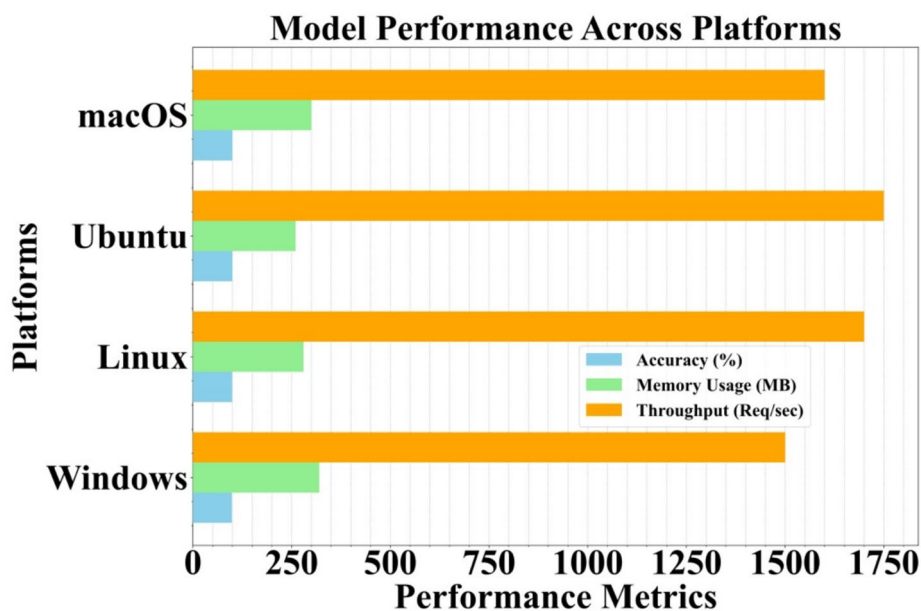


Fig. 21. Cross platform performance evaluation.

Methods	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Edge attention-residual graph convolutional networks (EA-RGCN) ²⁷	90.47	91.16	89	90.03
BERT ³⁴	92.53	94.21	95.77	89.47
Adaptive neuro-fuzzy inference system (ANFIS) ²⁶	96.4	99.4	96	99.1
CNN-LSTM ⁴³	91	92	99	95
HB ³ LSTM + explainable AI (proposed)	99.68	99.43	99.54	99.40

Table 13. Ablation study of proposed method.

Extending explainability modules to incorporate user-friendly visualizations could facilitate broader adoption by auditors and developers who may not be familiar with machine learning outputs but require actionable insights.

Conclusion

In the final analysis, by comparing several approaches, the suggested methodology successfully improves smart contract vulnerability identification. The method uses IQPO for effective feature selection, cutting down on processing time and improving scalability after preprocessing the data and extracting features using NLP and TF-IDF. By identifying important code segments and preventing dangerous transactions, the novel HB³LSTM enhances detection even further and works in combination with expert models. The final stage, which uses XAI with SHAP values, allows for the identification of critical opcodes that impact smart contract vulnerabilities and transparent decision-making. A reliable, scalable, and interpretable system for identifying and reducing smart contract risks is produced by this multi-step process.

However, adversarial testing reveals certain robustness gaps in the current model, suggesting the need for proactive defense mechanisms. Future research can explore the integration of adversarial training and lightweight defense modules to improve resilience against sophisticated evasion attempts. Additionally, extending the system to operate within decentralized or federated environments can enhance both data privacy and scalability. Embedding the approach within blockchain-native security frameworks and enabling real-time anomaly monitoring will further ensure adaptability to evolving threat landscapes. These enhancements can lay the groundwork for a more trustworthy and future-ready smart contract analysis platform.

Data availability

The data that support the findings of this study are openly available at <https://www.kaggle.com/datasets/tranduongminhdai/smartbug-dataset> reference number⁴².

Received: 2 March 2025; Accepted: 24 June 2025

Published online: 18 July 2025

References

1. He, D. et al. Smart contract vulnerability analysis and security audit. *IEEE Netw.* **34**(5), 276–282 (2020).
2. Xu, Y., Hu, G., You, L. & Cao, C. A novel machine learning-based analysis model for smart contract vulnerability. *Secur. Commun. Netw.* **2021**(1), 5798033 (2021).
3. Singh, A., Parizi, R. M., Zhang, Q., Choo, K. K. R. & Dehghantaha, A. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Comput. Secur.* **88**, 101654 (2020).
4. Xing, C. et al. A new scheme of vulnerability analysis in smart contract with machine learning. *Wirel. Netw.* **30**(7), 6325–6334 (2020).
5. Chu, H. et al. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.* **159**, 107221 (2023).
6. Tang, X., Du, Y., Lai, A., Zhang, Z. & Shi, L. Deep learning-based solution for smart contract vulnerabilities detection. *Sci. Rep.* **13**(1), 20106 (2023).
7. Zhang, L. et al. Smart contract vulnerability detection combined with multi-objective detection. *Comput. Netw.* **217**, 109289 (2022).
8. Huang, J. et al. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Trans. Inf. Forens. Secur.* **16**, 2144–2156 (2021).
9. Ma, F. et al. Security reinforcement for Ethereum virtual machine. *Inf. Process. Manage.* **58**(4), 102565 (2021).
10. Jj, L., Singh, K. & Chakravarthi, B. Digital forensic framework for smart contract vulnerabilities using ensemble models. *Multimed. Tools Appl.* **83**(17), 51469–51512 (2024).
11. Porkodi, S. & Kesavaraja, D. Smart contract: A survey towards extortionate vulnerability detection and security enhancement. *Wirel. Netw.* **30**(3), 1285–1304 (2024).
12. Dingman, W. et al. Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. *Int. J. Netw. Distrib. Comput.* **7**(3), 121–132 (2019).
13. Bhardwaj, A. et al. Penetration testing framework for smart contract blockchain. *Peer-to-Peer Netw. Appl.* **14**, 2635–2650 (2021).
14. Hu, T. et al. Transaction-based classification and detection approach for Ethereum smart contract. *Inf. Process. Manage.* **58**(2), 102462 (2021).
15. Vangala, A., Sutrala, A. K., Das, A. K. & Jo, M. Smart contract-based blockchain-envisioned authentication scheme for smart farming. *IEEE Internet Things J.* **8**(13), 10792–10806 (2021).
16. Lu, N., Wang, B., Zhang, Y., Shi, W. & Esposito, C. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Softw. Pract. Exp.* **51**(10), 2065–2084 (2021).
17. Charmet, F. et al. Explainable artificial intelligence for cybersecurity: A literature survey. *Ann. Telecommun.* **77**(11), 789–812 (2022).
18. Cao, S. et al. EXVUL: Towards effective and explainable vulnerability detection for IoT devices. *IEEE Internet Things J.* **11**(12), 22385–22398 (2024).
19. Tanga, C., Madhukar Mulpuri, D. A., Hemalatha, P. K., Singh, G., Pattewar, T. & Parveen, N. Advanced deep learning techniques for information security vulnerability detection using machine learning.
20. Aquilina, S. J., Casino, F., Vella, M., Ellul, J. & Patsakis, C. EtherClue: Digital investigation of attacks on Ethereum smart contracts. *Blockchain: Res. Appl.* **2**(4), 100028 (2021).
21. He, F., Li, F. & Liang, P. Enhancing smart contract security: Leveraging pre-trained language models for advanced vulnerability detection. *IET Blockchain* **4**, 543–554 (2024).
22. Dong, J. et al. Decentralized peer-to-peer energy trading strategy in energy blockchain environment: A game-theoretic approach. *Appl. Energy* **325**, 119852 (2022).
23. Dong, J. et al. Efficient and privacy-preserving decentralized energy trading scheme in a blockchain environment. *Energy Rep.* **8**, 485–493 (2022).
24. Dong, J., Song, C., Zhang, T., Li, Y. & Zheng, H. *Integration of Edge Computing and Blockchain for Provision of Data Fusion and Secure Big Data Analysis for Internet of Things* (Wiley, 2022).
25. Wu, H., Dong, H., He, Y. & Duan, Q. Smart contract vulnerability detection based on hybrid attention mechanism model. *Appl. Sci.* **13**(2), 770 (2023).
26. Yazdinejad, A., Dehghantaha, A., Parizi, R. M., Srivastava, G. & Karimipour, H. Secure intelligent fuzzy blockchain framework: Effective threat detection in IoT networks. *Comput. Ind.* **144**, 103801 (2023).
27. Chen, D., Feng, L., Fan, Y., Shang, S. & Wei, Z. Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention. *J. Syst. Softw.* **202**, 111705 (2023).

28. Ma, C., Liu, S. & Xu, G. HGAT: smart contract vulnerability detection method based on hierarchical graph attention network. *J. Cloud Comput.* **12**(1), 93 (2023).
29. Jie, W. et al. A novel extended multimodal AI framework towards vulnerability detection in smart contracts. *Inf. Sci.* **636**, 118907 (2023).
30. Dong, J., Song, C., Sun, Y. & Zhang, T. DAON: A decentralized autonomous oracle network to provide secure data for smart contracts. *IEEE Trans. Inf. Forens. Secur.* **18**, 5920–5935 (2023).
31. Osei, S. B., Ma, Z. & Huang, R. Smart contract vulnerability detection using wide and deep neural network. *Sci. Comput. Program.* **238**, 103172 (2024).
32. Sharma, B., Sharma, L., Lal, C. & Roy, S. Explainable artificial intelligence for intrusion detection in IoT networks: A deep learning based approach. *Expert Syst. Appl.* **238**, 121751 (2024).
33. Zhen, Z., Zhao, X., Zhang, J., Wang, Y. & Chen, H. DA-GNN: A smart contract vulnerability detection method based on dual attention graph neural network. *Comput. Netw.* **242**, 110238 (2024).
34. Wu, H., Peng, Y., He, Y. & Lu, S. EDSCVD: Enhanced dual-channel smart contract vulnerability detection method. *Symmetry* **16**(10), 1381 (2024).
35. Mothukuri, V., Parizi, R. M., Massa, J. L. & Yazdinejad, A. An AI multi-model approach to DeFi project trust scoring and security. in *2024 IEEE International Conference on Blockchain (Blockchain)* 19–28 (IEEE, 2024).
36. Dong, J. et al. Decentralized peer-to-peer energy trading: A blockchain-enabled pricing paradigm. *J. King Saud. Univ. Comput. Inf. Sci.* **37**, 10 (2025).
37. <https://www.kaggle.com/datasets/tranduongminhdai/smartbug-dataset>, and <https://github.com/smartbugs/smartbugs-wild>
38. Li, P. et al. A smart contract vulnerability detection method based on deep learning with opcode sequences. *Peer-to-Peer Network. Appl.* **17**(5), 3222–3238 (2024).
39. Lim, D. & Rebentrost, P. A quantum online portfolio optimization algorithm. *Quant. Inf. Process.* **23**(3), 63 (2024).
40. Bazionis, I. K., Kousounadis-Knousen, M. A., Katsigiannis, V. E., Catthoor, F. & Georgilakis, P. S. An advanced hybrid boot-LSTM-ICSO-PP approach for day-ahead probabilistic PV power yield forecasting and intra-hour power fluctuation estimation. *IEEE Access* **12**, 43704–43720 (2024).
41. Watanabe, A., Tamura, R., Takano, Y. & Miyashiro, R. Branch-and-bound algorithm for optimal sparse canonical correlation analysis. *Expert Syst. Appl.* **217**, 119530 (2023).
42. Annuzzi, G. et al. Exploring nutritional influence on blood glucose forecasting for Type 1 diabetes using explainable AI. *IEEE J. Biomed. Health Informat.* **28**(5), 3123–3133 (2023).
43. Nazir, A. et al. A deep learning-based novel hybrid CNN-LSTM architecture for efficient detection of threats in the IoT ecosystem. *Ain Shams Eng. J.* **15**(7), 102777 (2024).

Acknowledgements

None

Author contributions

Contributed to overall draft writing.

Funding

No funding was received.

Declarations

Competing interests

The authors declare no competing interests.

Ethical approval

This study did not involve any animal subjects.

Informed consent

Formal consent is not essential for this particular type of research.

Additional information

Correspondence and requests for materials should be addressed to B.R.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

© The Author(s) 2025