Information Sciences 722 (2026) 122645

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

An attack detection mechanism in smart contracts based on deep learning and feature fusion

Peiqiang Li [a,b], Guojun Wang [a,*], Wanyi Gu [a], Xubin Li [a], Xiangyong Liu [a], Yuheng Zhang [a]

[a] School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou 510006, China
[b] School of Computer and Information Technology, Xinyang Normal University, Xinyang 464000, China

ARTICLE INFO

Keywords:
Smart contracts
Opcode sequences
Feature fusion
Vulnerabilities
Attack detection

ABSTRACT

The rapid growth of Ethereum has spurred widespread adoption of smart contracts, enabling substantial financial transactions. Once deployed on the blockchain, smart contracts are immutable, rendering them unmodifiable even if vulnerabilities are present. In recent years, numerous attacks exploiting these vulnerabilities have caused significant financial losses. Although prior research has improved vulnerability detection in source code or bytecode before deployment, identifying attacks that exploit vulnerabilities during the execution phase after deployment remains a significant challenge. These challenges arise from the limited adaptability of predefined detection rules and an overreliance on opcode sequence names, which often neglects a comprehensive analysis of opcode sequence properties. In this study, we propose an advanced multidimensional feature fusion technique designed to detect attacks during the execution phase of smart contracts. By leveraging deep learning, our approach enhances detection accuracy through a comprehensive analysis of attack behaviors across four dimensions: operation objects, action behaviors, functional categories, and gas consumption. Extensive experiments demonstrate that our method achieves a detection accuracy of 97.21% and a weighted F1-score of 97.21%, confirming its effectiveness in identifying attacks.

1. Introduction

Blockchain technology is transforming the digital landscape, with smart contracts serving as a cornerstone of this evolution [1]. These self-executing contracts are valued for their automation, decentralized governance, and enhanced transparency, significantly contributing to the advancement of the digital economy. However, as smart contracts gain widespread adoption in global trade, their susceptibility to attacks has emerged as a critical concern.

Smart contract vulnerabilities pose significant risks, resulting in substantial financial losses. For example, in June 2016, a reentrancy vulnerability in the DAO smart contract was exploited, leading to the theft of approximately $60 million in Ether [2]. Since 2020, multiple smart contract-based platforms, including FarmEOS [3], Playgames [4], and EOSPlaystation [5], have been targeted, with cumulative losses reaching millions of dollars. These incidents highlight the persistent and severe security threats to blockchain technology, with smart contract security presenting a particularly formidable challenge.

* Corresponding author.
E-mail address: csgjwang@gzhu.edu.cn (G. Wang).

https://doi.org/10.1016/j.ins.2025.122645
Received 2 December 2024; Received in revised form 25 August 2025; Accepted 25 August 2025

Available online 2 September 2025

The prevalence of security vulnerabilities in smart contracts underscores the limitations of current detection tools in addressing complex attack scenarios. Traditional detection methods primarily rely on static analysis techniques, such as symbolic execution [6–8] and fuzz testing [9,10], to identify vulnerabilities in source code or bytecode before smart contract deployment [11]. However, these approaches often prove inadequate once contracts are deployed on the blockchain and enter the operational phase, as they struggle to address security threats in dynamic runtime environments. Furthermore, existing on-chain detection tools, such as SODA [12] and TxSpector [13], offer limited runtime behavior monitoring but rely heavily on predefined rules tailored to specific vulnerability types. This rule-based approach lacks the flexibility to adapt to rapidly evolving attack patterns and necessitates frequent updates to the rule base to address emerging vulnerabilities, significantly increasing maintenance costs and complexity.

To address these limitations, we propose an innovative approach for detecting smart contract attacks, utilizing a sophisticated feature fusion framework. Our method examines opcode behavior across multiple dimensions, including operation objects, action behaviors, functional categories, and gas consumption. By leveraging real-time transactional data, our system proactively identifies potential security threats, overcoming the constraints of conventional detection tools. In contrast to existing solutions, our approach integrates opcode sequences with a behavioral feature chain, significantly enhancing detection performance.

The key contributions of this paper are as follows:

- We construct a dataset of transaction opcode sequences by instrumenting the Geth client and replaying transactions, thereby comprehensively capturing the dynamic behavioral information of smart contracts at runtime. Through an in-depth analysis of opcode behaviors, we extract four highly discriminative feature types: operation objects, action behaviors, functional categories, and gas consumption. These features significantly enhance the dataset's quality and robustness.
- We propose an innovative method for detecting smart contract vulnerability attacks, leveraging a feature fusion approach that effectively integrates transaction opcode sequences with their corresponding behavioral features to create a comprehensive feature representation. Experimental evaluations on our meticulously curated dataset demonstrate that the method achieves a detection accuracy of 97.21% and a weighted F1-score of 97.21%. These results validate the effectiveness and robustness of the proposed approach, highlighting its substantial potential to enhance smart contract security.
- By dynamically analyzing opcode sequences during transaction execution, our approach effectively identifies anomalous behaviors and potential security threats often overlooked by static analysis tools.

The remainder of this paper is structured as follows: Section 2 reviews related work in the field. Section 3 provides essential background information, including an overview of smart contracts, transactions, and opcode sequences. Section 4 elaborates on the proposed methodology, detailing the feature fusion model and its specific design components. Section 5 presents a comprehensive evaluation of the system's performance through experimental results and analysis. Finally, Section 6 concludes the paper with a summary of key findings and directions for future research.

2. Related work

With the rising incidence of vulnerability attacks targeting Ethereum smart contracts, the demand for robust detection methods has become increasingly urgent. Current mainstream approaches encompass symbolic execution [14–16], fuzz testing [17–20], formal verification [21,22], and deep learning [23–26].

Oyente [27] analyzes smart contract bytecode using techniques like control flow graph generation, symbolic execution, and Z3 solvers to identify vulnerabilities such as reentrancy and exception handling. However, it faces efficiency issues with complex execution paths. Fuzz testing tools, like ContractFuzzer [28] and Reguard [29], inject random data to simulate diverse execution scenarios and detect vulnerabilities. Their probabilistic approach, while effective, may miss some execution paths, produce false positives, and increase analysis time and complexity.

Formal verification tools, such as F* [30] and Zeus [31], use mathematical reasoning, abstract interpretation, and symbolic model checking for rigorous security analysis of smart contracts. These methods are thorough but challenging for complex contracts, requiring specialized expertise and balancing complexity with efficiency. Deep learning-based approaches have emerged for vulnerability detection. Torres et al. [32] transform the problem into image classification using RGB color coding. Zhang et al. [33] developed a multi-objective detection algorithm based on implicit features. SCVDIE [34] uses information graphs and ensemble learning to vectorize operand co-occurrence matrices across seven models. Cbgru [35] integrates word embeddings with deep learning for a hybrid model, while Mi et al.'s VSCL [36] uses a deep neural network with metric learning. SmartEmbed [37] detects vulnerabilities by comparing contract similarity. Despite strong detection capabilities, deep learning models often lack transparency, complicating result interpretation.

Transaction-based security analysis for Ethereum is underexplored. Sereum [38] employs dynamic taint tracking to detect reentrancy attacks, and Grossman et al. [39] use transaction analysis to evaluate contract resilience. These tools focus on specific vulnerabilities, potentially missing complex attacks. TxSpector, an open-source framework, conducts bytecode-level analysis of Ethereum transactions using user-defined rules to identify Ethereum-specific attacks, but its effectiveness depends heavily on the accuracy of these rules.

Compared to existing approaches, our model introduces a feature fusion framework that provides a more comprehensive dynamic analysis for detecting diverse potential attacks in on-chain smart contracts. A preliminary version of this work was previously published [40], and in this extended version, we have significantly optimized the model to enhance both detection accuracy and robustness. Specifically, we have restructured the model architecture to improve the synergy among different feature modules, thereby

enhancing overall performance. Additionally, we refined the behavioral feature chain to capture opcode sequence patterns with greater precision. To further improve detection capabilities, we optimized the dataset and enhanced the vectorized representation of opcodes to better reflect their semantic relationships. Extensive experiments validate the superior performance of our optimized model in detecting various attack types.

3. Background

To provide context for our research, this section presents a concise overview of key concepts, including smart contracts, Ethereum Virtual Machine, transactions, and opcode sequences.

3.1. Smart contracts

Ethereum smart contracts are self-executing programs that encode and automatically enforce the terms of an agreement. Deployed on the Ethereum blockchain, these contracts are decentralized, transparent, and immutable, meaning they cannot be altered once deployed. Activated by predefined conditions, smart contracts autonomously execute specified actions without intermediaries. Typically written in high-level languages such as Solidity, they are compiled into bytecode and executed by the Ethereum Virtual Machine (EVM). Smart contracts underpin a wide range of decentralized applications (DApps), from financial systems [41] to supply chain management [42], offering secure, efficient, and automated solutions for blockchain-based operations. However, their widespread adoption introduces significant security and compliance challenges.

3.2. Ethereum virtual machine

The Ethereum Virtual Machine (EVM) is a decentralized computational environment forming the core of the Ethereum blockchain. It executes smart contracts, which are self-executing agreements with terms directly encoded in code. Operating as a stack-based virtual machine, the EVM processes bytecode derived from smart contracts. Designed for complete isolation, it ensures that contract execution remains independent of the host system, guaranteeing consistency and security across the distributed network. Each Ethereum node runs an EVM instance, responsible for validating transactions and maintaining blockchain integrity. The EVM executes instructions via opcodes and manages resources, such as gas, required for operations within the Ethereum network. As the foundation for decentralized applications (DApps) and complex computations, the EVM is critical to maintaining the decentralized, secure, and transparent nature of the Ethereum ecosystem.

3.3. Transactions

In the Ethereum network, a transaction represents a state transition, such as transferring value between accounts or executing a smart contract [43]. Ethereum transactions can be categorized into two primary types based on their initiator and execution method [44]:

- **External Transactions**: Initiated by externally owned accounts (user accounts), these transactions involve direct financial transfers between accounts, modifying account balances. Their details and outcomes are transparently recorded on the blockchain.
- **Internal Transactions**: Triggered by smart contracts, these transactions are executed autonomously according to the contract's internal logic, independent of external accounts. Internal transactions enable dynamic interactions within the Ethereum ecosystem, facilitating communication between contracts and between contracts and external accounts.

3.4. Opcode sequences

To interact with a smart contract deployed on the blockchain, a user initiates a transaction by specifying the target contract's address, along with the function to be called and its parameters. This transaction triggers the execution of the corresponding function within the smart contract. If the function invokes another function in a different smart contract, it further triggers the execution of that function. This process generates a transaction execution trace, spanning from the user's initiation of the transaction to the completion of all involved function executions. During the EVM execution, we collect the sequentially executed opcodes along the transaction trace using an instrumented Geth client, forming the transaction's opcode sequence.

The generation of a transaction's opcode sequence is illustrated in Fig. 1. The central portion of Fig. 1 depicts the control flow graph (CFG) constructed from the smart contract's opcodes, which includes a test() function. Once the smart contract is deployed on the blockchain, a user can initiate a call to the test() function. If the input parameter is 8, the execution follows Path 1: Block1→Block2→Block4; if the parameter is 5, it follows Path 2: Block1→Block3→Block4. Consequently, the opcodes along Path 1 or Path 2 constitute the transaction's opcode sequence. To streamline this sequence for further processing, we normalize it by removing the operands following the opcodes.

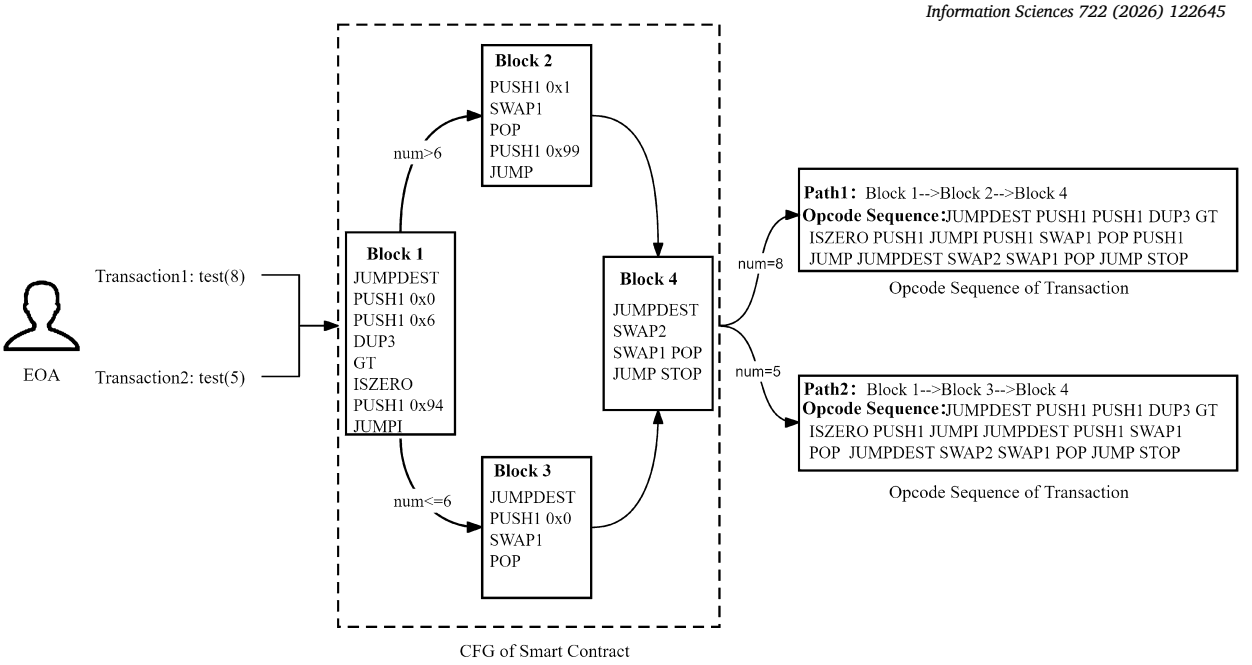


Fig. 1. Opcode Sequence of Transaction.

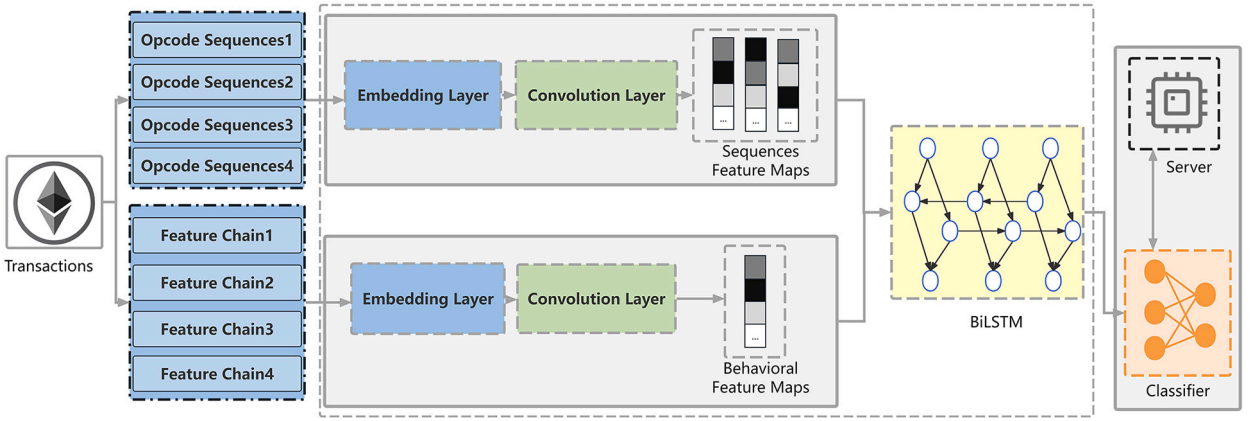


Fig. 2. System Framework.

4. Methodology

4.1. System overview

Current research on smart contract vulnerabilities predominantly focuses on pre-deployment code analysis, employing static examination of source code, bytecode, or opcode sequences to identify potential security risks. However, detecting vulnerabilities in smart contracts already deployed on public blockchains is equally critical. To address this, we propose a novel approach for detecting smart contract attacks by leveraging transaction opcode sequences and behavioral features.

As depicted in Fig. 2, our approach utilizes transactions executed by smart contracts as input samples for attack detection. This method employs deep learning techniques to comprehensively analyze runtime transaction opcode sequences and their associated behavioral features. By extracting attack-related features across multiple dimensions, our approach significantly enhances the efficiency of vulnerability attack detection. The methodology comprises three core modules:

- **Opcode Sequence Module**: This module processes transaction opcode sequences using embedding and convolutional layers, transforming each transaction into a structured opcode sequence. This transformation captures underlying patterns, enabling the model to distinguish routine operations from anomalous behaviors.

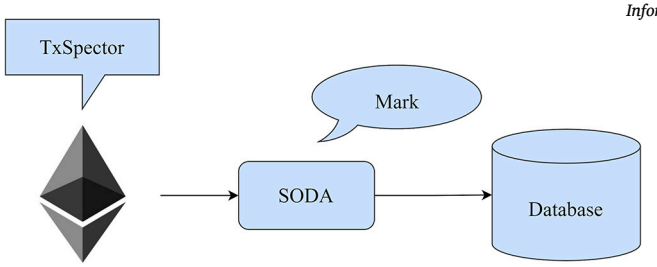


Fig. 3. The Process of Data Collection.

- **Behavioral Feature Chain Module**: This module constructs a feature chain to represent the behavioral characteristics of a transaction, which is then mapped into a comprehensive feature map through embedding and convolutional layers. This process provides a robust representation of behavioral features for attack detection and facilitates the differentiation of various attack types, thereby improving the precision of threat identification and analysis.
- **BiLSTM Module**: As a higher-order component, the BiLSTM module captures and interprets complex dynamic behaviors within transaction sequences. It effectively analyzes multi-level relationships and temporal variations in opcodes, identifying behavioral patterns and their potential correlations. This enhances the model's accuracy in detecting attacks.

In addition to the aforementioned encoder modules, the framework integrates a multilayer perceptron (MLP) classifier. The MLP is trained to efficiently analyze and classify input sequences, delivering robust attack detection outcomes. By scrutinizing the detailed behavioral characteristics of opcodes and their interrelationships across various stages of transaction execution, the framework effectively identifies potential attacks on smart contracts deployed on the blockchain, overcoming the limitations of conventional detection tools.

4.2. Opcode sequence module

In this phase, we instrumented the Ethereum client, Geth, by extending the TxSpector framework through the integration of additional code into the Ethereum source code. This enhancement enabled real-time observation of smart contract dynamic behavior, providing a foundation for a comprehensive understanding of their operational mechanisms. We established a full Ethereum node to replay transactions, with our custom code tracking and recording each transaction to monitor the specific execution paths triggered within the smart contract. This process included capturing function calls and their associated parameters, facilitating accurate collection of corresponding opcode sequences. These sequences were subsequently processed and labeled using the SODA framework, and both the sequences and their labels were stored in a database. The methodology for collecting and labeling opcode sequence data is depicted in Fig. 3.

Transaction opcode sequences capture the instruction sequences executed by a smart contract within a blockchain virtual machine, such as the Ethereum Virtual Machine (EVM), reflecting the contract's runtime logic and dynamic behavior. Unlike static analysis, which examines only source code or bytecode, opcode sequences effectively capture anomalous behaviors tied to the runtime context, including reentrancy attacks, logic errors from dynamic calls, or unexpected state changes. This dynamic information is critical for identifying runtime-triggered vulnerabilities, such as those dependent on timing or transaction ordering.

For each opcode sequence, the first step involves embedding each opcode into a k-dimensional vector, resulting in an embedded matrix $O \in \mathbb{R}^{n \times k}$, where n is the length of the opcode sequence and k is the dimensionality of the embedding. Each row of the matrix corresponds to the feature vector of the respective opcode in the sequence.

$$O = [o_1, o_2, \ldots, o_n]^T \tag{1}$$

Subsequently, the matrix is processed through a convolutional neural network (CNN) layer to extract higher-level features. This operation is represented as follows:

$$C_{opcode} = f(W * O + b) \tag{2}$$

where C_{opcode} is the feature map output by the CNN, capturing high-level features from the opcode sequence; W represents the weights of the convolutional filter, which learns to identify specific patterns in the sequence; b is the bias term, allowing for the adjustment of the filter's output; and f is the activation function (e.g., ReLU), which introduces non-linearity and helps model complex relationships.

Fig. 4 illustrates the process of transforming an opcode sequence into a dense feature vector and feature map using embedding and CNN layers. This transformation is critical for processing sequence data in machine learning models for smart contract attack detection. The embedding layer maps discrete opcode symbols into fixed-size, high-dimensional dense vectors, generating a richer feature representation that captures semantic relationships between opcodes. Subsequently, the CNN layer processes these dense vectors to produce a feature map that encapsulates both local and global contextual information within the opcode sequence.

The primary objective of this transformation is to enhance the model's ability to interpret the semantics of opcode sequences and identify interdependencies among opcodes. Specifically, the dense vectors capture both the intrinsic properties of individual opcodes

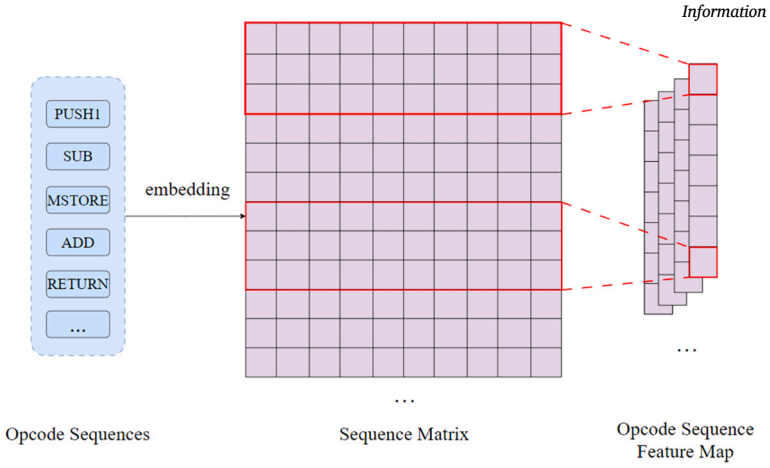


Fig. 4. Opcode Sequence Feature Representation.

and their complex relationships through high-dimensional spatial representations. This approach enables the model to detect patterns and potential anomalies in opcode sequences, thereby improving the accuracy of attack detection.

4.3. Behavioral feature chain module

This stage aims to extract four characteristics from each opcode and to construct a behavioral feature chain for the opcode sequences. Each opcode is described as a 4-tuple:

$$op = \langle \text{object}, \text{class}, \text{category}, \text{gas} \rangle \tag{3}$$

In the given context, op represents the opcode, $object$ denotes the operation objects by the opcode, $class$ describes the action behaviors of the opcode from the perspective of blockchain system resources, $category$ signifies the functional categories of the opcode, and gas indicates the gas consumption of the opcode, which represents the level of resource consumption.

1) The $object$ is categorized into five types based on the operation objects of the opcode [45]:

$$\text{object} \in \{\text{‘stack’}, \text{‘memory’}, \text{‘storage’}, \text{‘blockchain’}, \text{‘extra’}\} \tag{4}$$

The ‘stack’ category contains opcodes that manipulate the execution stack or operate on existing values in the stack, such as PUSHx and ADD. These opcodes are responsible for handling temporary data, pushing data into or out of the stack to support complex computational implementations. The ‘memory’ class of opcodes is used to perform read and write operations on the memory in which the contract is executed, such as MSTORE and MLOAD. These opcodes provide temporary storage for the runtime by interacting with memory locations and data to support complex logic and data processing. The ‘storage’ category includes opcodes that read and write to persistent storage, such as SSTORE and SLOAD. Since storage operations change the on-chain state of a contract, they involve persistent storage. The ‘blockchain’ category includes opcodes that extract information from the blockchain or the current transaction, such as TIMESTAMP and ORIGIN. These opcodes provide smart contracts with the external contextual data they need to execute by accessing the contextual information of the blockchain without changing the state of the contract. The ‘extra’ category includes opcodes that do not operate directly on a data structure (such as the stack, memory, or storage), such as JUMPDEST. These operations are mainly used to control the implementation of logic, such as setting jump targets.

Classifying opcodes by their operational objects enhances vulnerability detection by identifying security risks associated with specific data interactions. For instance, stack operations may lead to stack overflow or underflow, storage operations are susceptible to reentrancy attacks, and blockchain operations can be exploited in timestamp-dependent attacks. By analyzing these operational objects, the model effectively detects anomalous interaction patterns and identifies potential vulnerabilities.

2) The $class$ attribute characterizes the behavioral impact of opcodes on blockchain system resources and is categorized into three types:

$$\text{class} \in \{\text{‘select’}, \text{‘update’}, \text{‘other’}\} \tag{5}$$

The ‘select’ opcodes are used to access the blockchain’s system resources, such as reading data from storage or accessing information about an account. These operations do not result in a change of state, as they only involve the extraction of data. Typical query opcodes include BALANCE, SLOAD, etc. These opcodes are only used to retrieve information during execution and do not affect the integrity or consistency of the data in the chain. The main function of ‘update’ opcodes is to change the system resources of the blockchain, which not only involves changing the system state but also increases the consumption of resources, e.g., changing the account balance, writing new memory values, etc. Common update opcodes are SSTORE and SELFDESTRUCT, which make changes to on-chain data and thus have a real impact on the blockchain’s ledger and storage state. The ‘other’ type of opcodes does not involve

querying system resources or changing state and is mainly used for purely computational tasks or control logic. For example, JUMP and JUMPI are responsible for controlling process jumps. As these operations do not affect the state of the blockchain, they are lighter in terms of resource consumption.

Classifying opcodes by their action behaviors elucidates their impact on the blockchain state and facilitates the detection of vulnerabilities associated with state changes. For example, select opcodes (e.g., SLOAD) may risk information leakage, update opcodes (e.g., SSTORE) are susceptible to reentrancy attacks, and other opcodes (e.g., JUMP) may lead to control flow errors. By distinguishing these action types, the model effectively monitors update operations to detect anomalous state changes and identify potential vulnerabilities.

3) Opcodes are classified into 10 functional categories as defined in the Ethereum Yellow Paper, based on their roles within the Ethereum network [46]:

$$\begin{aligned} \text{category} \in \{&\text{'arithmetic'}, \text{'comparisonlogic'}, \text{'environmental'}, \text{'blockinfo'}, \text{'smsinfo'}, \text{'push'}, \text{'duplication'}, \\ &\text{'exchange'}, \text{'logging'}, \text{'system'}\} \end{aligned} \tag{6}$$

The 'arithmetic' category includes the basic arithmetic operation opcodes in the smart contract, such as ADD, SUB, MUL, and DIV. The 'comparelogic' class opcodes are used to perform logical evaluation and comparison operations, including logical operations (e.g., AND, OR, NOT) and conditional comparisons (e.g., GT, LT, EQ, etc.). The opcodes in the 'environment' category are used to obtain necessary information from the blockchain environment, usually including the current transaction or the state of the blockchain. The 'blockinfo' category contains opcodes to retrieve information about the current block on the blockchain, such as TIMESTAMP, NUMBER, etc. The 'smsinfo' class of opcodes is responsible for reading or querying internal data structures such as MSTORE, SSTORE, etc. The 'push' class of opcodes is used to place specific values on the execution stack, such as PUSH1, PUSH2, etc. The 'duplication' category includes opcodes that copy existing data on the stack, such as DUP1, DUP2, etc. The 'swap' category opcodes are used to swap data positions on the stack, such as SWAP1, SWAP2, etc. The 'logging' category opcodes are used to create event log records on the blockchain, such as LOG0, LOG1, etc. The 'system' category opcodes are typically concerned with the lifecycle management of smart contracts, such as contract creation and destruction (e.g., CREATE, SELFDESTRUCT).

Functional categorization provides a granular perspective on opcode roles, enhancing fine-grained vulnerability detection. For instance, arithmetic operations may lead to integer overflows, environment and block information operations are susceptible to external data attacks, and logging and system operations may trigger resource exhaustion or self-destruct vulnerabilities. By leveraging functional categorization alongside known vulnerability patterns, the model enables targeted inspection of high-risk functions.

4) The gas attribute represents the computational cost of an opcode, categorized into 18 levels as specified in Ethereum's official documentation:

$$gas \in \{\text{'0'}, \text{'1'}, \text{'2'}, \text{'3'}, \text{'5'}, \text{'8'}, \text{'10'}, \text{'20'}, \text{'30'}, \text{'100'}, \text{'375'}, \text{'750'}, \text{'1125'}, \text{'1500'}, \text{'1875'}, \text{'5000'}, \text{'32000'}, \text{'NAN'}\} \tag{7}$$

Each level corresponds to the resource consumption of the opcode, with higher values indicating greater gas usage. The gas required for an opcode directly reflects its computational resource demands. Opcodes with high gas costs, such as those involving complex computations or memory operations, typically consume significant resources. In cases where gas consumption reaches the maximum defined value, it indicates extremely high resource usage, often considered 'infinite' in practical terms. To facilitate system calculations and ensure robust handling of such extreme cases, we assign a value of '99999' to represent extreme resource consumption, replacing 'NaN' to enable seamless subsequent computations.

Gas consumption analysis reveals the computational complexity and potential security risks associated with opcodes. High-gas operations, such as SSTORE, are susceptible to exploitation in denial-of-service attacks, while anomalous gas consumption patterns may signal vulnerabilities. By examining gas usage patterns, the model identifies resource inefficiencies or attack vectors, optimizes execution processes, and prioritizes the scrutiny of high-consumption operations to mitigate security threats.

The behavioral feature chain module is designed to accurately capture the behavioral characteristics of opcodes, providing comprehensive information for smart contract analysis. As illustrated in Fig. 5, this module extracts behavioral features through a two-step process: first, constructing the behavioral feature chain, and second, generating the behavioral feature map.

In the first step, outlined in Algorithm 1, the module extracts four key features from each opcode in the sequence—operation objects, action behaviors, functional categories, and gas consumption—forming a 4-tuple. These features are concatenated to create a structured sequence, termed the behavioral feature chain, as shown in Table 1. This chain provides detailed insights into the behavior of each opcode.

In the second step, the behavioral feature chain is transformed into a feature map to support the deep learning process. This transformation involves embedding and convolutional layers. The embedding layer maps the discrete feature symbols of the behavioral feature chain into dense vectors in a high-dimensional space, while the convolutional layer analyzes complex dependencies among features to generate the behavioral feature map. This map captures both local and global contextual information, preserving subtle behavioral differences among opcodes and revealing their dynamic interrelationships. Consequently, the model gains a deeper understanding of the smart contract's execution logic, significantly enhancing the detection model's ability to identify anomalous behaviors and improving the accuracy of risk detection.

Consider a behavioral feature chain denoted as $\{op_i\}_{i=1}^{n}$, where each op_i is a tuple $(object_i, class_i, category_i, gas_i)$. The transformation through an embedding layer is represented as:

$$\mathcal{Q}_i = \Phi(\omega_{object_i}, \omega_{class_i}, \omega_{category_i}, \omega_{gas_i}), \quad \forall i \in \{1, \ldots, n\} \tag{8}$$

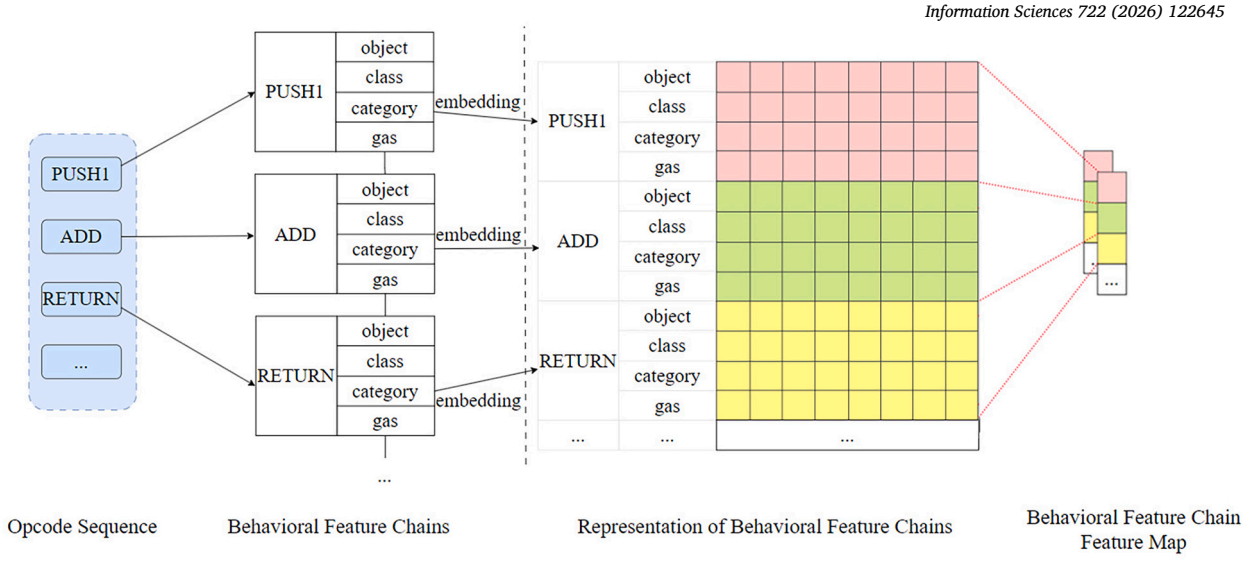


Fig. 5. Opcode Sequence Behavioral Feature Chain Extraction Process.

Algorithm 1: Extraction of Behavioral Feature Chains.

Input: $OS \leftarrow$ set of opcode sequences, $OFD \leftarrow$ opcode feature dictionary
Output: $BFC \leftarrow$ list of behavioral feature chains
$BFC \leftarrow \emptyset$; // Initialize the Behavioral Feature Chains list
foreach *opcode sequence* $os \in OS$ **do**
  $bfci \leftarrow \emptyset$; // Initialize the feature chain for the current sequence
  $sequence\_length \leftarrow \text{length}(os)$; // Get the length of the opcode sequence
  if $sequence\_length = 0$ **then**
    continue ; // Skip empty sequences
  foreach *opcode* $op \in os$ **do**
    $(object, class, category, gas) \leftarrow$ NULL;
    $found\_feature \leftarrow$ False;
    foreach *feature* $f \in OFD$ **do**
      if $f.op = op$ **then**
        Extract $object, class, category, gas$ from f;
        $found\_feature \leftarrow$ True;
        break;
    if not $found\_feature$ **then**
      // Handle cases where no feature is found
      $object \leftarrow$ "default_object";
      $class \leftarrow$ "default_class";
      $category \leftarrow$ "default_category";
      $gas \leftarrow 0$;
    $tuple \leftarrow (object, class, category, gas)$;
    $bfci$.append($tuple$);
    // Optional logging for debugging
    if $DEBUG$ **then**
      print("Appended tuple: ", $tuple$)
  // Perform additional checks before appending
  if len(bfci) > 0 **then**
    BFC.append($bfci$);
return BFC

where Φ symbolizes the embedding operation turning each element into a vector in $\mathbb{R}^k$, and $Q_i \in \mathbb{R}^{4\times k}$ represents the embedded matrix.

The overall representation of the behavioral feature chain in high-dimensional space is then given by:

$$Q = \bigoplus_{i=1}^{n} Q_i \tag{9}$$

where $\bigoplus$ denotes a tensorial operation aggregating the individual matrices Q_i into a higher-order tensor $Q \in \mathbb{R}^{n\times 4\times k}$.

Table 1
An Example of Conversion from Opcode to 4-tuple.

Opcode	*object*	*class*	*category*	*gas*
ADD	stack	update	arithmetic	3
TIMESTAMP	blockchain	select	blockinfo	2
MSTORE	memory	update	smsinfo	3
SSTORE	storage	update	smsinfo	100
JUMPI	extra	other	smsinfo	10
...	...	...	...	...

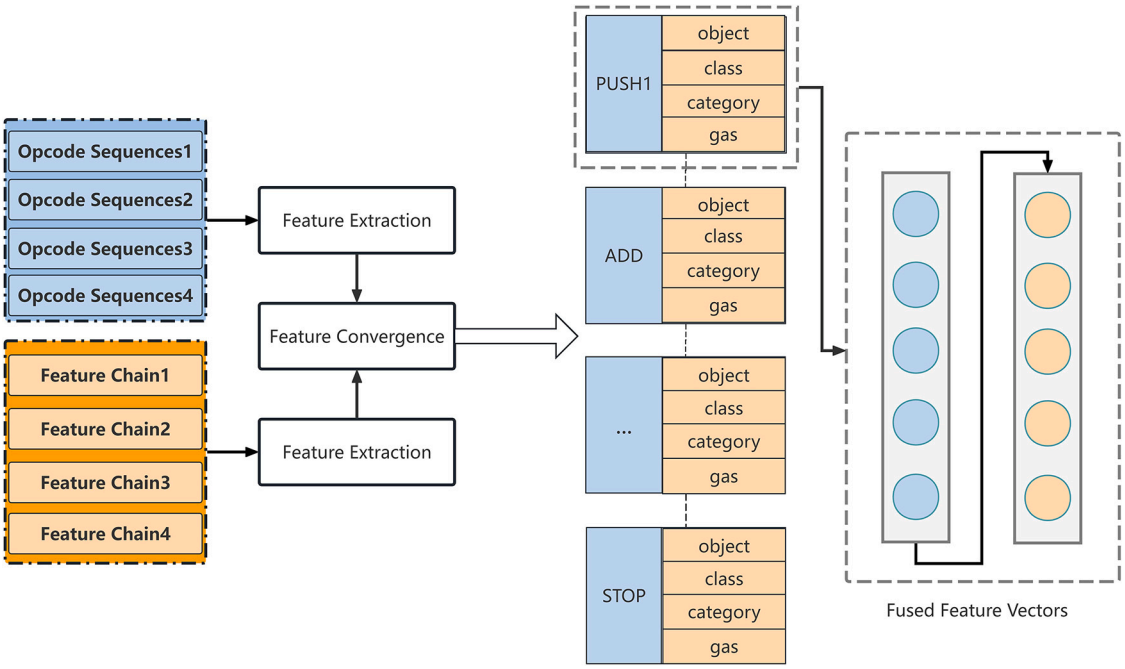


Fig. 6. Feature Convergence.

Application of a convolutional layer to obtain the feature map C from Q is abstracted as:

$$C_{behavior} = \sigma(\mathcal{W} \star Q + \beta) \tag{10}$$

where σ denotes the nonlinear activation function, $\mathcal{W}$ denotes the convolution filter weights, β denotes the bias, and $\star$ denotes the convolution operation.

By integrating embedding and convolutional operations, we derive a refined feature map C_{behavior}. The embedding operation maps features into a high-dimensional space, preserving their contextual relationships, while the convolutional operation captures local patterns and interactions among features. This combined approach enables the model to discern subtle behavioral characteristics of each opcode, thereby improving the detection of complex attack patterns.

4.4. BiLSTM module

Given the vectorized representation from the opcode sequences, $C_{opcode} \in \mathbb{R}^{n\times m}$ and the vectorized representation from the behavioral feature chains, $C_{behavior} \in \mathbb{R}^{n\times p}$, we aim to fuse these matrices into a unified representation, $C_{unified} \in \mathbb{R}^{n\times(m+p)}$ for processing by the BiLSTM architecture. As shown in Fig. 6, this fusion is done using a concatenation paradigm, briefly expressed as:

$$C_{unified} = \left[C_{opcode} \oplus C_{behavior}\right],$$
$$\oplus \equiv \text{Concatenation Operation} \tag{11}$$

Here, $\oplus$ symbolizes the intricate process of concatenating feature dimensions from C_{opcode} and $C_{behavior}$.

Following the fusion, $C_{unified}$ is processed through a bidirectional long short-term memory (BiLSTM) network to capture both forward and backward sequential dependencies, yielding a composite feature representation $H \in \mathbb{R}^{n\times d}$, where d represents the dimensionality of the BiLSTM outputs. The transformation performed by the BiLSTM is mathematically expressed as:

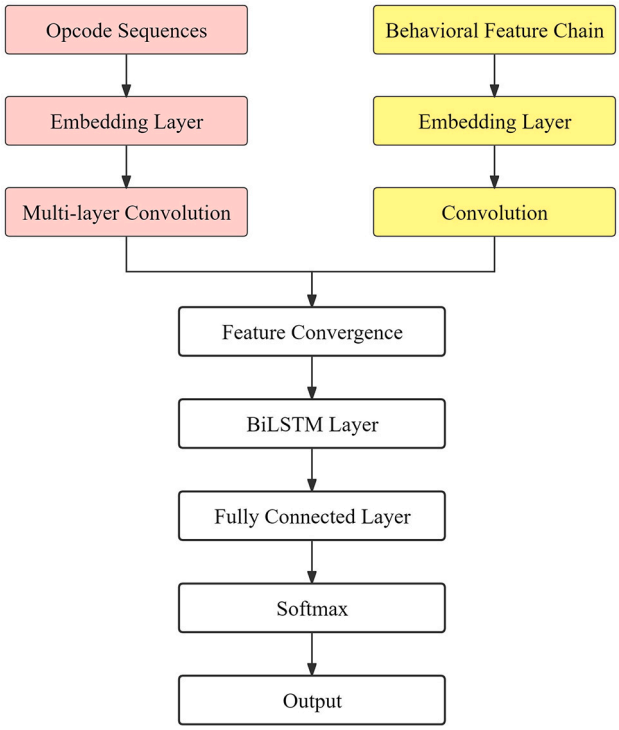


Fig. 7. Model Architecture.

$$H = \text{BiLSTM}(C_{unified}) \tag{12}$$

This approach enables the model to leverage both the detailed operational patterns captured by opcode sequences and the higher-level behavioral patterns derived from behavioral feature chains. By integrating these opcode-level and behavior-level insights, the model significantly enhances its capability to detect vulnerability attacks.

4.5. Model architecture

The final model architecture, depicted in Fig. 7, is a deep neural network designed to identify distinct attack patterns within input opcode sequences. The model comprises two primary components: an intrinsic feature encoder and a multilayer perceptron (MLP) classifier. The intrinsic feature encoder consists of three modules: the opcode sequence module, the behavioral feature chain module, and the bidirectional long short-term memory (BiLSTM) module. These modules collaboratively extract detailed features from opcodes and capture global relationships within the behavioral feature chain, thereby enhancing the model's precision in detecting potential threats.

The opcode sequence module employs an advanced feature extraction approach. It first maps opcodes into a high-dimensional vector space using an embedding layer. This representation is then processed by multiple parallel convolutional layers with a multi-scale filter configuration, enabling the capture of local patterns at varying granularities. This approach significantly improves the quality of the feature representation and enhances the model's overall performance.

The behavioral feature chain module constructs a behavioral feature chain by applying embedding and convolutional layers, configured with a step size of 4 and convolutional filters of size 4. This setup transforms the behavioral feature chain into a feature matrix, allowing the convolutional layers to accurately extract opcode feature patterns and generate a robust sequence of features.

The BiLSTM module further processes the integrated information from the opcode sequence and behavioral feature chain modules. It captures dependencies among opcodes and identifies higher-level feature patterns. The bidirectional structure of the BiLSTM incorporates both preceding and succeeding contextual information from the opcode sequences, providing a comprehensive understanding of opcode interactions and behavioral patterns. This capability is critical for detecting complex attack scenarios.

Following the BiLSTM module, the MLP classifier performs the final classification task. It comprises two dense layers, each followed by a dropout layer to mitigate overfitting. The classification results are generated using a softmax function, which computes the probability distribution for each input sequence and assigns the corresponding attack type label. This layered architecture ensures efficient and accurate attack detection within a deep learning framework.

Table 2
Dataset Overview.

Vul Type	Vul Name	Number
V0	Normal	4188
V1	Incorrect Check for Authorization	3665
V2	No Check after Contract Invocation	1353
V3	Missing the Transfer Event	1574
V4	Strict Check for Balance	744
V5	Timestamp & Block Number Dependency	7008

Table 3
Hyperparameter Definitions and Options.

Hyperparameter	Definition	Options
sequence_units	Number of input units in the opcode sequence module	{8, 16, 32}
chain_units	Number of input units in the behavioral chain module	{8, 16, 32}
conv_filter	Number of filters in the convolutional layer	{32, 64, 128}
lstm_units	Number of units in the BiLSTM layer	{100, 200}
dense1_units	Number of units in the first dense layer	{64, 128}
dense2_units	Number of units in the second dense layer	{32, 64}
dropout_rate	Dropout rate	{0.2, 0.5}
optimizer	Optimizer	{Adam, SGD}
learning_rate	Learning rate	$\{10^{-4}, 10^{-3}, 10^{-2}\}$

Table 4
Hyperparameter Settings for Ten Bayesian Optimization Trials (batch size = 80).

Trial_id	sequence_units	chain_units	conv_filter	lstm_units	dense1_units	dense2_units	dropout_rate	optimizer	learning_rate
0	16	8	64	100	64	32	0.2	Adam	10^{-3}
1	8	16	32	200	128	64	0.5	SGD	10^{-2}
2	32	8	128	100	64	32	0.2	Adam	10^{-4}
3	8	32	64	200	128	64	0.5	Adam	10^{-3}
4	16	16	128	100	64	32	0.5	SGD	10^{-3}
5	32	32	64	200	128	64	0.2	Adam	10^{-4}
6	8	8	32	100	64	32	0.2	SGD	10^{-2}
7	16	32	128	200	128	64	0.5	Adam	10^{-3}
8 (best)	**16**	**8**	**128**	**100**	**64**	**32**	**0.2**	**Adam**	$\mathbf{10^{-3}}$
9	32	16	64	200	128	64	0.2	SGD	10^{-4}

5. Experiments

This section provides an overview of the experimental dataset, followed by a discussion of the hyperparameter settings and evaluation metrics. Finally, it presents an analysis of the experimental results.

5.1. Data collection

As described in Section 4.2, a total of 265,561,265 transaction opcode sequences were collected by instrumenting the Geth client and replaying transactions. These sequences were subsequently categorized into distinct datasets, comprising normal samples (V0) and five attack sample categories: incorrect check for authorization (V1), no check after contract invocation (V2), missing the transfer event (V3), strict check for balance (V4), and timestamp & block number dependency (V5).

Statistical analysis revealed that over 90% of the collected opcode sequences were normal. Furthermore, many transactions exhibited repetitive execution paths, resulting in a significant number of duplicate opcode sequences. After deduplication and retaining a representative subset of normal sequences, the final dataset was constructed, as detailed in Table 2.

5.2. Hyperparameter settings

The experiment utilizes Keras Tuner for Bayesian optimization of hyperparameters in the CNN-BiLSTM multi-classification model to enhance the detection of smart contract vulnerabilities. Table 3 lists the hyperparameters and their possible values. Table 4 details the specific hyperparameter configurations for each of the ten Bayesian optimization trials, with a fixed batch size of 80. The eighth trial produced the optimal hyperparameter configuration.

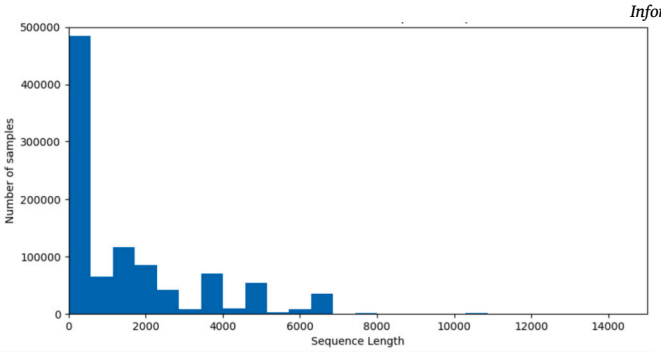


Fig. 8. Frequency Distribution of Transaction Opcode Lengths.

5.3. Evaluation metrics

To comprehensively evaluate the performance of the proposed multi-class classification model, we adopt the following evaluation metrics: accuracy, weighted average precision, weighted average recall, and weighted average F1-score. These metrics provide a robust assessment of the model's performance in multi-class tasks, particularly for imbalanced datasets.

1) Accuracy: This metric represents the proportion of correct predictions made by the model, defined as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{13}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. Accuracy measures how often the model's predictions are correct across all classes.

2) Weighted Average Precision: This metric measures the model's precision on a per-class basis, weighted by the proportion of each class in the dataset. It is defined as:

$$\text{Weighted Precision} = \sum_{i=1}^{N} \left(\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i} \times w_i \right) \tag{14}$$

where TP_i and FP_i represent the true and false positives for the i-th class, and w_i is the weight representing the proportion of the i-th class in the dataset.

3) Weighted Average Recall: This metric assesses the model's ability to correctly identify all instances of each class, with weights reflecting the class distribution in the dataset. It is defined as:

$$\text{Weighted Recall} = \sum_{i=1}^{N} \left(\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i} \times w_i \right) \tag{15}$$

where TP_i and FN_i are the true positives and false negatives for the i-th class.

4) Weighted Average F1-score: This metric balances precision and recall, computed as the weighted harmonic mean of the two, defined as:

$$\text{Weighted F1-score} = \sum_{i=1}^{N} \left(\frac{2 \times \text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i} \times w_i \right) \tag{16}$$

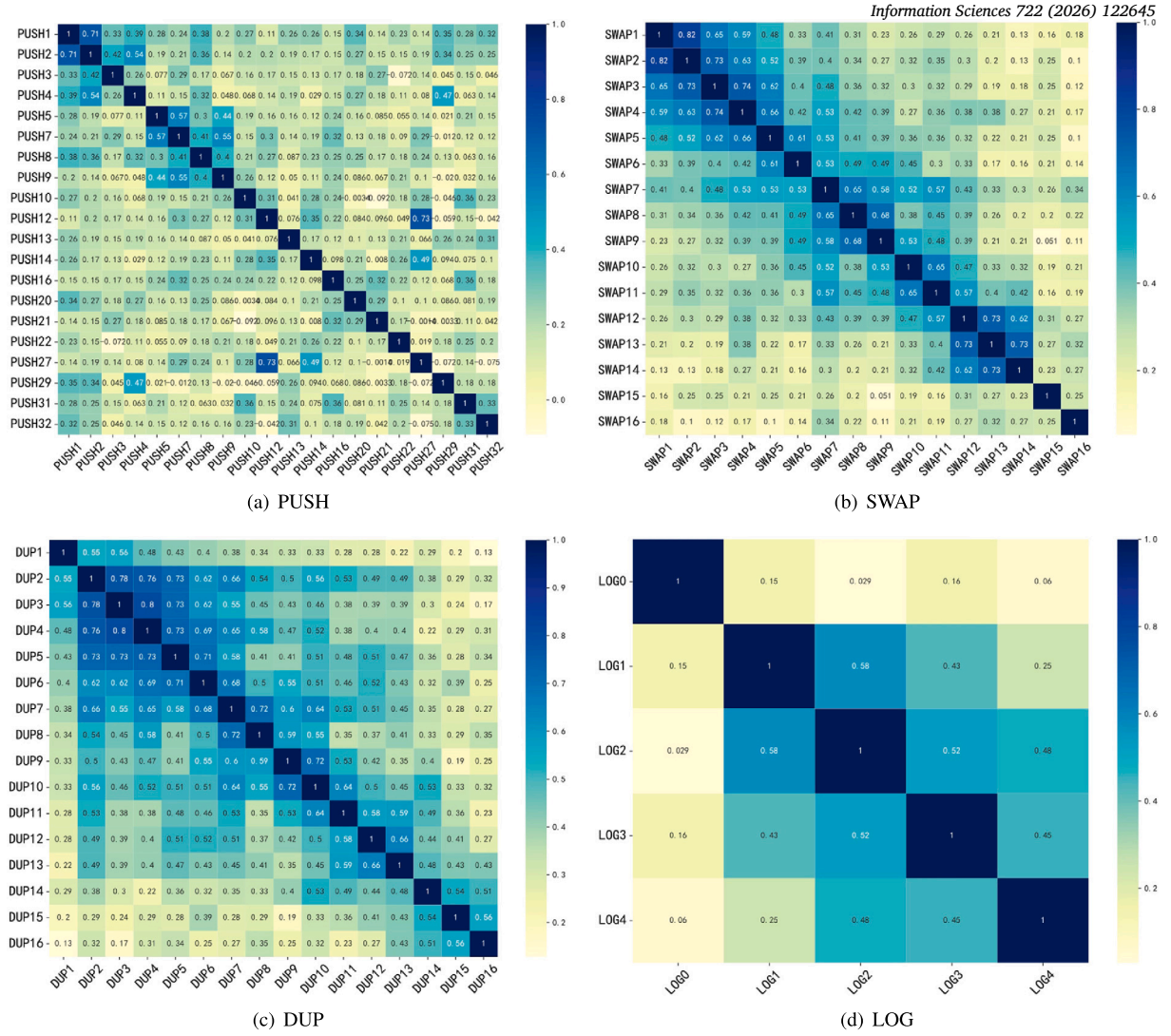
where Precision_i and Recall_i are the precision and recall values for the i-th class, and w_i is the weight of the class.

5.4. Experimental result

As depicted in Fig. 8, a large volume of opcode sequences was collected using the Geth instrumentation technique and subjected to statistical analysis. The results indicated that approximately 80% of the opcode sequences have a maximum length of 3000. Accordingly, we configured the maximum input length for the opcode sequence module to 3000 and for the behavioral feature chain module to 12,000. This configuration ensures that the model efficiently captures the majority of critical features within transaction data while optimizing computational efficiency. By aligning with the observed data distribution, this approach balances model performance and practical functionality, minimizing the need to process excessively long sequences. Although some opcode sequences may exceed these thresholds, this strategy prioritizes the efficient handling of most cases, thereby maintaining both accuracy and computational efficiency.

5.4.1. Opcode relationship studies

The dataset used in this study comprises opcode sequences extracted from blockchain transactions. To explore the relationships among these opcodes and generate corresponding word vectors, we employ Word2Vec models to capture their semantic features.



(a) PUSH (b) SWAP

(c) DUP (d) LOG

Fig. 9. The Cosine Similarity Matrix of PUSH, SWAP, DUP, and LOG.

Although the official Gensim library provides numerous pre-trained models for large corpora, the semantics of Ethereum Virtual Machine (EVM) opcodes differ significantly from natural language, necessitating the retraining of a specialized model. Using the Gensim library, we apply the skip-gram algorithm with negative sampling to train a Word2Vec model tailored to opcodes, enabling the generation of word vectors that effectively capture their semantic properties.

Initially, the opcode sequences from the original transactions were preprocessed and normalized to ensure consistency. These preprocessed sequences were then used to train the Word2Vec model. During training, each opcode was mapped to a word vector, resulting in a high-dimensional dictionary encompassing 142 opcodes, as defined in the latest Ethereum Yellow Paper. Opcodes with similar functions are expected to have smaller Euclidean distances and higher cosine similarity values in their word vectors. This training process successfully captures the semantic relationships among opcodes, providing a robust foundation for subsequent model applications.

Notably, four operation types—PUSH, SWAP, DUP, and LOG—are the most frequently used in the EVM. Fig. 9 presents the cosine similarity matrix for these operations, illustrating their distribution within the EVM semantic space. Further statistical analysis in Fig. 10 reveals significant variations in the frequency of these opcode types, leading to distinct semantic features in the Word2Vec model. For example, PUSH1 and PUSH2, which push 1-byte and 2-byte values onto the stack, respectively, are frequently used due to the prevalence of 2-byte constant values (e.g., gas, transfer amounts, jump addresses, and block parameters) in Ethereum transaction execution. This pattern is evident in the cosine similarity matrix heatmap, where darker regions indicate smaller cosine distances between the word vectors of PUSH1 and PUSH2, reflecting their semantic proximity. Similar patterns are observed for other high-frequency operations, such as SWAP, DUP, and LOG. By capturing these semantic similarities, the model gains a deeper understanding of EVM opcodes, enhancing its ability to analyze and detect smart contract behaviors.

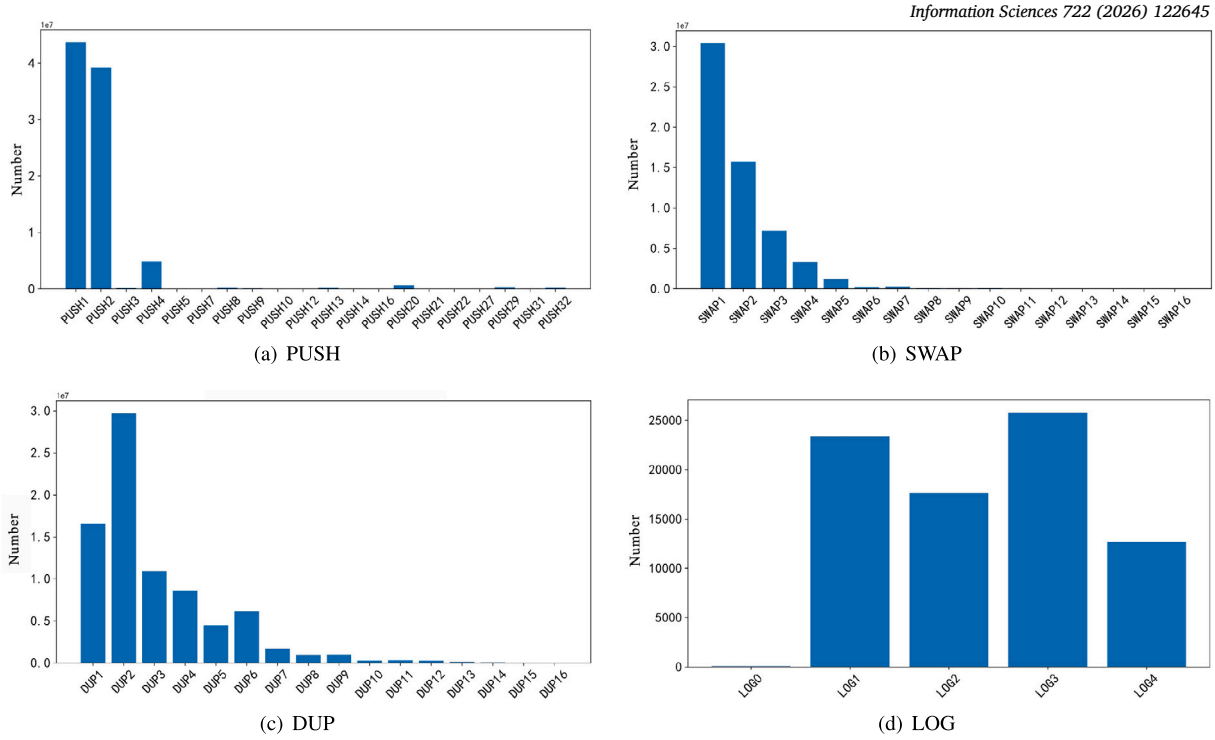


Fig. 10. The Number Distribution of PUSH, SWAP, DUP, and LOG.

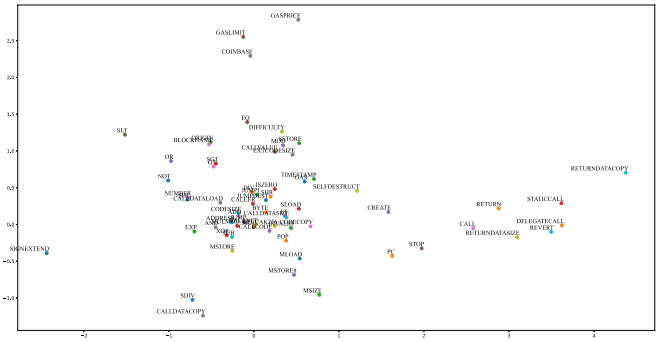


Fig. 11. T-SNE Plot of Embedding Vectors for Opcodes.

Fig. 11 presents a t-SNE plot visualizing the relationships among other EVM opcodes, excluding PUSH, SWAP, DUP, and LOG. In this plot, opcodes with similar functionalities are clustered closely together. For instance, CALL, STATICCALL, and DELEGATECALL form a cluster because they all represent calls to smart contracts. Similarly, opcodes such as OR, NOT, GT, and SHL are grouped together because they reflect the most basic arithmetic and comparison functions in the EVM. These results demonstrate that the trained Word2Vec model effectively captures the semantic relationships among opcodes, enabling robust analysis of their behavioral patterns.

5.4.2. Ablation studies

This study's framework focuses on three primary feature types: opcode sequence features, behavioral features of opcode sequences, and inter-opcode relationships. To assess the contribution of each feature type, we systematically removed individual feature modules from the model architecture and evaluated the resulting performance. Additionally, the bidirectional long short-term memory (BiLSTM) module's flexible structure allowed us to adjust LSTM directionality to investigate the impact of different model configurations.

The performance of the smart contract attack detection model was systematically evaluated under various conditions, with results presented in Figs. 12– 15. Fig. 12 demonstrates the model's robust performance across diverse attack scenarios, with area under the curve (AUC) values approaching 1 for all attack categories, indicating high accuracy and stability in detecting various attack types. Fig. 13 assesses the model's detection efficiency when the behavioral feature chain module is excluded, highlighting its adaptability and robustness despite reduced feature dimensionality. Conversely, Fig. 14 illustrates the impact of omitting opcode sequence features,

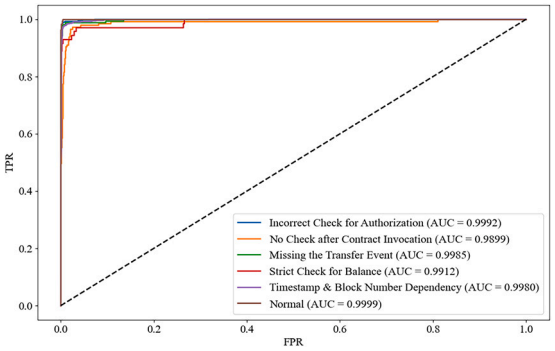


Fig. 12. ROC Curve of the Feature Fusion Method.

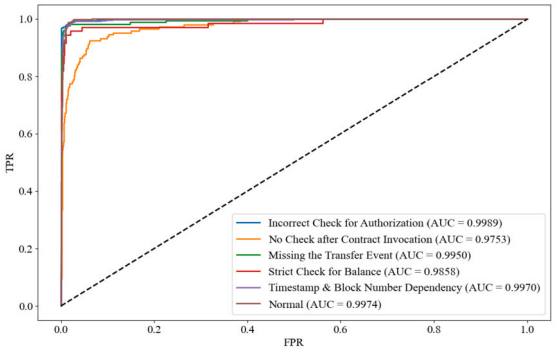


Fig. 13. ROC Curve of the Method Without Behavioral Feature Chain.

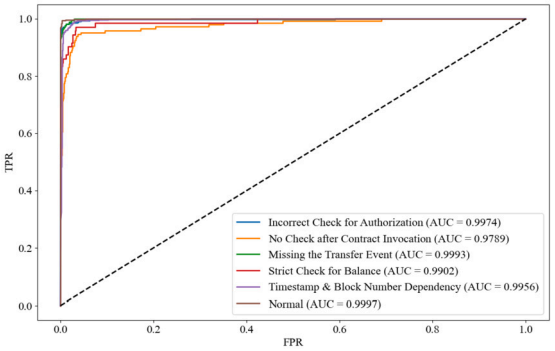


Fig. 14. ROC Curve of the Method Without Opcode Sequence.

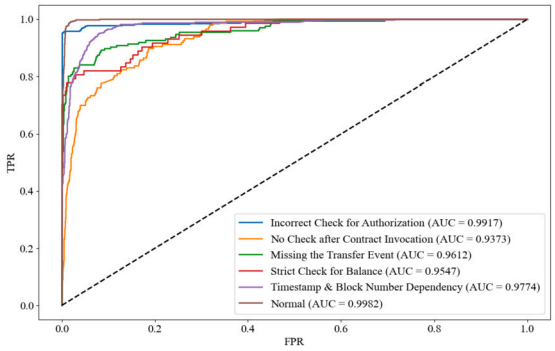


Fig. 15. ROC Curve of the Method Without Bi-LSTM.

Table 5
The Ablation Experimental Results.

Method	Accuracy	F1-score
Without Bi-LSTM	0.8763	0.8573
Without Opcode Sequence	0.9514	0.9503
Without Behavioral Feature Chain	0.9581	0.9581
Feature Fusion Model	**0.9721**	**0.9721**

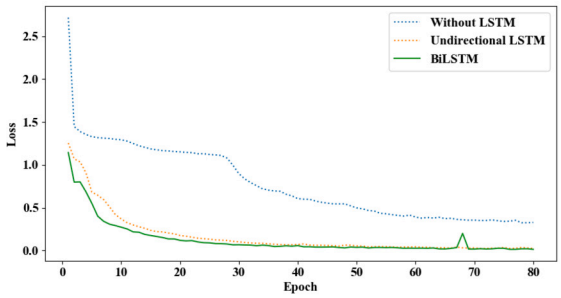


Fig. 16. Comparisons of Loss of the LSTM with Different Structures.

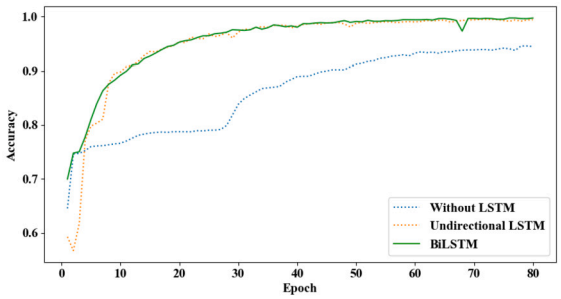


Fig. 17. Comparisons of Accuracy of the LSTM with Different Structures.

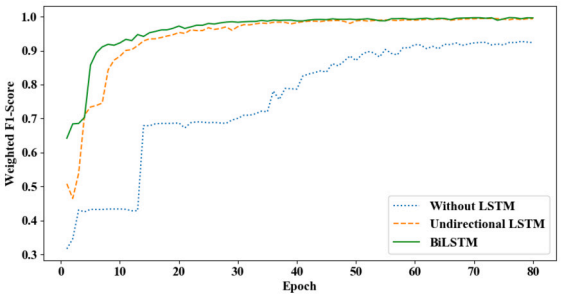


Fig. 18. Comparisons of Weighted F1-Score of the LSTM with Different Structures.

evidenced by a decline in AUC values for specific attack categories, underscoring their critical role in effective attack detection. Fig. 15 evaluates the effect of removing the BiLSTM layer, revealing a significant AUC reduction, particularly for the “no check after contract invocation” category, which highlights the BiLSTM’s unique ability to capture sequence dependencies among opcodes. Collectively, these findings confirm that the synergy among feature modules significantly enhances the model’s detection capabilities, enabling robust adaptation to diverse attack detection scenarios.

Further quantitative analysis of feature module contributions is detailed in Table 5. Removing the BiLSTM layer results in a 9.58% reduction in test set accuracy, emphasizing its essential role in feature fusion. Similarly, the opcode sequence and behavioral feature chain modules contribute significantly, improving the weighted F1-score by 1%–3%. These results indicate that the modules complement each other in feature extraction and decision-making, collectively enhancing the model’s overall performance.

The BiLSTM layer was incorporated into the feature fusion model to effectively capture complex inter-opcode relationships. To evaluate its impact, we conducted experiments comparing scenarios with no LSTM layer, a unidirectional LSTM layer, and a BiLSTM layer, while keeping other network parameters constant. As shown in Figs. 16–18, incorporating an LSTM layer—whether

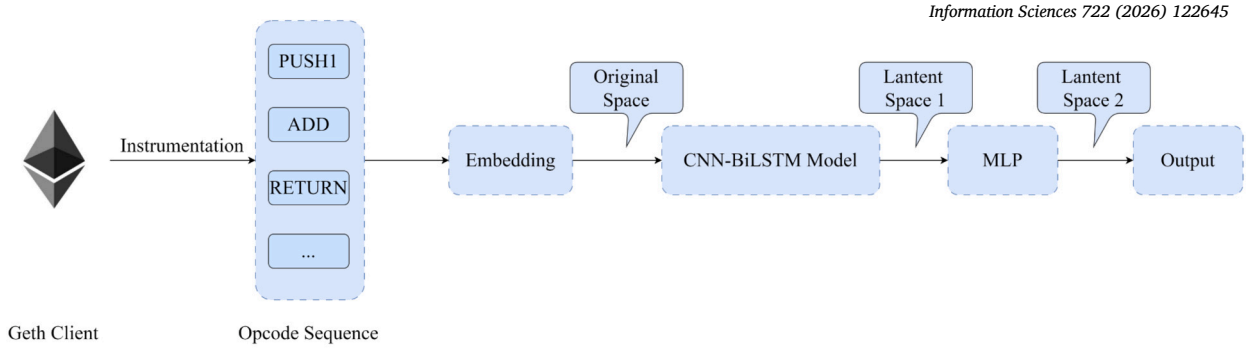


Fig. 19. Feature Space Definition.

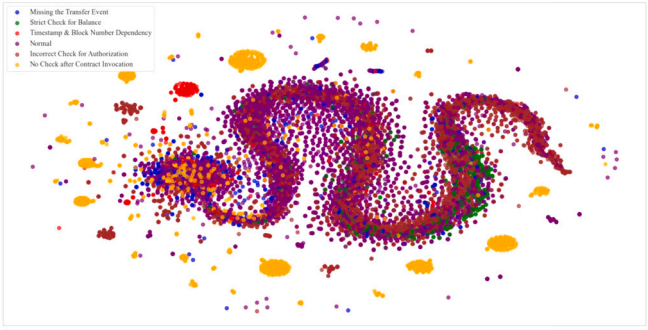


Fig. 20. T-SNE Plot in the Native Feature Space.

unidirectional or bidirectional—significantly accelerates model convergence and improves performance compared to models without an LSTM layer, demonstrating the importance of modeling opcode relationships in detection tasks. Notably, the BiLSTM model exhibits comparable performance to the unidirectional LSTM in terms of accuracy and weighted F1-score, both underscoring the value of capturing sequence dependencies in opcode analysis.

5.4.3. Impact of feature fusion model

To evaluate the impact of each processing stage on the model's performance, we defined three distinct feature spaces, as depicted in Fig. 19:

Native Feature Space: This initial layer captures the raw information extracted from the original opcode sequence. Features in this space represent the numerical properties of opcodes, serving as the input for subsequent processing stages.

Latent Space 1: Following processing by the CNN-BiLSTM model, features are mapped into Latent Space 1. This stage integrates the convolutional network's ability to extract local patterns with the BiLSTM's capacity to model temporal dependencies. The resulting features incorporate contextual information from both the opcode sequence and the behavioral feature chain, enhancing the model's sensitivity to attack-related behaviors.

Latent Space 2: In the final stage, features are refined by the multilayer perceptron (MLP) hidden layers and projected into Latent Space 2. These high-level feature representations are optimized to improve classification performance, enabling effective differentiation among attack categories.

To validate the model's feature learning at each stage, we employed t-SNE visualizations to assess its interpretative depth and classification performance. Fig. 20 illustrates the feature distribution in the native feature space, where data points from different attack categories show initial clustering. However, the boundaries between categories remain indistinct, indicating that the model has not yet fully differentiated attack features at this stage.

Fig. 21 depicts the feature projection into Latent Space 1 after processing by the CNN-BiLSTM module. Here, data points for different attack types form tighter clusters with clearer separation, demonstrating that the CNN-BiLSTM module effectively captures local opcode patterns and temporal dependencies. This improved clustering reflects the model's enhanced ability to extract intrinsic features relevant to attack detection.

Fig. 22 presents the feature distribution in Latent Space 2 following MLP processing. Compared to earlier stages, the clusters exhibit significantly clearer boundaries with minimal overlap between categories. This indicates that the MLP layer optimizes high-level features, enhancing their discriminative power and enabling precise differentiation of attack types. The distinct clustering in Latent Space 2 validates the model's deep understanding of attack characteristics and underscores the efficacy of its multi-stage feature processing architecture.

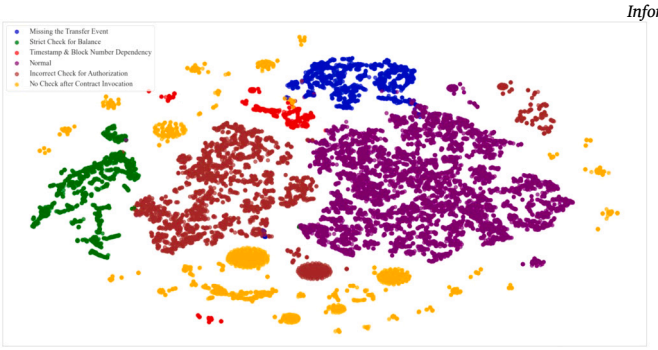


Fig. 21. T-SNE Plot in the Latent Space 1.

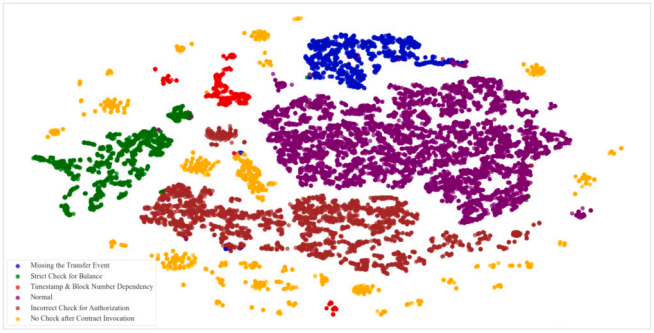


Fig. 22. T-SNE Plot in the Latent Space 2.

Table 6
Performance Comparison with Other Tools.

Method	Total	Timeout or Self-destruction	Remaining	Flagged
Mythril	438	88	350	50 (28)
Feature Fusion Model	**438**	**0**	**438**	**228**

In summary, the t-SNE visualizations across these three feature spaces illustrate the model's progressive ability to learn complex behavioral representations from opcode sequences. Each processing stage refines the feature representations, ultimately enhancing the model's capacity to accurately distinguish between various attack types.

5.4.4. Comparison with other tools

This study focuses on detecting opcode sequences generated during the execution of smart contract transactions. Traditional static analysis tools, such as Mythril, primarily target the bytecode of smart contracts. For comparative experiments, we extracted opcode sequences from 438 contracts in our database, comprising 238 contracts with vulnerabilities related to timestamp & block number dependencies and 200 normal contracts. Additionally, we extracted the bytecodes for all 438 contracts.

We evaluated the open-source version of Mythril on this dataset. Of the 438 contracts, 9 were incomplete due to timeouts, and 79 failed to yield detection results due to contract self-destruction. Among the remaining 350 contracts, Mythril identified 50 as exhibiting timestamp & block number dependency vulnerabilities. However, only 28 of these detections aligned with the results of our feature fusion model.

In contrast, our feature fusion model demonstrated superior performance across the entire dataset, successfully identifying 228 contract samples with potential timestamp & block number dependency vulnerabilities. Notably, it was unaffected by timeouts or contract self-destruction events. This significant performance advantage underscores the model's ability to address the limitations of static analysis tools, particularly in detecting vulnerabilities that may be missed when source code is unavailable and only bytecode is accessible.

A detailed comparison of detection results between Mythril and our model is presented in Table 6. Mythril employs conventional techniques, such as taint analysis, which are difficult to apply across all runtime scenarios, resulting in limited scalability and reduced detection performance. In contrast, our feature fusion model effectively captures anomalous patterns during transaction execution, significantly improving the accuracy of attack detection in smart contracts.

6. Conclusion

This study proposes an innovative approach for detecting attacks on smart contracts through feature fusion technology. The method conducts a comprehensive analysis of opcode behavior across four dimensions: operation objects, action behaviors, functional categories, and gas consumption. By integrating convolutional neural networks (CNNs) and bidirectional long short-term memory (BiLSTM) models, our approach highlights the pivotal role of opcode sequence analysis in bolstering smart contract security. The model achieves a detection accuracy of 97.21% and a weighted F1-score of 97.21%, demonstrating its effectiveness and significant potential for enhancing the security framework of smart contracts.

Despite these promising results, several directions for future research warrant exploration:

- **Exploring Additional Feature Dimensions**: Future work will investigate supplementary feature dimensions to enhance the model's detection efficiency. For example, analyzing temporal patterns within opcode sequences or incorporating external data, such as transaction source addresses, could yield deeper insights.
- **Leveraging Multimodal Approaches**: Further research will explore multimodal techniques to improve smart contract attack detection. By integrating diverse data sources—such as contract code structure, transaction history, and network behavior—the model can achieve a more comprehensive contextual understanding, potentially improving its performance.

CRediT authorship contribution statement

Peiqiang Li: Writing – original draft, Methodology, Investigation. **Guojun Wang:** Supervision, Funding acquisition. **Wanyi Gu:** Writing – review & editing, Data curation, Conceptualization. **Xubin Li:** Methodology, Investigation, Formal analysis. **Xiangyong Liu:** Software, Resources, Project administration. **Yuheng Zhang:** Visualization, Validation, Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China under Grant 62372121, and in part by the National Key Research and Development Program of China (2020YFB1005804).

Data availability

Data will be made available on request.

References

[1] P. Sharma, R. Jindal, M.D. Borah, A review of smart contract-based platforms, applications, and challenges, Clust. Comput. 26 (1) (2023) 395–421.
[2] M.I. Mehar, C.L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H.M. Kim, M. Laskowski, Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack, J. Cases Inf. Technol. 21 (1) (2019) 19–32.
[3] FarmEOS, https://www.bitdegree.org/crypto-tracker/top-eos-dapps/farmeos, 2019.
[4] Playgames, https://www.bitdegree.org/crypto-tracker/top-eos-dapps/playgames, 2019.
[5] EOSlots, https://www.mycryptopedia.com/top-40-eos-dapps/, 2019.
[6] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, X. Zhang, Towards saving money in using smart contracts, in: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, 2018, pp. 81–84.
[7] P. Zheng, Z. Zheng, X. Luo, Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 740–751.
[8] Y. Liu, Y. Li, S.-W. Lin, C. Artho, Finding permission bugs in smart contracts with role mining, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 716–727.
[9] V. Wüstholz, M. Christakis, Harvey: a greybox fuzzer for smart contracts, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1398–1409.
[10] V. Wustholz, M. Christakis, Targeted greybox fuzzing with static lookahead analysis, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE Computer Society, 2020, pp. 789–800.
[11] M. Eshghie, C. Artho, D. Gurov, Dynamic vulnerability detection on smart contracts using machine learning, in: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, 2021, pp. 305–312.
[12] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, et al., Soda: a generic online detection framework for smart contracts, in: NDSS, 2020.
[13] M. Zhang, X. Zhang, Y. Zhang, Z. Lin, Txspector: uncovering attacks in Ethereum from transactions, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2775–2792.
[14] W. Zhao, W. Zhang, J. Wang, H. Wang, C. Wu, Smart contract vulnerability detection scheme based on symbol execution, J. Comput. Appl. 40 (4) (2020) 947.
[15] S. So, M. Lee, J. Park, H. Lee, H. Oh, Verismart: a highly precise safety verifier for Ethereum smart contracts, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1678–1694.
[16] S. So, S. Hong, H. Oh, Smartest: effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution, in: 30th USENIX Security Symposium (USENIX Security, vol. 21, 2021, pp. 1361–1378.

[17] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, L. Wu, Wasai: uncovering vulnerabilities in wasm smart contracts, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 703–715.
[18] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, X. Zhang, Rethinking smart contract fuzzing: fuzzing with invocation ordering and important branch revisiting, IEEE Trans. Inf. Forensics Secur. 18 (2023) 1237–1251.
[19] Q. Han, L. Wang, H. Zhang, L. Shi, D. Wang, Ethchecker: a context-guided fuzzing for smart contracts, J. Supercomput. (2024) 1–27.
[20] H. Yang, X. Gu, X. Chen, L. Zheng, Z. Cui, Crossfuzz: cross-contract fuzzing for smart contract vulnerability detection, Sci. Comput. Program. 234 (2024) 103076.
[21] Z. Yang, H. Lei, Fether: an extensible definitional interpreter for smart-contract verifications in coq, IEEE Access 7 (2019) 37 770–37 791.
[22] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, M. Vechev, Verx: safety verification of smart contracts, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1661–1677.
[23] X. Wang, S. Tian, W. Cui, Contractcheck: checking Ethereum smart contracts in fine-grained level, IEEE Trans. Softw. Eng. (2024).
[24] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, Y. Liu, Gptscan: detecting logic vulnerabilities in smart contracts by combining gpt with program analysis, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[25] Y. Chen, Z. Sun, Z. Gong, D. Hao, Improving smart contract security with contrastive learning-based vulnerability detection, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–11.
[26] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, S. Li, Scvhunter: smart contract vulnerability detection based on heterogeneous graph attention network, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[27] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254–269.
[28] B. Jiang, Y. Liu, W.K. Chan, Contractfuzzer: fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 259–269.
[29] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, Reguard: finding reentrancy bugs in smart contracts, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, 2018, pp. 65–68.
[30] I. Grishchenko, M. Maffei, C. Schneidewind, A semantic framework for the security analysis of Ethereum smart contracts, in: Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7, Springer, 2018, pp. 243–269.
[31] S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: analyzing safety of smart contracts, in: Ndss, 2018, pp. 1–12.
[32] C.F. Torres, J. Schütte, R. State, Osiris: hunting for integer bugs in Ethereum smart contracts, in: Proceedings of the 34th Annual Computer Security Applications Conference, 2018, pp. 664–676.
[33] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, H. Chen, Smart contract vulnerability detection combined with multi-objective detection, Comput. Netw. 217 (2022) 109289.
[34] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, H. Chen, A novel smart contract vulnerability detection method based on information graph and ensemble learning, Sensors 22 (9) (2022) 3581.
[35] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, H. Chen, Cbgru: a detection method of smart contract vulnerability based on a hybrid model, Sensors 22 (9) (2022) 3577.
[36] F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, L. Khan, Vscl: automating vulnerability detection in smart contracts with deep learning, in: 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), IEEE, 2021, pp. 1–9.
[37] Z. Gao, L. Jiang, X. Xia, D. Lo, J. Grundy, Checking smart contracts with structural code embedding, IEEE Trans. Softw. Eng. 47 (12) (2020) 2874–2891.
[38] M. Rodler, W. Li, G.O. Karame, L. Davi, Sereum: protecting existing smart contracts against re-entrancy attacks, arXiv preprint, arXiv:1812.05934, 2018.
[39] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, Y. Zohar, Online detection of effectively callback free objects with applications to smart contracts, Proc. ACM Program. Lang. 2 (POPL) (2017) 1–28.
[40] W. Gu, G. Wang, P. Li, X. Liu, Y. Zhang, G. Zhai, Discovering attacks against smart contracts using opcode sequences with feature fusion, in: 2024 IEEE Smart World Congress (SWC), IEEE, 2024, pp. 969–975.
[41] M. Javaid, A. Haleem, R.P. Singh, R. Suman, S. Khan, A review of blockchain technology applications for financial services, BenchCouncil Trans. Benchmarks Stand. Eval. 2 (3) (2022) 100073.
[42] R.K. Singh, R. Mishra, S. Gupta, A.A. Mukherjee, Blockchain applications for secured and resilient supply chains: a systematic literature review and future research agenda, Comput. Ind. Eng. 175 (2023) 108854.
[43] W. Gu, G. Wang, P. Li, X. Li, G. Zhai, X. Li, M. Chen, Detecting unknown vulnerabilities in smart contracts with multi-label classification model using cnn-bilstm, in: International Conference on Ubiquitous Security, Springer, 2022, pp. 52–63.
[44] P. Li, G. Wang, X. Xing, X. Li, J. Zhu, Detecting unknown vulnerabilities in smart contracts using opcode sequences, Connect. Sci. 36 (1) (2024) 2313853.
[45] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M.H. Au, X. Zhang, An adaptive gas cost mechanism for Ethereum to defend against under-priced dos attacks, in: Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13, Springer, 2017, pp. 3–24.
[46] G. Wood, et al., Ethereum: a secure decentralised generalised transaction ledger, Ethereum Proj. Yellow Pap. 151 (2014) (2014) 1–32.