

Malicious Code Detection in Smart Contracts via Opcode Vectorization

HUANHUAN ZOU, School of Cyberspace Security, Hainan University, China

ZONGWEI LI, School of Cyberspace Security, Hainan University, China

XIAOQI LI, School of Cyberspace Security, Hainan University, China

With the booming development of blockchain technology, smart contracts have been widely used in finance, supply chain, Internet of things and other fields in recent years. However, the security problems of smart contracts become increasingly prominent. Security events caused by smart contracts occur frequently, and the existence of malicious codes may lead to the loss of user assets and system crash. In this paper, a simple study is carried out on malicious code detection of intelligent contracts based on machine learning. The main research work and achievements are as follows: Feature extraction and vectorization of smart contract are the first step to detect malicious code of smart contract by using machine learning method, and feature processing has an important impact on detection results. In this paper, an opcode vectorization method based on smart contract text is adopted. Based on considering the structural characteristics of contract opcodes, the opcodes are classified and simplified. Then, N-Gram (N=2) algorithm and TF-IDF algorithm are used to convert the simplified opcodes into vectors, and then put into the machine learning model for training. In contrast, N-Gram (N=2) algorithm and TF-IDF algorithm are directly used to quantify opcodes and put into the machine learning model training. Judging which feature extraction method is better according to the training results. Finally, the classifier chain is applied to the intelligent contract malicious code detection.

Additional Key Words and Phrases: Smart contract; Malicious code detection; Operation code

ACM Reference Format:

Huanhuan Zou, Zongwei Li, and Xiaoqi Li. 2025. Malicious Code Detection in Smart Contracts via Opcode Vectorization. 1, 1 (April 2025), 22 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Authors' Contact Information: Huanhuan Zou, School of Cyberspace Security, Hainan University, Haikou, China; Zongwei Li, School of Cyberspace Security, Hainan University, Haikou, China; Xiaoqi Li, School of Cyberspace Security, Hainan University, Haikou, China, csxqli@ieee.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

1 Introduction

With the rapid development of blockchain technology, smart contracts have seen increasingly widespread applications in finance, supply chains, insurance, and other fields. A smart contract is a computer protocol that automatically executes predefined conditions and logic. It is essentially a piece of code designed to facilitate, verify, or enforce the terms of a contract [38]. Although smart contracts offer advantages such as decentralization, transparency, and immutability, they still face security risks. In 2017, the Parity Multisig wallet had a fatal unauthorized access vulnerability, resulting in \$300 million being frozen [33]. In 2018, an integer overflow vulnerability in the BEC campaign caused the instantaneous evaporation of over \$900 million. Security incidents in blockchain can be categorized into the following types: trading platform security incidents, user account and key security incidents, miner node system security incidents, smart contract security incidents, and other security incidents. Among these, smart contract security incidents account for approximately 6%, but the losses incurred are substantial, accumulating to \$5.1 billion by the end of 2022. Therefore, addressing smart contract security issues is urgent [7].

Traditional static and dynamic analysis methods face certain limitations in detecting malicious code in smart contracts, including low analysis efficiency, high false-positive rates, and an inability to identify new and complex vulnerabilities [42]. In recent years, machine learning technology has achieved remarkable results in fields such as computer vision and natural language processing, opening up new research directions for detecting malicious code in smart contracts.

Machine learning-based malicious code detection technology for smart contracts can more effectively identify and prevent vulnerabilities and malicious behaviors in smart contracts [11]. It enables automated and intelligent auditing of smart contract code, reducing the cost of manual audits while improving audit efficiency and accuracy [1]. In the future, as research deepens and technology advances, machine learning-based malicious code detection technology is certain to play an even greater role in the field of smart contracts.

2 Background

Smart contracts are self-executing and self-verifying computer protocols designed to facilitate, verify, or enforce the negotiation of contracts between parties [4]. Smart contracts enable trustworthy transactions without third-party intermediaries, thereby reducing the risks of fraud and default. The concept of smart contracts was first proposed by computer scientist Nick Szabo in 1994 [13].

Smart contracts are one of the core applications of blockchain technology, implemented by deploying programmable scripts on the blockchain [39]. These scripts can automatically trigger, execute, and verify relevant operations when specific conditions are met [43]. Since smart contracts run on decentralized blockchain networks, they have the following characteristics:

(1) Transparency: The code and execution results of smart contracts are visible to all participants, ensuring transaction transparency.

(2) Immutability[37]: Due to the properties of blockchain technology, once smart contracts are deployed on the blockchain, their code cannot be modified, ensuring the reliability and security of the contract.

(3) Automatic Execution: Smart contracts execute automatically when specific conditions are met, without requiring manual intervention.

(4) Decentralization: Smart contracts operate on distributed blockchain networks, independent of any centralized institution, reducing centralization risks.

3 Method

3.1 Feature Extraction

The features obtained from Ethereum smart contracts cannot be directly used as machine learning inputs, thus vectorization of smart contract features becomes a necessary step [46]. This paper employs an opcode-based text vectorization method that first categorizes opcodes according to their semantic meanings in the Ethereum Virtual Machine (EVM), then transforms the simplified opcodes into vectors using N-Gram and TF-IDF algorithms.

3.1.1 Common Feature Extraction Methods.

(1) Opcodes Feature

Opcodes represent the basic execution units of smart contracts. By analyzing opcode sequences, we can capture the execution logic of contracts. Common methods include using N-Gram and TF-IDF algorithms to convert opcode sequences into vector representations.

(2) Control Flow Feature

Control flow features describe the execution order between basic blocks in contracts. These can be extracted by constructing Control Flow Graphs (CFGs). Subsequently, graph embedding techniques (e.g., Graph2Vec) can convert CFGs into vector representations [8].

(3) Data Flow Feature

Data flow features characterize data transmission and transformation within contracts. These can be obtained by building Data Flow Graphs (DFGs), which can then be vectorized using graph embedding methods like Graph2Vec [48].

(4) Function Call Feature

Function call features represent invocation relationships between functions in contracts. This information can be extracted by analyzing smart contract source code or bytecode, then transformed into vectors using either N-Gram algorithms or graph embedding techniques [19].

(5) Variable and Data Type Feature

These features describe the variables and their types used in contracts. By examining source code, such features can be extracted and converted into vector representations using text representation methods like Bag-of-Words.

The image shows a screenshot of a web-based interface for viewing smart contract bytecode. At the top, there are two buttons: "Decompile ByteCode" and "Switch to Opcodes View". Below the buttons, a large text area contains a long string of hexadecimal characters representing the bytecode. The text is wrapped across multiple lines. The interface also shows a file name "Contract Creation Code" and a small icon of a code editor.

Fig. 1. Bytecode of Smart Contracts

(6) Code Complexity Feature

Code complexity features measure programming complexity metrics such as loop nesting depth and conditional branch counts. These can be obtained through static code analysis and directly used as numerical inputs for machine learning models.

3.2 Problem Analysis

In recent years, machine learning-based smart contract vulnerability detection techniques have made significant progress. These methods analyze smart contract source code, bytecode, and opcodes to better detect potential vulnerabilities [50].

Giacomo Ibba [15] et al. extracted Abstract Syntax Trees (AST) from Solidity source code and then parsed the resulting AST. Based on the structural and syntactic definitions of the AST, they performed feature processing [9]. This approach leverages syntactic information from source code to extract features related to contract structure and behavior, thereby facilitating the identification of potential vulnerabilities [2].

Kim [16] conducted research based on bytecode characteristics. By using encoders to highlight unique portions of raw bytecode and treating them as contract attribute tags, their method provides training data for neural network models. This bytecode-level approach can detect potential vulnerabilities that might be overlooked at the source code level.

Tann [41] performed analytical research on smart contract opcode layers. They constructed an Ethereum opcode sequence model using Long Short-Term Memory (LSTM) networks for vulnerability detection. This method utilizes opcode-level information to identify potential vulnerabilities at the underlying instruction level.

Ivica Nikolić [35] et al. conducted large-scale analysis of Ethereum smart contracts, revealing that source code is available for only about 1% of approximately 1 million smart contracts. Consequently, bytecode and opcodes are substantially more accessible from Ethereum.

As shown in Fig 1 and 2, although both bytecode and opcodes are compiled from source code in Ethereum, feature extraction from bytecode may lead to semantic loss, making it difficult to adequately reflect structural features and call relationships in smart contracts. This often results in undetected vulnerabilities. Therefore, this paper extracts syntactic and semantic information characterizing vulnerabilities from smart contract opcode sequences, which better represents dataset characteristics.

```
</> Contract Creation Code
PUSH1 0x80
PUSH1 0x40
MSTORE
PUSH1 0x04
CALLDATASIZE
LT
PUSH2 0x03e8
JUMPI
PUSH1 0x00
```

Fig. 2. Opcodes of Smart Contracts

For opcode vectorization, text vectorization methods can indeed serve as references. However, directly applying algorithms like Word2Vec may inadequately capture sequential relationships between opcodes, potentially compromising program context representation. Thus, this paper employs N-Gram and TF-IDF algorithms for opcode vectorization.

The N-Gram algorithm better describes static program features by capturing sequential relationships between opcodes [10]. By grouping N adjacent opcodes, it preserves program context information to some extent. The TF-IDF algorithm measures the importance of specific features within the entire program, helping highlight opcodes with significant impacts on program behavior and thereby improving vector representation effectiveness.

Prior to vectorization, opcodes are categorized to prevent the curse of dimensionality and semantic loss. This simplification of dimensions during subsequent N-Gram processing and statistical feature extraction preserves contract information while reducing feature dimensionality, ultimately enhancing model training efficiency.

3.3 Opcodes Classification

Based on the semantic definitions of opcodes in the Ethereum Virtual Machine (EVM) [29], this paper categorizes opcodes into distinct classes. Excluding exception handling cases, opcodes with similar execution logic are grouped together. This classification approach simplifies the vectorization process while preserving critical contract information.

3.3.1 Operational Instructions. As Table 1, these fundamental instructions handle data processing, storage, and logging in smart contracts. They affect only stack data without altering program structure.

3.3.2 Predictable Variable Instructions. As Table 2, used to access blockchain-specific information, storing results in memory without affecting program structure.

3.3.3 Logic Instructions. As Table 3, perform bitwise operations (AND, OR, XOR) and logical negation (NOT), impacting only the top two stack values without altering program structure.

Table 1. Operational Instructions

Opcodes	Description
PUSH1-PUSH32	Pushes 1-32 bytes of data onto the stack
DUP1-DUP16	Duplicates stack items at positions 1-16
SWAP1-SWAP16	Swaps values between stack top and positions 1-16
LOG0-LOG4	Creates log entries in smart contracts

Table 2. Predictable Variable Instructions

Opcodes	Description
BLOCKHASH	Gets the hash of specified block
COINBASE	Gets current block beneficiary address
TIMESTAMP	Gets current block timestamp
NUMBER	Gets current block number
DIFFICULTY	Gets current block difficulty
GASLIMIT	Gets current block gas limit

Table 3. Logic Instructions

Opcodes	Description
AND	Bitwise AND operation
OR	Bitwise OR operation
XOR	Bitwise XOR operation
NOT	Logical NOT operation

3.3.4 *Arithmetic Instructions.* As Table 4, execute basic mathematical operations, affecting only the top two stack values.

Table 4. Arithmetic Instructions

Opcodes	Description
ADD	Addition
MUL	Multiplication
SUB	Subtraction
DIV	Unsigned division
SDIV	Signed division
MOD	Unsigned modulo
SMOD	Signed modulo
ADDMOD	Modular addition
MULMOD	Modular multiplication
EXP	Exponentiation

3.3.5 *Comparison Instructions.* As Table 5, perform relational operations crucial for conditional jumps. Results (0/1) are pushed onto the stack without directly affecting program structure.

Table 5. Comparison Instructions

Opcodes	Description
LT	Less than (unsigned)
GT	Greater than (unsigned)
SLT	Less than (signed)
SGT	Greater than (signed)
EQ	Equality comparison
ISZERO	Tests if value equals zero

3.3.6 *Address and Balance Instructions.* As Table 6, provide address-related queries for transaction processing (transfers, authorizations), storing results in memory.

Table 6. Address and Balance Instructions

Opcodes	Description
ADDRESS	Gets current contract address
BALANCE	Queries address balance
ORIGIN	Gets transaction origin address
CALLER	Gets immediate caller address

3.3.7 *CALL-family Instructions.* As Table 7, enable inter-contract calls by constructing msg.data structures containing method selectors and parameters. The called contract accesses the caller address through the msg.sender global variable. Boolean return values indicate execution status (0=failure, 1=success), affecting program flow.

Table 7. CALL-family Instructions

Opcodes	Description
CALL	Invokes another contract's function
CALLCODE	Executes target contract code in current context
DELEGATECALL	Preserves original msg.sender and msg.value
STATICCALL	Performs static calls without state modification

3.3.8 *SSTORE-family Instructions.* As Table 8, handle temporary (memory) and persistent (storage) data operations. Storage modifications directly affect contract state.

Table 8. SSTORE-family Instructions

Opcodes	Description
MLOAD	Loads 32-byte word from memory
MSTORE	Stores 32-byte word to memory
MSTORE8	Stores single byte to memory
SLOAD	Loads word from storage
SSTORE	Stores word to storage

Table 9. Termination Instructions

Opcodes	Description
RETURN	Terminates with specified return data
STOP	Halts contract execution
REVERT	Reverts state changes with return data
INVALID	Triggers on undefined opcodes
SELFDESTRUCT	Destroys current contract

3.3.9 *Termination Instructions.* As Table 9. handle execution conclusions including normal termination, exceptions, and self-destruction. Subsequent execution requires JUMP instructions, directly affecting program structure.

3.3.10 *Jump Instructions.* As Table 10, implement control flow logic through conditional/unconditional jumps, directly modifying program execution paths.

Table 10. Jump Instructions

Opcodes	Description
JUMPDEST	Marks valid jump destination
JUMP	Unconditional jump
JUMPI	Conditional jump (executes if top stack value \neq 0)

3.4 Opcode Vectorization

In the analysis of smart contract opcodes, drawing an analogy to word vectorization techniques in natural language processing is an effective approach. Mapping opcodes into vector space can reveal relationships and similarities between opcodes, thereby providing valuable information for subsequent malicious code detection and vulnerability identification tasks. Using the N-Gram algorithm can preserve some sequential information of opcodes and capture associations between adjacent opcodes. This is because the N-Gram algorithm divides text into sequences of N consecutive elements to capture collocations and relationships between these elements. Thus, we can obtain a feature representation that reflects the contextual relationships between opcodes. Additionally, the TF-IDF algorithm can be used to measure the importance of

opcodes within the entire program. By calculating the term frequency (TF) and inverse document frequency (IDF) of each opcode, we can assign a weight to each opcode to highlight those that have a greater impact on the program's behavior.

3.4.1 N-Gram Algorithm. The N-Gram algorithm is a statistical language model-based method used to analyze and represent relationships between adjacent elements (such as words or characters) in text. The core idea of the N-Gram algorithm is to divide text into sequences of N consecutive elements to capture collocations and relationships between these elements. Each sequence is called a gram, and each gram represents a dimension of the feature vector. Considering the large number of smart contract opcodes, we set $N=2$ (i.e., bigram) to keep the dimensionality manageable. The specific steps are as follows:

- (1) First, split the smart contract's opcode sequence into pairs of adjacent opcodes (i.e., bigrams). For example, given the opcode sequence A B C D, the generated bigrams are: (A,B), (B,C), (C,D).
- (2) Calculate the frequency of each bigram in the opcode sequence. This can be achieved by traversing the entire opcode sequence and counting the occurrences of each bigram.
- (3) Convert the bigram frequency data into vector representation. Create a vector where each element corresponds to a possible bigram, and set the value of each element to the frequency of the corresponding bigram in the opcode sequence. For example, if all possible bigrams are $\{(A,B), (A,C), (B,C), (B,D), (C,D)\}$, the vector representation of the opcode sequence A B C D might be [1, 0, 1, 0, 1].

3.4.2 TF-IDF Algorithm. The TF-IDF algorithm is a common text mining method used to measure the importance of words in a document. It is based on two main concepts: Term Frequency (TF) and Inverse Document Frequency (IDF). By combining these two concepts, the TF-IDF algorithm can highlight words that appear frequently in a specific document but are rare across the entire document set, thereby assigning them higher weights.

Term Frequency (TF) refers to the number of times a term appears in a document. Generally, the higher the term frequency, the greater its importance in the document. The formula is as follows:

$$TF(t, D) = \frac{t}{D} \quad (1)$$

where t represents a specific bigram in a smart contract, and D represents the total number of bigrams in that smart contract.

Inverse Document Frequency (IDF) is a correction applied to term frequency to reduce the influence of high-frequency terms (here, high-frequency bigrams). The formula is as follows:

$$IDF(t, N) = \log \left(\frac{N}{n_t + 1} \right) \quad (2)$$

where N is the total number of smart contracts in the training set, n_t is the number of smart contracts containing a specific bigram, and $n_t + 1$ is added to prevent division by zero in case no smart contract contains the bigram.

The TF-IDF value is calculated as:

$$TF-IDF(t, D, N) = TF(t, D) \times IDF(t, N) \quad (3)$$

3.4.3 Opcodes Simplification. To reduce dimensionality, we simplify the number of opcodes before applying the N-Gram algorithm. Based on the roles and impacts of various opcodes discussed in Section 3.3, this paper adopts the following simplification rules (Table 11):

After simplification, 35 opcodes remain: PUSH, DUP, SWAP, LOG, COUNT, PREDICT, JUDGE, COMPARISON, ADDRESS, JUMPDEST, JUMP, JUMPI, CALLVALUE, ALLDATALOAD, CALLDATASIZE, CALLDATACOPY, CALL, CODESIZE, CODECOPY, GASPRICE, CREATE, EXTCODESIZE, EXTCODECOPY, GAS, STOP, EQ, CALLCODE, DELEGATECALL, SELEGATECALL, SELFDESTRUCT, REVERT, MSTORE, SHA3, POP, RETURN.

Table 11. Instruction Simplification Mapping

Original Instruction	Simplified Instruction
Operation Instructions	PUSH1-PUSH16 → PUSH DUP1-DUP16 → DUP SWAP1-SWAP16 → SWAP LOG0-LOG4 → LOG
Arithmetic Instructions	All → COUNT
Predictable Instructions	All → PREDICT
Judgment Instructions	All → JUDGE
Comparison Instructions	All → COMPARISON
Address/Wallet Instructions	All → ADDRESS
Instructions affecting program structure	No simplification
Instructions not closely related to vulnerabilities	Removed

The 2-Gram algorithm is used to extract bigrams from the simplified contract opcodes. The extraction process is shown in Fig 3.

This process ultimately extracts 1225-dimensional bigrams, which effectively reflects the relationships between opcodes while avoiding dimensionality issues during model training.

Since smart contracts vary in length, the types of extracted bigrams may differ. If we only calculate TF-IDF values for bigrams present in a particular smart contract’s opcodes, the resulting vector dimensions would vary across contracts, creating difficulties for subsequent model training. To address this, we standardize the vector dimension to 1225 for all smart contracts. Each dimension corresponds to a specific bigram, and if a particular bigram doesn’t appear in a contract’s opcodes, its corresponding TF-IDF value is set to 0.

3.4.4 Pseudocode Implementation. Table 12 extracts all possible bigrams from the simplified opcodes, ultimately generating 1225-dimensional bigrams.

Table 13 categorizes smart contract opcodes according to the classification rules.

Table 14 divides each smart contract’s opcodes using the 2-Gram algorithm.

Table 15 calculates the TF-IDF values for each smart contract’s bigrams.

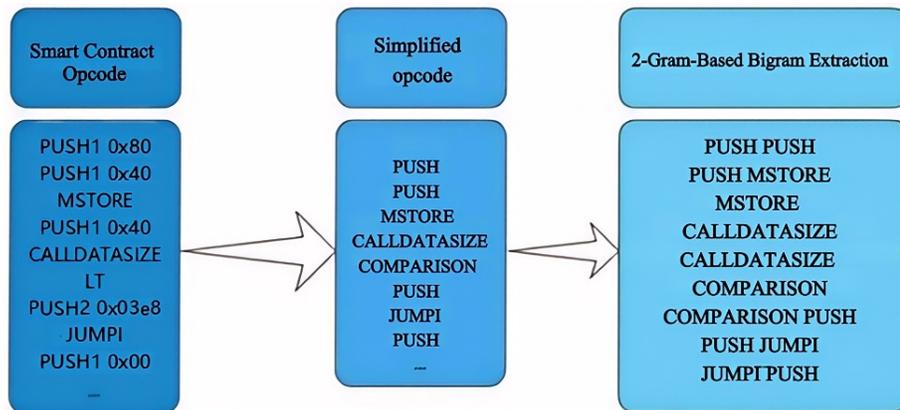


Fig. 3. Flowchart of bigram extraction from simplified opcodes

Table 12. Pseudocode for Constructing Bigrams

Construct all possible bigram combinations
1: Define opcode list opcodes = ['PUSH','DUP','SWAP',...]
2: Define an empty list all_bigrams for all possible combinations
3: for each element a in opcodes:
4: for each element b in opcodes:
5: Concatenate opcode a and opcode b with a space
6: Add the concatenated string to all_bigrams list

Table 13. Pseudocode for Opcode Classification Algorithm

Algorithm 1: Opcode Classification
Input: Pre-simplified opcode
Output: Classified opcode
Function simplify_opcode(opcode):
if opcode starts with "PUSH": return "PUSH"
elif opcode starts with "DUP": return "DUP"
elif opcode starts with "SWAP": return "SWAP"
elif opcode starts with "LOG": return "LOG"
elif opcode belongs to arithmetic instructions: return "COUNT"
elif opcode belongs to predictable instructions: return "PREDICT"
elif opcode belongs to judgment instructions: return "JUDGE"
elif opcode belongs to comparison instructions: return "COMPARISON"
elif opcode belongs to address/wallet instructions: return "ADDRESS"
else: return opcode

Table 14. Pseudocode for N-Gram Algorithm

<p>Algorithm 2: N-Gram Algorithm for Smart Contract Opcodes Input: Smart contract opcodes Output: Bigram frequency list</p>
<p>Function bigrams_frequency(contracts): Initialize an empty bigram frequency list bigrams_freq for each contract in contracts: Apply simplify_opcode function to each opcode in contract Create bigrams contract_bigrams from simplified opcodes Add contract_bigrams to bigrams_freq list return bigrams_freq</p>

Table 15. Pseudocode for TF-IDF Algorithm

<p>Algorithm 3 TF-IDF Algorithm Input: Smart contract opcodes Output: TF-IDF matrix</p>
<p>Function TF_IDF(contracts): Call bigrams_frequency to compute bigram frequencies Create index mapping bigram_to_index for all bigrams Initialize TF-IDF vectorizer with vocabulary=bigram_to_index, lowercase=False Fit and transform bigrams_freq using the vectorizer return TF-IDF matrix</p>

4 Experiment

4.1 Dataset Collection

Etherscan [40] is an Ethereum blockchain explorer and analytics platform. It allows users to query and retrieve information about transactions, addresses, blocks, tokens (including ERC-20 and ERC-721 tokens), and smart contracts on the Ethereum network. Etherscan provides developers, researchers, and other participants in the Ethereum ecosystem with a user-friendly interface to track and analyze activities on the Ethereum blockchain.

For this experiment, we manually downloaded the source code of 500 smart contracts published on <https://etherscan.io/contractsVerified>. We then decoded the bytecode into opcodes using an open-source opcode tool, copied each smart contract’s opcodes, and saved them as corresponding ".txt" files.

4.2 Label Annotation

We used the Ethereum static analysis tool Slither [12] to detect vulnerabilities in smart contracts. After configuring the Slither tool and selecting the compiler version (solc) according to the pragma solidity ^0.6.0 statement in the smart contract code, we performed the detection. The detection results for one sample

smart contract (as shown in Figure 4) revealed both access control vulnerabilities and reentrancy attack vulnerabilities.

```
F:\..\SourceCode> slither .\HONGKONGPEPE.sol
INFO:Detectors:
HONGKONGPEPE.swapBack() (HONGKONGPEPE.sol#349-366) sends eth to arbitrary user
  Dangerous calls:
  - marketingWallet.transfer(ethRemain / 2) (HONGKONGPEPE.sol#364)
  - devWallet.transfer(ethRemain / 2) (HONGKONGPEPE.sol#365)
HONGKONGPEPE.addLiquidity(uint256,uint256) (HONGKONGPEPE.sol#521-534) sends eth to arbitrary user
  Dangerous calls:
  - uniswapV2Router.addLiquidityETH(value: ethAmount)(address(this),tokenAmount,0,0,address(owner()),block.timestamp) (HONGKONGPEPE.sol#526-533)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Reentrancy in HONGKONGPEPE._transfer(address,address,uint256) (HONGKONGPEPE.sol#392-464):
  External calls:
  - swapBack() (HONGKONGPEPE.sol#427)
    - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(thi
```

Fig. 4. Vulnerability detection in smart contracts using Slither

Among the 500 smart contracts analyzed, 80 were found to contain vulnerabilities.

Table 16. Vulnerability distribution in the dataset

Vulnerability Type	Number of Contracts
Access Control Vulnerabilities	37
Reentrancy Attack Vulnerabilities	58

As Table 16 We divided the dataset into 70% for training and 30% for testing. Based on Slither’s detection results, for each smart contract, we labeled it as 1 if a particular vulnerability existed, and 0 otherwise.

For each vulnerability type, we calculate detection performance metrics derived from confusion matrix analysis. The overall detection effectiveness is subsequently determined by averaging these metrics across all vulnerability categories. The evaluation system employs four key indicators: average accuracy (acc), average precision (pre), average recall (rec), and average F1-score (F1), formally defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (4)$$

where Accuracy measures the proportion of correct predictions among all predictions made, reflecting the model’s overall detection capability.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (5)$$

Precision quantifies the ratio of correctly identified vulnerabilities to all positive predictions, indicating the model’s prediction reliability.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (6)$$

Recall represents the fraction of actual vulnerabilities successfully detected, measuring the model’s capability to identify all relevant cases.

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (7)$$

The F1-score provides a harmonic mean of precision and recall, offering a balanced assessment of model performance.

In these formulations: **TP (True Positive)**: Contracts containing specific vulnerabilities that are correctly identified. **TN (True Negative)**: Vulnerability-free contracts properly recognized as secure. **FP (False Positive)**: Secure contracts erroneously flagged as vulnerable. **FN (False Negative)**: Vulnerable contracts that remain undetected by the system.

4.3 Experimental Results and Analysis

In this experiment, we employed two feature extraction methods:

- **Method 1**: Vectorization based on smart contract opcode text.
- **Method 2**: Using 2-Gram algorithm to split opcode sequences into bigram features, then calculating TF-IDF values for each dimension as vector input.

We trained five models using these methods: SVM, Decision Tree, Random Forest, KNN, and Logistic Regression.

Due to the limited size of the smart contract opcode dataset, the access control vulnerability could not produce results in any of the five models. While reentrancy attack vulnerabilities yielded results, the detection performance across models was not significantly distinguishable, with substantial observed variance.

As shown in Table 17, the small dataset size led to identical training results across all five models when using Method 1.

Table 17. Training results of different models using Method 1

Machine Learning Model	Accuracy	Precision	Recall	F1-score
Support Vector Machine	0.583	0.583	1.0	0.737
Decision Tree	0.583	0.583	1.0	0.737
Random Forest	0.583	0.583	1.0	0.737
K-Nearest Neighbors	0.583	0.583	1.0	0.737
Logistic Regression	0.583	0.583	1.0	0.737

Table 18 presents the results using Method 2, where the Decision Tree model demonstrates comparatively better performance.

Figures 5 visually compares the performance of Decision Tree and Random Forest models under both feature extraction methods.

Table 18. Training results of different models using Method 2

Machine Learning Model	Accuracy	Precision	Recall	F1-score
Support Vector Machine	0.583	0.583	1.0	0.737
Decision Tree	0.667	0.65	0.929	0.765
Random Forest	0.542	0.565	0.929	0.703
K-Nearest Neighbors	0.583	0.583	1.0	0.737
Logistic Regression	0.583	0.583	1.0	0.737

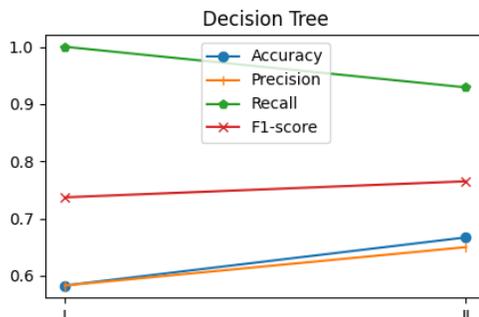


Fig. 5. Comparison of model training results between two methods

Key observations from the results:

- Most models showed identical performance with both methods.
- Decision Tree using Method 2 achieved better Accuracy and F1-score than Method 1.
- Random Forest with Method 1 demonstrated superior performance across all metrics compared to Method 2.

The imbalanced dataset significantly impacts training outcomes, leading to substantial variance in results. Method 2's bigram-based feature extraction showed particular promise with Decision Tree models, suggesting this approach may better capture relevant patterns when sufficient training data is available.

5 Classifier Chain-Based Malicious Code Detection in Smart Contracts

In practical applications, many scenarios require handling multi-label problems, such as detecting multiple types of malicious code in smart contracts. Decomposing multi-label problems into multiple independent binary classification problems may indeed ignore potential relationships between labels, thereby reducing the classifier's detection capability. The classifier chain model is an effective method for solving multi-label learning problems, which fully considers the correlations between labels while maintaining acceptable computational complexity. By linking binary classifiers in a directed structure, the classifier chain makes individual label predictions serve as features for other classifiers. This approach can capture dependencies between labels and improve the classifier's detection capability. Applying the classifier chain model to

smart contract malicious code detection is a reasonable choice. In practice, we collect a multi-labeled smart contract dataset and then train it using the classifier chain model. The trained model will be able to simultaneously identify multiple types of vulnerabilities in a single contract, thereby improving detection accuracy and efficiency. The overall process of malicious code detection is shown in the following figure 6:

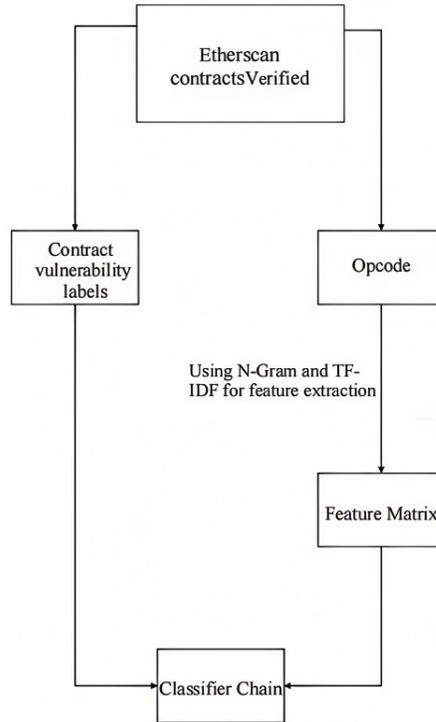


Fig. 6. Overall flowchart of malicious code detection

5.1 Principles of Classifier Chains

The classifier chain is a method for handling multi-label learning problems. Multi-label learning refers to cases where each sample in a dataset may have multiple related labels. The goal of classifier chains is to capture correlations between labels to improve the predictive performance of classifiers. In multi-label learning, classifier chains connect multiple binary classifiers in a directed structure, making individual label predictions serve as features for other classifiers. The working principle of classifier chains is as follows:

- (1) For a given multi-label dataset, first determine an order of labels, such as sorting them by frequency of occurrence.

- (2) For each label, train an independent binary classifier that predicts whether the current label exists. For the first label, the binary classifier uses only the original features as input. For subsequent labels, the binary classifier uses both the original features and the predictions of all preceding labels as input.
- (3) For predicting a new sample, first use the first binary classifier to predict the first label. Then, combine the prediction result with the original features and pass them as input to the second binary classifier. This process continues until all labels are predicted.

5.2 Experimental Results and Analysis

This paper employs the "ClassifierChain" class with Random Forest classifiers (RandomForestClassifier) as base classifiers. When comparing with other supervised algorithm models, we continue to use the confusion matrix. Since the classifier chain handles multi-label learning problems, the vulnerability labels here include reentrancy attacks and access control vulnerabilities. The feature vectors are processed using a smart contract opcode vectorization method based on contract text. Four metrics—Accuracy, Precision, Recall, and F1-Score—are used to evaluate the effectiveness of smart contract malicious code detection.

The training results of the classifier chain are shown in Table 19:

Table 19. Training Results

Model	Accuracy	Precision	Recall	F1-Score
Classifier Chain	0.333	0.272	0.467	0.343

From the training results, we can observe that the performance of the classifier chain model is suboptimal. The accuracy, precision, recall, and F1-Score are all relatively low. This outcome is likely attributed to imbalanced dataset collection.

6 Related Work

In machine learning-based malicious code detection, feature extraction and representation are crucial [30]. Researchers attempt to extract various features from smart contract code, including syntactic features, semantic features, control flow, and data flow [6]. These features can be represented using techniques such as the bag-of-words model, word vector embeddings, and graph representation learning [3]. Recent studies further demonstrate the importance of transaction pattern analysis, as seen in [21], which detects malicious Web3 accounts through transaction graph analysis. Effective feature extraction and representation methods help improve the detection performance of machine learning models [26].

In a 2019 study by Wang et al. [34], ASTs were extracted from Solidity source code as an intermediate representation of programs for smart contract vulnerability detection. Subsequent work by [32] introduced LLM-based approaches for automated smart contract summarization, enhancing AST utilization through control flow prompts. They experimented with four common binary classifiers, including Support Vector Machines (SVM), Neural Networks (NNs), Random Forests (RF), and Decision Trees (DT), to train models.

In this study, they built and evaluated a total of 184 machine learning models to predict 16 types of vulnerabilities. By comparing the performance of these models, researchers could identify the most suitable methods for smart contract vulnerability detection [5]. Such research is highly significant for improving the security and reliability of smart contracts [44].

In 2021, Xu et al. [47] extracted corresponding abstract syntax trees from the source code of each vulnerability dataset, as well as from a large dataset. They then extracted feature vectors based on subnodes shared by the two ASTs. Finally, they used the K-Nearest Neighbor (KNN) algorithm and Stochastic Gradient Descent (SGD) to train models. Both methods leveraged abstract syntax trees as the basis for feature extraction, avoiding the need to set up complex patterns or execute them on Ethereum. Recent advancements like [27] and [20] further extend this paradigm by detecting state defects in decentralized exchanges and analyzing cross-contract execution patterns. This makes these methods relatively simple for professionals to use. However, a limitation of these approaches is that obtaining source code is more difficult compared to other contract attributes, reducing their practical utility for third-party detection. Alternative solutions like [23] and [31] address this challenge by characterizing account behaviors and detecting malicious activities through gas usage patterns.

The classifier chain method has become a popular approach for solving multi-label learning problems. By linking binary classifiers in a directed structure, individual label predictions serve as features for other classifiers [22]. This method is flexible and effective, achieving state-of-the-art experimental performance across multiple datasets and multi-label evaluation metrics [24]. In recent years, many studies have explored the theoretical foundations of classifier chains and continued to improve them.

The Bayesian classifier chain algorithm proposed by Zaragoza et al. [49] combines Bayesian networks with classifier chains. By inferring a Tree Augmented Network (TAN) from the training dataset, the correlations between labels are represented in a tree structure. Then, a node is randomly selected from the TAN as the root, and different paths from this root are used to form different chains.

Hernandez et al. [14] proposed a hybrid binary-chain multi-label classifier that defines different chains based on the correlation coefficients calculated between each pair of labels. Each chain consists of labels identified as strongly positively correlated. Kumar et al. [18] adopted a kernel-target alignment technique with tunable input parameter beam width to determine suitable chain orders. By performing beam search on a tree, where each distinct path represents a different label permutation, they addressed the label ordering problem.

Research on machine learning-based malicious code detection in smart contracts has made progress but still faces many challenges and problems. The integration of AI and blockchain for privacy preservation [25], the detection of emerging threats like NFT wash trading [36], and the need to characterize complex ecosystems like Solana NFTs [17] highlight the evolving nature of this field. Additionally, emerging tools like [45] statistically quantify external data dependencies in real-world contracts, while [28] demonstrates how LLMs can automate bad practice detection, pointing to future research directions.

7 Conclusion and Future Work

7.1 Conclusion

Smart contracts containing malicious code may lead to significant security issues and asset losses, necessitating effective detection and prevention mechanisms. Consequently, machine learning-based malicious code detection techniques for smart contracts have garnered widespread application. The main contributions of this work include:

- (1) This paper provides a brief introduction to machine learning-based malicious code detection in smart contracts, with a focus on an opcode vectorization method based on contract text.
- (2) It discusses five machine learning models and a classifier chain-based approach for malicious code detection.

The experimental data reveals significant issues with the collected dataset, leading to substantial errors in training results. However, in the realm of smart contracts, malicious contracts are relatively rare and difficult to collect. This results in severe dataset imbalance, with an abundance of normal contracts and very few malicious ones. Such imbalance may introduce bias during model training, thereby affecting detection performance. Additionally, dataset quality is another critical issue, including label errors and duplicate samples, which may further degrade model performance.

7.2 Future Work

This study utilizes an opcode-based feature vectorization method, but due to the very limited smart contract opcode dataset available, the effectiveness of this feature extraction method remains unverified. Future work will focus on constructing a large and representative dataset of smart contract samples.

- (1) Developing web crawlers to extract deployed contract code from Ethereum Etherscan could save considerable time. Current smart contract vulnerability detection tools cannot perform cross-version detection. For example, if a contract is written in Solidity versions 0.6.0 and 0.7.0, it becomes unsuitable as a sample, leading to significant time waste during manual screening.
- (2) To address the lack of malicious contract samples in existing datasets, semi-supervised and unsupervised learning methods should be explored to leverage large quantities of unlabeled contract samples for training.
- (3) Privacy protection and security challenges must be addressed to ensure that machine learning-based malicious code detection systems do not leak sensitive information or become vulnerable to adversarial attacks.

References

- [1] Nemitari Ajenka, Peter Vangorp, and Andrea Capiluppi. 2020. An Empirical Analysis of Source Code Metrics and Smart Contract Resource Consumption. *Journal of Software: Evolution and Process* 32, 10 (2020), 1–22.

- [2] Fahad Al Debeyan, Tracy Hall, and David Bowes. 2022. Improving the Performance of Code Vulnerability Prediction Using Abstract Syntax Tree Information. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 2–11.
- [3] Deepak Suresh Asudani, Naresh Kumar Nagwani, and Pradeep Singh. 2023. Impact of Word Embedding Models on Text Analytics in Deep Learning Environment: A Review. *Artificial Intelligence Review* 56, 9 (2023), 10345–10425.
- [4] Morena Barboni, Andrea Morichetta, and Andrea Polini. 2023. Smart Contract Testing: Challenges and Opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 21–24.
- [5] Jiuyang Bu, Wenkai Li, Zongwei Li, Zeng Zhang, and Xiaoqi Li. 2025. Enhancing Smart Contract Vulnerability Detection in DApps Leveraging Fine-Tuned LLM. *arXiv preprint arXiv:2504.05006* (2025).
- [6] Jiuyang Bu, Wenkai Li, Zongwei Li, Zeng Zhang, and Xiaoqi Li. 2025. SmartBugBert: BERT-Enhanced Vulnerability Detection for Smart Contract Bytecode. *arXiv preprint arXiv:2504.05002* (2025).
- [7] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *Comput. Surveys* 53, 3 (2020), 1–43.
- [8] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162.
- [9] Jacob Curtis. 2022. On Language-Agnostic Abstract-Syntax Trees: Student Research Abstract. In *Proceedings of the 37th ACM/SI-GAPP Symposium on Applied Computing (SAC)*. 1619–1625.
- [10] R Dhaya and M Poongodi. 2014. Detecting software vulnerabilities in android using static analysis. In *Proceedings of the IEEE International Conference on Advanced Communications, Control and Computing Technologies*. 915–918.
- [11] Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Nort, and Dirk Draheim. 2021. Legally Enforceable Smart-Contract Languages: A Systematic Literature Review. *Comput. Surveys* 54, 5 (2021), 1–34.
- [12] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15.
- [13] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. 945–956.
- [14] Pablo Hernandez-Leal, Felipe Orihuela-Espina, Enrique Sucar, and Eduardo F Morales. 2012. Hybrid binary-chain multi-label classifiers. In *Proceedings of the 6th European Workshop Probabilistic Graphical Models*.
- [15] Giacomo Ibb, Giuseppe Antonio Pierro, and Marco Di Francesco. 2021. Evaluating machine-learning techniques for detecting smart ponzi schemes. In *Proceedings of the IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 34–40.
- [16] Yuntae Kim, Dohyun Pak, and Jonghyup Lee. 2019. ScanAT: identification of bytecode-only smart contracts with multiple attribute tags. *IEEE Access* 7 (2019), 98669–98683.
- [17] Dechao Kong, Xiaoqi Li, and Wenkai Li. 2024. Characterizing the Solana NFT Ecosystem. In *Companion Proceedings of the ACM on Web Conference (WWW)*. 766–769.
- [18] Abhishek Kumar, Shankar Vembu, Aditya Krishna Menon, and Charles Elkan. 2013. Beam search algorithms for multilabel learning. *Machine learning* 92 (2013), 65–89.
- [19] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. 2024. COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1358–1369.
- [20] Wenkai Li, Xiaoqi Li, Yuqing Zhang, and Zongwei Li. 2024. DeFiTail: DeFi Protocol Inspection through Cross-Contract Execution Analysis. In *Companion Proceedings of the ACM on Web Conference (WWW)*. 786–789.
- [21] Wenkai Li, Zheng Liu, Xiaoqi Li, et al. 2024. Detecting Malicious Accounts in Web3 through Transaction Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2482–2483.
- [22] Xiaoqi Li. 2021. Hybrid analysis of smart contracts and malicious behaviors in ethereum.

- [23] Xiaoqi Li, Ting Chen, Xiapu Luo, and Jiangshan Yu. 2020. Characterizing erasable accounts in ethereum. In *Proceedings of the 23rd International Conference on Information Security (ISC)*. 352–371.
- [24] Xiaoqi Li, Le Yu, and Xiapu Luo. 2017. On Discovering Vulnerabilities in Android Applications. 155–166 pages.
- [25] Zongwei Li, Dechao Kong, Yuanzheng Niu, Hongli Peng, Xiaoqi Li, and Wenkai Li. 2023. An overview of AI and blockchain integration for privacy-preserving. *arXiv preprint arXiv:2305.03928* (2023).
- [26] Zongwei Li, Wenkai Li, Xiaoqi Li, and Yuqing Zhang. 2024. Guardians of the ledger: Protecting decentralized exchanges from state derailment defects. *IEEE Transactions on Reliability* (2024).
- [27] Zongwei Li, Wenkai Li, Xiaoqi Li, and Yuqing Zhang. 2024. StateGuard: Detecting State Derailment Defects in Decentralized Exchange Smart Contract. In *Companion Proceedings of the ACM on Web Conference (WWW)*. 810–813.
- [28] Zongwei Li, Xiaoqi Li, Wenkai Li, et al. 2025. SCALM: Detecting Bad Practices in Smart Contracts Through LLMs. *arXiv:2502.04347*
- [29] Huipeng Liu, Baojiang Cui, Jie Xu, and Lihua Niu. 2024. An Efficient Cross-Contract Vulnerability Detection Model Integrating Machine Learning and Fuzz Testing. In *Proceedings of the International Conference on Emerging Internet, Data & Web Technologies*. 297–306.
- [30] Zekai Liu and Xiaoqi Li. 2025. SoK: Security Analysis of Blockchain-based Cryptocurrency. *arXiv preprint arXiv:2503.22156* (2025).
- [31] Zekai Liu, Xiaoqi Li, Hongli Peng, and Wenkai Li. 2024. GasTrace: Detecting Sandwich Attack Malicious Accounts in Ethereum. In *2024 IEEE International Conference on Web Services (ICWS)*. 1409–1411.
- [32] Yingjie Mao, Xiaoqi Li, Wenkai Li, Xin Wang, and Lei Xie. 2024. SCLA: Automated Smart Contract Summarization via LLMs and Control Flow Prompt. *arXiv preprint arXiv:2402.04863* (2024).
- [33] Cade Metz. 2016. The biggest crowdfunding project ever—the DAO—is kind of a mess.
- [34] Pouyan Momeni, Yu Wang, and Reza Samavi. 2019. Machine learning model for smart contracts security analysis. In *Proceedings of the 17th international conference on privacy, security and trust (PST)*. 1–6.
- [35] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.
- [36] Yuanzheng Niu, Xiaoqi Li, Hongli Peng, and Wenkai Li. 2024. Unveiling Wash Trading in Popular NFT Markets. In *Companion Proceedings of the ACM on Web Conference (WWW)*. 730–733.
- [37] Ahmad Sghaier Omar and Otman Basir. 2018. Identity management in IoT networks using blockchain and smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 994–1000.
- [38] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *Ieee Access* 8 (2020), 24416–24427.
- [39] Tanusree Sharma, Zhixuan Zhou, Andrew Miller, and Yang Wang. 2023. A Mixed-Methods Study of Security Practices of Smart Contract Developers. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. 2545–2562.
- [40] Wei Chuan Tan Matai. [n. d.]. Blockchain Explorer. Available: <https://cn.etherscan.io/>.
- [41] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632* (2018).
- [42] Franklin Tchakounté, Koudanbe Amadou Calvin, Ado Adamou Abba Ari, and David Jaures Fotsa Mbogne. 2022. A Smart Contract Logic to Reduce Hoax Propagation across Social Media. *Journal of King Saud University - Computer and Information Sciences* 34, 6 (2022), 3070–3078.
- [43] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *Comput. Surveys* 54, 7 (2021), 1–38.
- [44] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 390–400.

- [45] Yishun Wang, Xiaoqi Li, Shipeng Ye, Lei Xie, and Ju Xing. 2024. Smart Contracts in the Real World: A Statistical Exploration of External Data Dependencies. arXiv:2406.13253
- [46] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
- [47] Yingjie Xu, Gengran Hu, Lin You, and Chengtang Cao. 2021. A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks* 2021, 1 (2021), 5798033.
- [48] Donghan Yu, Yiming Yang, Ruohong Zhang, and Yuexin Wu. 2021. Knowledge Embedding Based Graph Convolutional Network. In *Proceedings of the Web Conference (WWW)*. 1619–1628.
- [49] Julio H Zaragoza, Luis Enrique Sucar, Eduardo F Morales, Pedro María Larrañaga Múgica, and María Concepción Bielza Lozoya. 2011. Bayesian chain classifiers for multidimensional classification.
- [50] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection Using Graph Neural Network. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*. 3283–3290.