A Multiple-Label Vulnerabilities Detection Scheme for Smart Contract with Ensemble Learning

Mingwu Zhang, Meng Huang, Peng Yu, Gang Shen, Mengya Lei, Jia Yang

***Abstract*—Smart contracts stored on a blockchain can automatically execute when predetermined terms and conditions are met. With the abundance of smart contract applications in blockchain systems, a great number of smart contracts has exploded to automate transactions and enhance trust across supply chains, trade finance and other domains. Meanwhile, due to the complexity of blockchain protocols, it is difficult to avoid human mistakes in the developing and coding of smart contracts, and the emergence of several serious security incidents has intensified the concerns about blockchain security. Therefore, detecting the vulnerabilities in smart contract has become a crucial issue for blockchain and smart contract security. Traditional smart contract vulnerability detection methods suffer from low code coverage, high false alarm rate, low detection efficiency, and excessive resource demand. In this paper, we propose a multi-label vulnerability detection system for smart contracts that is based on the ensemble learning, namely MLCD scheme, which can detect seven kinds of vulnerabilities in smart contracts. The vulnerability detection tool Vandal compiles the smart contract to get the control flow information, which includes the information of basic code blocks. We mainly extract the opcode features in each basic block and the edge features between all basic blocks from the control flow graph. Next, we use five machine learning algorithms and three sampling algorithms to construct the best model of MLCD scheme. On the publicly available SmartBugs Wild dataset, our MLCD scheme provides maximum prediction values for both Micro-F1 and Macro-F1 are 99.4%. Experimential results indicate that, compared with related works, the MLCD model gives a better smart contract vulnerability detection performance. We also provide the pratical coding implementation and analysis via *VulnerabilityDetnSys: https://github.com/0929hua/VulnerabilityDetnSys*.**

***Index Terms*—Vulnerability Detection, Multi-label Classification, Machine Learning, Smart Contract**

I. Introduction

A. Vulnerability in Smart Contracts

Smart contract was first introduced by Nick Szabo [1], which is stored on a blockchain and can automatically execute when predetermined terms and conditions are met. It can provide the functionalities for disseminating, verifying or enforcing by information-based approaches. Smart contracts were not used in reality for the short of believable execution environment and suitable contract writing techniques during the period. It was not until the appearance of Bitcoin and blockchain technology that the above problems were solved [2]. However, the incomplete nature of blockchain Turing prevents it from providing complex services [3], [4].

The first white paper on Ethereum was published by Vitalik Buterin that was developed Ethereum and introduced smart contracts to the blockchain [5]. With the continuous development of Ethereum, the applications of smart contracts become explosive growth. Ethereum provides the complete programming language (Solidity) and believable environment for smart contracts, which enables developers to deploy smart contract-based applications (dapp) on the blockchain. Due to its characteristics of decentralization, verifiability, and certainty. Smart contracts can be used in a wide range of scenarios including digital identity [6], securities [7], [8], financial trading [9], and IoT [10], etc.

With the abundance of smart contract applications, the number of smart contracts has exploded. Meanwhile, due to the complexity of blockchain protocols, it is difficult to avoid human mistakes in the developing and coding of smart contracts, and the emergence of several serious security incidents has intensified the concerns about blockchain security. For example, in June 2016, US hackers made use of the DAO [11] contract's reentrant vulnerability to steal about 60 million ether coins; since the Delegatecall breach in Parity's multi-signature wallet [12], nearly 300 million worth of Ether was frozen in July 2017. The attacker launched a continuous random number attack on EOS.WIN, earning more than 20,000 EOS digital currency; in 2022, Beanstalk Farm was hit by a flash loan attack that cost close to 2182 million.

In Ethereum system, smart contracts are run as digital protocol, and could not be modified after effectly deployed. In terms of blockchain security, the detection of smart contract vulnerability has become a crucial issue. Currently, the traditional methods for detecting smart contract vulnerabilities are formal verification, symbolic execution, intermediate representation and fuzzy testing. Formal verification methods include F*framework, Securify, which uses a formal language to transform smart contracts into formal models, and verifies the security of smart contracts by mathematical logic and proofs. Symbolic execution represents approaches like Oyente [13], Mythril [14]. These tools collect code execution path constraints and find security issues by symbolizing variables in the source code, which are complex and inefficient. The intermediate Representation method has SmartCheck [15], SmartCheck relies on simple logic rules to detect vulnerabilities, which has a higher false alarm rate. Fuzzy testing

The work was supported in part by the Major Research Plan of Hubei Provience (No. 2023BAA027, 2021BEA163), National Natural Science Foundation of China (No.62202146, 62072134 and U2001205), Natural Science Foundation of Hubei Province (No. 2022CFB914, Q20293410).

M. Zhang is with the School of Computer Science, Hubei University of Technology, and also with the School of Computer Science and Information Security, Guilin University of Electronic Technology. E-mail: mzhang}@hbut.edu.cn.

M. Huang, P. Yu and J. Yang are with the School of Computers, Hubei University of Technology.

Manuscript received XX-XX, 2024; revised XX–XX, 2024.

approaches such as ContractFuzzer, which generates fuzzy test cases based on the ABI specifications of smart contracts and defines detection schemes to identify security vulnerabilities. However, it relies excessively on well-designed test cases and is inefficient.

Traditional vulnerability detection methods have problems such as low code coverage, high false alarm rate, low detection efficiency, and excessive resource demand. Compared with them, machine learning methods can alleviate the above problems. Therefore, we propose a multi-label vulnerability detection system for smart contracts called MLCD, which combines smart contract control flow graph information and machine learning techniques. Firstly, the control flow information of the smart contract is compiled by Vandal, which contains the basic code block information. We mainly extract the opcode features in each basic block and the edge features between all basic blocks from the control flow graph. Second, we use five machine learning algorithms and three sampling algorithms to build optimal models called MLCD for efficient vulnerability detection.

B. Contribution

we propose a multi-label vulnerability detection system for smart contracts that is based on the ensemble learning, namely MLCD scheme, which can detect seven kinds of vulnerabilities in smart contracts. The main contributions of this work are listed as follows:

- Using the control flow graph of the smart contract to represent the execution logic in smart contract, this paper extracts the opcode features in each basic block and the edge features between all basic blocks from the control flow graph, and use PCA to reduce the dimensionality of the opcode features and edge features.
- We propose a smart contract vulnerability detection system using ensemble learning technique, the system's url is published at Github repositories [1]. We use five machine learning algorithms and three sampling algorithms to build the MLCD scheme. We also employ the machine learning algorithms such as LightGBM, RF, XGBoost, AdaBoost and SVM. Sampling algorithms include SMOTE, SMOTETomek, and SMOTENN etc.
- There are seven vulnerabilities in smart contracts that can be effectively and rapidly detected by MLCD, namely vulnerabilities include access control, arithmetic overflow, denial service, tod, reentrancy, time manipulation, and unchecked low calls vulnerabilities. Running on the Smartbugs Wild dataset, the predicted Micro-F1 and Macro-F1 values by this approach both exceed 99.4%.

C. Paper Organization

The remaining parts of this paper are arranged as below. Section II presents a summary of related work. Section III mainly focuses on Ethereum, the bytecode, opcode and compilation process of smart contracts. Section IV introduces seven kinds of vulnerabilities in smart contracts. Section V provides a description of our new vulnerability detection method. The experimental results are presented in Section VI. Section VII includes discussion and conclusion.

[1]VulnerabilityDetnSys: https://github.com/0929hua/VulnerabilityDetnSys

II. Related Work

In the past few years, the methods of smart contract vulnerability detection primarily contain five types, namely formal verification, symbol execution, intermediate representation, deep learning and fuzzy testing [16]. These methods can analyse the security of the source code, bytecode or opcode of smart contracts to detect vulnerabilities.

Formal Verification. The formal verification transforms code into a formal model, and uses strict maths theorem proving and sophisticated mechanisms for vulnerability validation. ZEUS [17], and F*framework [18] hardly support vulnerability detection at the EVM execution layer. KEVM [19] proposes formal validation methods, but is not appropriate for actual smart contract vulnerability detection. As formal verification relies on strict mathematical reasoning and validation, it is unable to perform dynamic analysis to detect and evaluate execution paths in the contract, resulting in a high rate of false positives and misses.

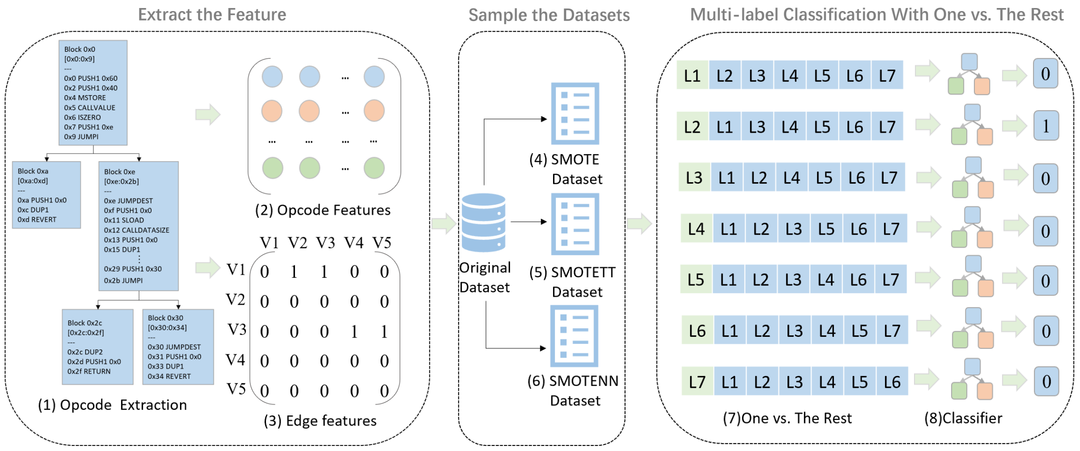
Symbol Execution. Symbolic execution mainly collects path constraints and finds security issues by symbolizing variables in the source code, then interpreting the instructions in the execution procedure line upon line and updating the state of execution. The current representative tools are Oyente [13], Mythril [14], but they cannot completely solve the problem of state space explosion and exponential growth of execution paths.

Intermediate Representation. The intermediate representation approach converts the source code or bytecode of the contracts into high-level semantic representations, and analyzes the representations to find security issues. Currently, smart contract tools that utilize intermediate representations include SmartCheck [15], and ContractGuard [20]. SmartCheck and ContractGuard support more vulnerability detection, but SmartCheck relies on simple logic rules to detect vulnerabilities, which has a higher false alarm rate.

Fuzzy Testing. Fuzzy testing produces extensive normal and abnormal inputs from the target application on the basis of specific rules, gives these inputs to the application and monitors whether the program contains abnormal state to identify security issues. Fuzzy test methods include ContractFuzzer [21], Reguard [22] and ILF [23], However, fuzzy testing relies excessively on well-designed test cases and is inefficient.

Deep Learning. In deep learning, the graph of the contract is built by capturing the essential semantics and control flow-related messages in the source code, which is combined with the graph neural network for vulnerability detection. Deep learning methods such as ReChecker [24], DR-GCN [25] and multimodal artifical intellgent framework [26] are available, but building graph structure information can be incomplete and take a long time. Recently, Zhai et al. [27] proposed a vulnerability detection approach using gated graph neural network to identify the vulnerability in smart contract.

By analyzing the traditional vulnerability detection methods, we find these methods have problems, like low code coverage,


```
Block 0x0
[0x0:0x9]
---
0x0 PUSH1 0x60
0x2 PUSH1 0x40
0x4 MSTORE
0x5 CALLVALUE
0x6 ISZERO
0x7 PUSH1 0xe
0x9 JUMPI
```
```
Block 0xa
[0xa:0xd]
---
0xa PUSH1 0x0
0xc DUP1
0xd REVERT
```
```
Block 0xe
[0xe:0x2b]
---
0xe JUMPDEST
0xf PUSH1 0x0
0x11 SLOAD
0x12 CALLDATASIZE
0x13 PUSH1 0x0
0x15 DUP1
       ⋮
0x29 PUSH1 0x30
0x2b JUMPI
```
```
Block 0x2c
[0x2c:0x2f]
---
0x2c DUP2
0x2d PUSH1 0x0
0x2f RETURN
```
```
Block 0x30
[0x30:0x34]
---
0x30 JUMPDEST
0x31 PUSH1 0x0
0x33 DUP1
0x34 REVERT
```

$$\begin{array}{c|ccccc} & V1 & V2 & V3 & V4 & V5 \\ \hline V1 & 0 & 1 & 1 & 0 & 0 \\ V2 & 0 & 0 & 0 & 0 & 0 \\ V3 & 0 & 0 & 0 & 1 & 1 \\ V4 & 0 & 0 & 0 & 0 & 0 \\ V5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Fig. 1: *Ensemble Training Framework of Our Vulnerability Detection Scheme*

high false alarm rate, low detection efficiency, and excessive resource demand. Compared with them, machine learning methods can alleviate the above problems. In this work, we present a multi-label vulnerability detection system for smart contracts, namely MLCD scheme, which combines smart contract control flow graph information and machine learning techniques.

III. Preminiaries

We provide a brief introduction to Ethereum, bytecode, opcode and compilation process of smart contracts, and give relationship between them. Actually, smart contracts written by Solidity are only discussed in Ethereum blockchain system.

A. *Ethereum Technique*

As a programmable and Turing-complete public blockchain platform, Ethereum [5] has strong adaptability and flexibility. Smart contracts are stored in the blockchain network with a particular binary format, which is called EVM bytecode. EVM uses a series of instructions called opcodes to perform different tasks. There are 140 opcodes in EVM that can execute different processes.

In fact, Ethereum is a decentralized public ledger for verifying and recording transactions. In the meanwhile, it needs to define fault-tolerant protocols in the default unreliable asynchronous network, which implements consistency and correctness of data on various ledger nodes. The consensus protocol called Proof of Work (PoW) [28] is used by Ethereum to solve the above problem. This consensus mechanism allows Ethereum nodes to reach agreement on the status of all information documented on the blockchain and avoids a number of attacks with economic impact.

B. *Bytecode, Opcode and Compilation Process of Smart Contracts*

The bytecode is decomposed into a series of opcodes and operands on Ethereum. The opcode is some predefined operation instructions, which can be executed after being recognized by EVM. Currently, the opcodes in Ethereum smart contracts are mainly divided into 11 classes, and each type of instruction has a different value range number, totaling 140 instructions. These include comparison and bitwise logical operations, stop and arithmetic operations, SHA3 (KECCAK256) operation, environment information operation, stack operation, memory operation, storage and stream operation, copy operation, replace operation, log operation, system operation. For example, the ADD operation code is 0x01, which means that two elements are removed from the stack and the result is pushed onto the stack.

Smart contracts are typically written in the Solidity language, while other writing languages include Vyper and Bamboo. Taking the Solidity as an example, the compilation process of smart contracts is divided into three steps:

1) The contract is written by Solidity language, which is saved as a *sol* file. This file contains functions, variables and other elements of the contract.
2) The *sol* file is compiled into bytecode by the Solidity compiler. Bytecode is a representation of efficiently stored machine-readable opcodes, which can be executed on the EVM. The opcode is a low-level instruction, which can be interpreted and implemented directly with the EVM.
3) When the contract is compiled into bytecode, it can be deployed to Ethereum. This usually includes sending transactions with contract bytecodes to the transaction pool. The transaction is finally merged into the Ethereum block, and then the contract is created.

IV. Seven Kinds of Vulnerabilities in Smart Contracts

At present, there are many kinds of vulnerabilities in smart contracts, which are tightly associated with the design logic of the contract, the developer's writing, and the blockchain. The smart contracts run as digital protocol, and can't be modified after deployment. To implement vulnerability detection for smart contracts before deployment, this paper investigates the below categories of vulnerabilities.

Access Control Vulnerability. The root cause of the access control vulnerability is that developers have not clearly defined the access rights to functions in smart contracts. Functions and variables in smart contracts have four types of access restrictions: public, private, external, and internal. If a function that should have been defined as a private type is not authorised, the function will be defined as a public type by default and ordinary users will not be entitled to access the function. This will allow an attacker to gain unauthorised access to a variable or function, leading to the serious consequences of an "injection" type attack.

Arithmetic overflow Vulnerability. In 2018, due to the arithmetic overflow vulnerability [29] in the USD Chain BEC contract, the value of its tokens went to zero instantly. Arithmetic overflows in smart contracts include addition overflows, subtraction overflows and multiplication overflows. Ethereum specifies fixed size and unsigned integer types for integers and Solidity supports uint8 to uint256. For variables of type uint8, Uint8 can store numbers in the range [0,255] and storing 256 in uint8 will result in the error of integer overflow.

Denial Service Vulnerability. The attacker consumes the resources (such as Ether and Gas) in the Ethereum network by destroying the original logic in the contract, so that the contract can not be executed normally or provide normal services for a period of time. There are usually three types of attacks, namely, DoS attack by (unexpected) Revert, DoS attack by Gas limit of Ether block, and DoS attack by contract owner account.

TOD Vulnerability. TOD is a vulnerability in the blockchain system layer. Since the blockchain competes for the book-keeping right through consensus algorithms such as PoW or PoS [28], the transaction packing sequence is decided by the nodes, which are competing for the book-keeping right. Different nodes may pack transactions in different orders, so different transaction orders often lead to different results.

Reentrancy Vulnerability. Reentrancy is a vulnerability in the code layer of smart contracts, which usually involves transfer operations. Since Solidity smart contracts have a special fallback mechanism, the fallback function of the receiving contract is called when the Ether transfer operation occurs. If the malicious code is inserted in the fallback function by the attacker, the Ether transfer function may be called recursively to steal Ether. The reentrant vulnerability is created by this mechanism.

Time Manipulation Vulnerability. Block timestamps are available for operation by miners to get a certain range of numerical range, and smart contracts take use of the block timestamps confirmed by miners to achieve time constraints. The contract can retrieve the block timestamp, and all transactions in the block have the same timestamp, ensuring the consistent execution status of the contract. Nevertheless, within an error range of about 900 seconds, the miner may adjust the value of the timestamp so that the attacker can use this vulnerability to generate a timestamp for the attack.

Unchecked Low Calls Vulnerability. The vulnerability is likely to occur in smart contracts related to Ether transactions. In Solidity smart contract, the four types of *call, send, callcode*, and *delegatecall* functions do not throw exceptions when the call fails, and return False to indicate the execution status. If return values of these four types of functions are not checked, this can results in continued execution of the code, generating a loss of Ether.

V. Detection Models

To solve the above-mentioned problems in traditional detection methods: low code coverage, high false alarm rate, low detection efficiency and excessive resource demand, this paper presents a multi-label vulnerability detection system for smart contracts called MLCD, which can automatically identify vulnerabilities in smart contracts. As can be seen from Fig. 1, MLCD scheme is composed of three parts: extract the feature, sample the datasets, and multi-label classification with OvR.

1) **Extract the feature**: It uses Vandal [30] to decompile the smart contract bytecode into the control flow information, which contains information the basic code blocks. We mainly extract the opcode features and edge features in each basic block from the control flow graph, convert the opcodes in each basic block into a word vector matrix, and convert the information of the edges between all the basic blocks into an adjacency matrix, and compose them into a feature matrix F.
2) **Sample the datasets:** We use three sampling algorithms to balance the original dataset, namely oversampling SMOTE, undersampling technology SMOTETomek and SMOTENN.
3) **Multi-label classification with OvR:** This work employs One vs Rest (OvR) algorithms and five machine learning algorithms for multi-label vulnerability classification. Machine learning algorithms include Light Gradient Boosting Machine (LightGBM) [31], Random Forest (RF) [32], Extreme Gradient Boosting (XGBoost) [33], Adaptive Boosting (AdaBoost) [34] and Support Vector Boosting (SVB) [35]. Vulnerabilities include access control, arithmetic overflow, denial service, tod, reentrancy, time manipulation,and unchecked low calls vulnerabilities.

A. Feature Extraction

Feature extract procedure is divided into three steps, which mainly includes *opcode features*, *edge features* and *dimension reduction*.

1) Opcode Features: The control flow information of the opcode consists of multiple basic block opcodes, the features of each basic block opcode are extracted and are combined to form a feature. Each basic block opcode generates a word vector by word2vec and all the values in the word vector are added together to obtain a feature, which represents the opcode feature of that basic block.

$$f_i = [f_{i_1}, f_{i_2}, f_{i_3}, ..., f_{i_j}, ..., f_{i_n}] \tag{1}$$

In equation 1, f_i indicates the opcode feature of the ith smart contract, n indicates the total number of blocks of i th smart contract. f_{i_j} denotes the opcode feature of the jth block in the ith smart contract.

2) Edge Features: The predecessor nodes' lists of each block node are extracted and converted to an adjacency matrix, and then the eigenvectors of the matrix are derived. For the feature vector corresponding to a block node, all the values in the vector are summed to get a feature, which represents the edge feature of that block node.

$$e_i = [e_{i_1}, e_{i_2}, e_{i_3}, ..., e_{i_j}, ..., e_{i_n}] \tag{2}$$

In equation 2, e_i indicates the edge feature of the ith smart contract, n indicates the total number of block nodes of i th smart contract. e_{i_j} denotes the edge feature of the jth block node in the ith smart contract.

3) Dimension Reduction: For the same class of vulnerabilities, PCA is used to perform dimensionality reduction on the opcode features and edge features. We use the length of the longest feature as a criterion and fill the feature with zeros if the width of other features is less than that criterion.

$$F = [PCA(f_i), PCA(e_i)] \tag{3}$$

In equation 3, F denotes the feature matrix of the dataset, $PCA(f_i)$ consists of N one-dimensional 3-tuple feature vectors, $PCA(e_i)$ consists of N one-dimensional 1-tuple feature vector, N denotes the dimension of F.

B. Sample the Datasets

The dataset used in this paper is the Smartbugs Wild dataset [36], which contains 47,518 smart contracts in the Ether blockchain, and based on the results of 11 integrated detection tools. Obviously, these data are authoritative and reliable. As shown in Table I, this dataset includes contracts that contain 7 vulnerabilities.

The number of arithmetic overflow vulnerability is the highest, and the number of Time Manipulation vulnerability is the lowest in our original training set. If the algorithm is trained to classify the original dataset, the training results may be poor. This is because the training randomly selects up to 80% of samples from the dataset. For example, if the number of samples labeled as arithmetic overflow vulnerabilities is large, the results of the evaluation method will be more favorable for the arithmetic overflow vulnerability. To reduce the impact of category imbalance, we adapt three sampling algorithms to solve the problem, namely oversampling SMOTE, combined sampling SMOTETomek and SMOTENN.

TABLE I: Origin Dataset

Vulnerability	This Type	The Rest	Total
Access Control (L1)	114	7021	7135
Arithmetic (L2)	6040	1095	7135
Denial Service (L3)	500	6635	7135
Front Running (L4)	139	6996	7135
Reentrancy (L5)	129	7006	7135
Time Manipulation (L6)	36	7099	7135
Unchecked Low Calls (L7)	177	6858	7135

1) SMOTE is an oversampling technique for synthesizing a few classes. It uses the k-nearest neighbor algorithm to create synthetic data to balance the dataset. However, the few class samples generated easily overlap with the majority of the surrounding class samples, which are hard to classify.
2) SMOTETomek is a combined sampling technology of SMOTE and Tomek.
3) SMOTENN is a combined sampling technology of SMOTE and KNN. Both Tomek and KNN are under-sampling technologies, which can remove overlapping samples. Tomek technique removes Tomek Links to clean data. KNN cleans the data by prediction, and rejects the sample if the prediction does not match the actual category label.

All above three sampling algorithms support multiple-label resampling. Table II shows the quantity of original dataset sampled by three sampling algorithms SMOTE, SMOTE-Tomek and SMOTETT. By SMOTE, SMOTETomek, and SMOTENN, it is obvious from Table II that the six vulnerabilities are comparatively average in number, and close to the ratio of 1:1.

TABLE II: Number of Origin Dataset After Sampling

Vulnerability	Origin	SMOTE	SMOTETT	SMOTENN
Access Control (L1)	114	6040	6039	5936
Arithmetic (L2)	6040	6040	6030	5675
Denial Service (L3)	500	6040	6034	5883
Front Running (L4)	139	6040	6037	5887
Reentrancy (L5)	129	6040	5619	5000
Time Manipulation (L6)	36	6040	5624	4216
Unchecked Low Calls (L7)	177	6040	6039	5993

C. Multi-label Classification With One vs. The Rest (OvR)

1) One vs. The Rest: For the access control, arithmetic overflow, denial service, tod, reentrancy, time manipulation, and unchecked low calls vulnerabilities, we use One-Vs-Rest and five algorithms for multi-classification, which mainly turns a seven-classification problem into seven binary classification problems. The method consists of the following steps:

1) Selecting one category as the positive category and making all other categories as the negative category;
2) Training one binary classifier for every binary classification task, and eventually training seven binary classifiers;
3) Collecting the outcomes of the seven binary classifiers to give the terminal result of multi-label classification.

2) Classification Algorithms: In binary classification tasks, the ensemble learning algorithm produces better prediction performance than a single learner, as it accomplishes the learning task by combining multiple base learners. For comparison, a single algorithm has been used.

There have two kinds of ensemble learning algorithms: *Bagging* and *Boosting*. Bagging is a kind of parallel method in which classifiers are trained independently. It improves the independence of each base model by randomly constructing training samples and randomly selecting features. Boosting is a type of sequential method in which training relies on the previous model. It trains different base models in a certain order, and each model is trained specifically for the errors of the previous model, which improves the model's accuracy. With regard to variance and bias, the aim of bagging is to reduce variance, which can be greatly reduced as each learner is independent. Boosting aims to reduce bias by continuously minimizing the loss function, where the bias is naturally reduced over time.

For the training set of one-dimensional 3-quadratic feature vector and labeled training set, we use four ensemble learning algorithms, namely LightGBM, RF, XGBoost, AdaBoost and a simple supervised classification algorithm, SVM to detect vulnerabilities in smart contracts.

- Light Gradient Boosting Machine (LightGBM): LightGBM is a new member of boosted ensemble models. Like XGBoost, it is the effective realization of Gradient Boosting Decision Tree (GBDT). The principle of this method is quite similar to GBDT and XGBoost, which fits a new decision tree with the loss function's negative gradient as a remaining approximation to the current decision tree. In comparison to XGBoost, LigthGBM has a lower memory footprint, faster training efficiency, and higher accuracy.
- Random Forest (RF): Random Forest is a particular bagging approach that consists of several decision trees. It is organized into 3 steps:
  1) N samples are randomly chosen from the sample set by the bagging approach.
  2) K features are randomly chosen from all features d (k $< d$), and the optimal segmentation features are chosen as nodes from k features to construct CART decision tree.
  3) The above-mentioned steps (1) and (3) are repeated m times to construct m-CART decision trees, which form the random forest.
- eXtreme Gradient Boosting (XGBoost): It is a typical boosting algorithm with high speed and efficiency. Compared with GBDT, a regular term is added to the objective function by XGBoost, which controls the complexity of the model and avoids overfitting. It makes the loss more accurate by using first-order and second-order derivatives, and also allows customization of the loss function. It also borrows from the practice of Random Forest and provides support for column sampling, which not only reduces the over-fitting, but also reduces the amount of calculation.
- *Adaptive Boosting (AdaBoost)*: AdaBoost is divided into 5 steps:
  1) Initializing the weight of the original dataset.
  2) Using weighted datasets to train weak learners.
  3) Calculating the weight of the weak learners on the basis of the errors.
  4) Increasing the weight of misclassified samples, and relatively decreasing the weight of correctly classified samples.
  5) Repeating steps 2-4 $K-1$ times, and combine the results of $K-1$ weak learners with weights.
- Support Vector Machine (SVM): SVM is a popular binary classification model, which can be classified as linear or non-linear. The main idea is to find a hyperplane for data segmentation, divide data into positive and negative categories, and make all data have the shortest distance to the hyperplane.

VI. Experiment

The comprehensive experiments are conducted on the test set in this section. First, the environment of the experiment and multi-label are presented. Next, we perform the comparison of sampling approaches and the comparison of classifiers to select the optimal model. The values of Micro-F1, Macro-F1 and F1-score are used for measuring the the classifier's performance. Then, we analyze the optimal model in detail and compare it with previous work. Finally, we introduce the smart contract vulnerability detection system.

A. Experimental Environment and Multi-Label

1) Experimental Environment: Since the amount of data of the experiment is very large, it requires high memory size, hard disk capacity and CPU performance of the experimental equipment. Table III describes our experimental environment. In the experiment, the dataset is separated into training set and test set in the proportion of 8:2. The results of all experiments are published on the project website **(https://github.com/0929hua/LightGBMVD)**.

TABLE III: Experiment Platform

Software and Hardware	Configuration
Sever Model	LAPTOP-MJME8I89
Memory Size	474GB
CPU	11th Gen Intel(R) Core(TM) i5-11320H
Disk Capacity	1.2TB
Operating System	Ubuntu 20.04.2 LTS

2) Multi-Label: The multi-label classification of smart contracts is defined as below:

- $X = R^d$ denotes the d-dimensional sample space, $y = \{y_1, y_2, , , y_q\}$ denotes the label space;
- For every multi-label sample (x_i, Y_i), $x_i \in X$, $Y_i \in y$, x_i denotes the d-dimensional feature vector, Y_i denotes the label set;
- The task of multi-label classification is to learn a decision function $h : X \rightarrow 2^y$ from the training set $D = \{(x_i, Y_i)|0 \leqslant i \leqslant m\}$;
- For new samples $x \in X$, the multi-label classification model predicts $h(x) \sqsubseteq y$ as the label set of the sample.

We focus on access control vulnerability, arithmetic overflow vulnerability, denial service vulnerability, tod vulnerability, reentrancy vulnerability, time manipulation vulnerability, and unchecked low calls vulnerability. Each contract has seven tags corresponding to seven types of vulnerabilities, the tags are independent of each other. For example, a contract is tagged with tag vector [0, 0, 1, 0, 0, 0, 0], indicating that it has some denial service vulnerabilities.

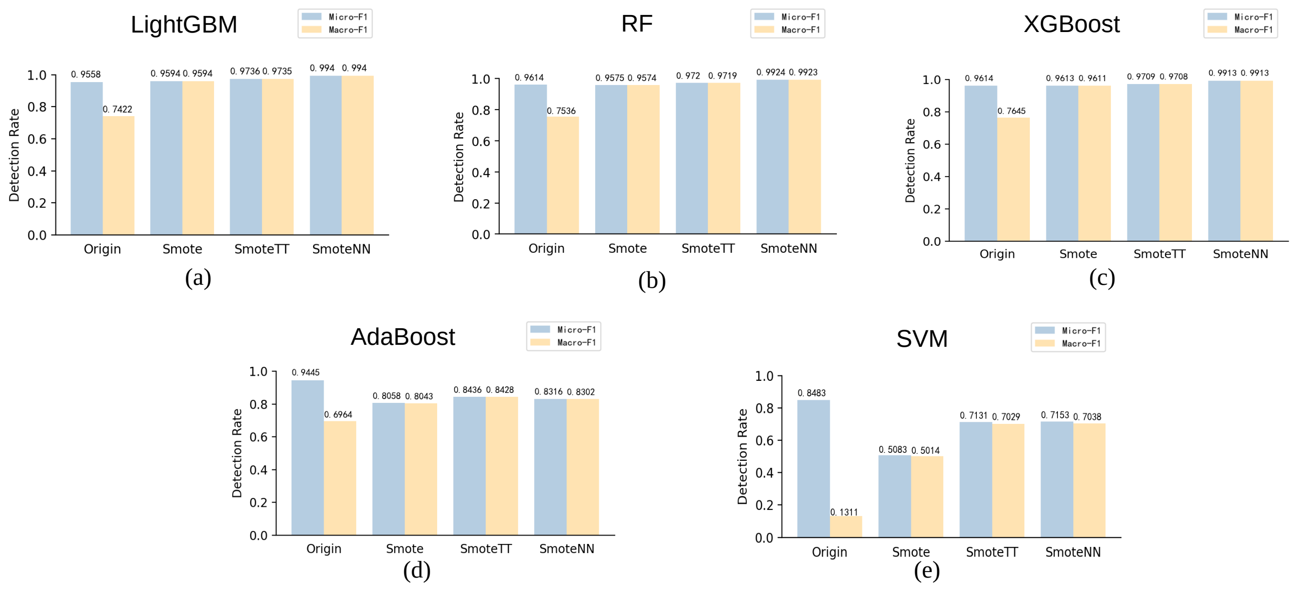


Fig. 2: *Comparison of Sampling Methods*

B. Selection of Optimal Model

1) Comparison of Sampling Approaches: We apply five classifiers for experiments, namely the LightGBM, RF, XGBoost, AdaBoost and SVM. Every classifier is trained with four different training sets, called the original training set, the training set balanced by SMOTE, the training set balanced by SMOTETomek and the training set balanced by SMOTENN. Micro-F1 and Macro-F1 are metrics for evaluating multi-label classifications. We choose Micro-F1 and Macro-F1 as the assessment metrics for classification.

In the calculation method, Micro-F1 is easily influenced by the classification results of categories with large sample sizes, while Macro-F1 assigns equal weight to every category, whatever the number of samples in every category. True positive (TP) means the number of samples correctly classified as positive. True negative (TN) means the number of correctly classified negative samples. False positive (FP) means the number of negative samples wrongly classified as positive samples. False negative (FN) means the number of positive samples wrongly classified as negative samples.

As shown in Fig. 2, for each classifier, the values of Micro-F1 and Macro-F1 based on the SMOTE, SMOTETomek, and SMOTENN balanced datasets are greater than those based on the original training set. Among the five classifiers, SMOTENN and SMOTETomek balance the dataset more effectively than SMOTE. Since SMOTENN ansd SMOTETomek have data cleaning functions than SMOTE, which can clean off useless data. More specifically, the maximum prediction values of Micro-F1 and Macro-F1 are more than 99% by LightGBM. As a result, SMOTE, SMOTETomek and SMOTENN sampling methods can reduce the effect of poor generalization ability of classifiers because of unbalanced datasets. The five classifiers based on the SMOTENN sampling dataset perform best, so the sampling method is chosen SMOTENN.

2) Comparison of Classifiers: We apply five multi-classifiers, namely LightGBM classifier, RF classifier, XGBoost classifier, AdaBoost classifier and SVM classifier, and three sampling techniques, namely SMOTE algorithm, SMOTETomek algorithm and SMOTENN algorithm to evaluate the performance. Micro-F1 and Macro-F1 are applied as assessment metrics for multiple classifications. The F1-score is the evaluation metric for binary classification and is a weighted average of Recall and Precision.

In Tables IV, V, VI, we can obviously see that the ensemble learning classifier outperforms the SVM classifier in this classification task. Moreover, all LightGBM classifiers produce higher F1 scores than RF classifiers, XGBoost classifiers, AdaBoost classifiers and SVM classifiers. In Table IV, the LightGBM classifier has the largest Micro-F1 and Macro-F1 values, both exceeding 99.4%. In Table V, the LightGBM classifier has Micro-F1 and Macro-F1 values of over 97%. In Table VI, the Micro-F1 and Macro-F1 values for the LightGBM classifier are 95%. The Micro-F1 values of the five classifiers are mostly greater than the Macro-F1 values in multi-label classification. This is caused by the large sample size for integer overflow and tod vulnerability testing, and these two categories have high F1-score. Compared with Tables V ,VI, it is obvious that LightGBM trained with the SMOTENN balanced set in Table IV performs best.

By the comparison of sampling approaches and the comparison of classifiers, we choose the LightGBM classifier and the SMOTENN sampling method to form the optimal model called MLCD.

C. Analysis of Our MLCD Scheme

1) TPR, FNR, TNR and FPR: We let TPR to denote the probability of correctly predicting positive classes among all positive classes, and let FNR to be the probability of incorrectly predicting positive classes among all positive classes, and TNR be the probability of correctly predicting negative classes among all negative classes. Also, we let FPR to denote the probability of incorrectly predicting negative classes

TABLE IV: Results of Five Classifiers based on SMOTENN Train Set

Classifiers	F1-socre							Micro-F1	Macro-F1
	C1	C2	C3	C4	C5	C6	C7		
LightGBM	**0.9940**	**0.9977**	**0.9936**	**0.9914**	**0.9950**	**0.9896**	**0.9986**	**0.9940**	**0.9940**
RF	0.9959	0.9919	0.9877	0.9905	0.9941	0.9891	0.9973	0.9924	0.9923
XGBoost	0.9941	0.9932	0.9854	0.9887	0.9918	0.9891	0.9968	0.9913	0.9913
AdaBoost	0.8679	0.9444	0.7572	0.7601	0.8649	0.7879	0.8292	0.8316	0.8303
SVM	0.7585	0.6606	0.7447	0.5695	0.8362	0.5714	0.7853	0.7153	0.7038

TABLE V: Results of Five Classifiers based on SMOTETomek Train Set

Classifiers	F1-socre							Micro-F1	Macro-F1
	C1	C2	C3	C4	C5	C6	C7		
LightGBM	**0.9911**	**0.9877**	**0.9861**	**0.9878**	**0.9328**	**0.9334**	**0.9958**	**0.9736**	**0.9735**
RF	0.9899	0.9844	0.9836	0.9826	0.9340	0.9363	0.9928	0.9720	0.9719
XGBoost	0.9881	0.9830	0.9797	0.9809	0.9336	0.9367	0.9937	0.9709	0.9708
AdaBoost	0.8625	0.9432	0.7755	0.7987	0.8265	0.8165	0.8765	0.8436	0.8428
SVM	0.7525	0.6597	0.7527	0.4895	0.7544	0.7507	0.7611	0.7131	0.7029

TABLE VI: Results of Five Classifiers based on SMOTE Train Set

Classifiers	F1-socre							Micro-F1	Macro-F1
	C1	C2	C3	C4	C5	C6	C7		
LightGBM	**0.9876**	**0.9879**	**0.9797**	**0.9830**	**0.8867**	**0.8987**	**0.9917**	**0.9594**	**0.9594**
RF	0.9864	0.9876	0.9756	0.9772	0.8884	0.8934	0.9934	0.9575	0.9574
XGBoost	0.9844	0.9875	0.9730	0.9785	0.9024	0.9141	0.9880	0.9613	0.9611
AdaBoost	0.8283	0.9096	0.7467	0.7420	0.7715	0.7977	0.8344	0.8043	0.8043
SVM	0.7736	0.2217	0.5935	0.4937	0.5153	0.3939	0.5184	0.5083	0.5014

among all negative classes. As shown in Fig. 3, we can see that the highest TPR for each vulnerability is 100%, all TPRs exceed 98%; the highest TNR is 99.7%, all TNRs exceed 98%. The higher TPRs and TNRs demonstrate the efficiency of the MLCD classifier.

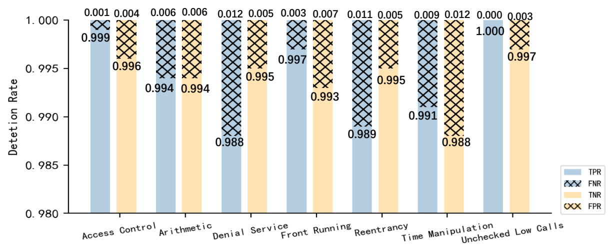


Fig. 3: TPRs, FNRs,TNRs,FPRs of our Scheme

2) ROC Curves: ROC curve is the abbreviation of receiver operating characteristic curve. The x-axis of the ROC curve denotes FPR and the y-axis denotes TPR. The area under the ROC curve represents the prediction accuracy and is known as the AUC value. Obviously, the bigger AUC means the higher prediction accuracy, whereas the opposite indicates the lower prediction accuracy. The ROC Curves of MLCD with LightGBM Classifier are presented in Fig. 4.

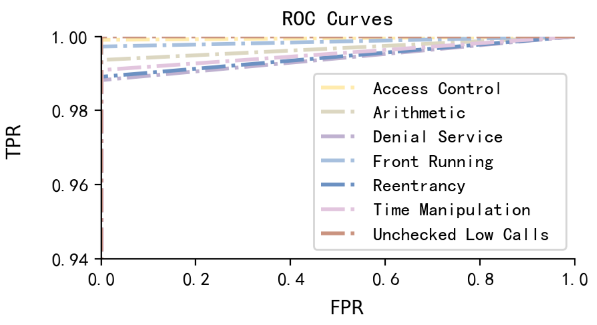


Fig. 4: ROC Curves of MLCD scheme with LightGBM Classifier

3) Comparison with Previous Studies: In the above results, the TPRs and TNRs values of seven vulnerabilities are used to prove the validity of the MLCD model. Based on the Smartbugs Wild dataset, we select EA-RGCN presented by

TABLE VII: Accuracy, Recall, Precision and F1-score

(a) Performance of arithmetic and reentrancy vulnerability detection tasks.

Methods	Arithmetic Vulnerability (%)				Reentrancy Vulnerability (%)			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
MLCD	**99.36**	**99.36**	**99.36**	**99.36**	**99.23**	**99.54**	**98.91**	**99.23**
EA-RGCN	90.47	91.16	89.00	90.03	94.00	94.92	93.96	94.42
CBGRU	86.54	87.23	85.66	86.43	93.30	96.30	85.95	90.92
HGAT	88.26	83.12	87.14	83.08	85.64	76.96	88.12	82.47

(b) Performance of timestamp manipulation and unchecked low calls vulnerability detection tasks.

Methods	Timestamp Manipulation Vulnerability (%)				Unchecked low calls Vulnerability (%)			
	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score
MLCD	**98.96**	**98.82**	**99.09**	**98.96**	**99.86**	**99.73**	**100.00**	**99.86**
EA-RGCN	91.98	95.06	91.66	93.35	83.33	81.48	89.50	84.65
CBGRU	93.02	89.47	97.45	93.29	-	-	-	-
HGAT	86.62	82.69	89.62	88.31	-	-	-	-

Chen et al. [37], CBGRU presented by Zhang et al. [38] and HGAT presented by Zhang et al. [39] for comparison. The comparative models are introduced as follows:

- EA-RGCN [37] uses RGCN to extract code content features, and uses multi-head attention to extract semantic features from edge sequences. Content and semantic features are connected as global code features for vulnerability identification.
- CBGRU [38] combines different word embeddings with different deep-learning methods to extract characteristics for vulnerability detection.
- HGAT [39] uses Abstract Syntax Trees (ASTs) and Control Flow Graphs to abstract functions into Code Graphs (CFGs), and then uses the Graph Attention Mechanism GAT to abstract each node in the code subgraphs and extract node features for vulnerability detection.

Actually, the accuracy, recall, precision and F1-score are applied to assess the vulnerability detection performance of the models, we choose four kinds of smart contract vulnerabilities for comparison. Table VII presents the comparison results.

The experimental results in Table VII show that our proposed MLCD outperforms EA-RGCN, CBGRU, and HGAT in terms of accuracy, precision, recall, and F1-score. MLCD achieves significant progress over EA-RGCN, which are all above 98%. In the unchecked low calls vulnerability detection, MLCD outperforms EA-RGCN by 16.53%, 18.25%, 10.5%, and 15.21% in accuracy, precision, recall, and F1-score. In the other three vulnerability detection tasks, MLCD can improve the performance of EA-RGCN by up to 3.82% to 10.36%; MLCD improves the performance of CBGRU by 1.64% to 13.70%; MLCD improves the performance of HGAT by 11.10% to 22.58%. Comparisons between MLCD, EA-RGCN, CBGRU, and HGAT show that our proposed smart contract representation of MLCD significantly improves the performance of the four vulnerability detection tasks.

Comparisons between MLCD, EA-RGCN, CBGRU, and HGAT show that our proposed smart contract representation significantly improves the performance of four vulnerability detection tasks. The EA-RGCN model combines content features and semantic features, and can only detect these four vulnerabilities, and the detection method is less general. CBGRU focuses on static code features and ignores data flow in the code, which is an important factor in detecting vulnerabilities. HGAT ignores instructions in code, whereas smart contracts run by instructions and can only detect specific vulnerabilities. on the other hand, the MLCD model extracts the opcode features and edge features of basic blocks based on the control flow graph information and uses PCA for feature dimensionality reduction. Our proposed MLCD can fully represent the statement execution order and data flow. Therefore, MLCD can describe more information about the code than EA-RGCN and CBGRU.

D. Implementation of Smart Contract Vulnerability Detection

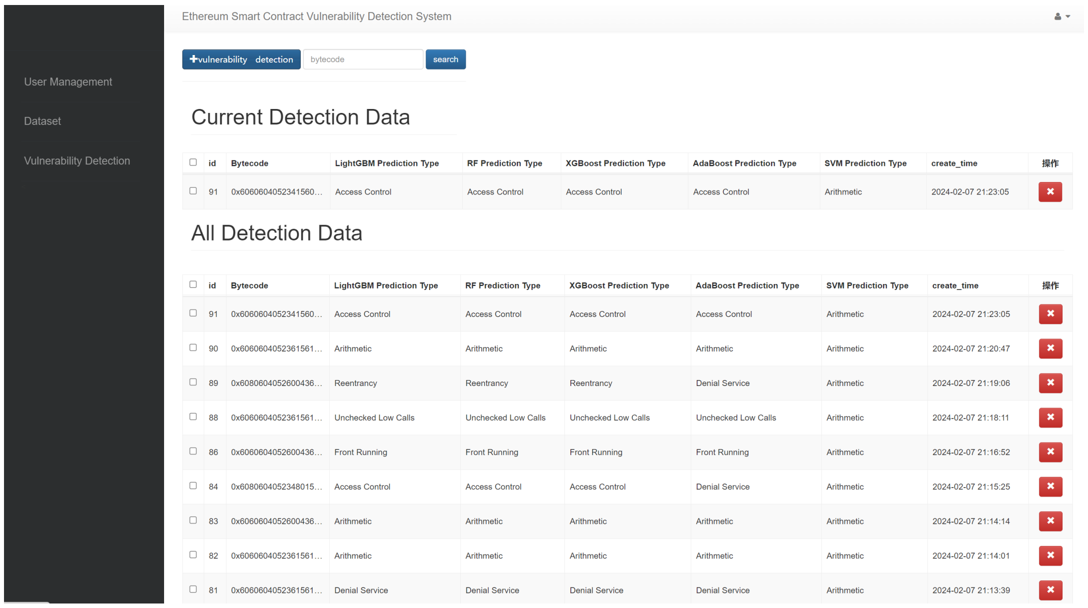
In Fig. 5, we develop a smart contract vulnerability detection system on the original training set. The vulnerability detection includes feature extraction module and classification module.

1) Feature extraction module: the control flow information of the smart contract is compiled by Vandal, which contains the basic code block information. We mainly extract the opcode features in each basic block and the edge features between all basic blocks from the control flow graph and the PCA is used for feature dimensionality reduction to get the dataset;
2) Classification module: firstly, the relevant classification models are obtained by extensive training on the training set. Secondly, the smart contract bytecodes are represented as feature vectors (prediction set). Finally, the data is entered into the classifier and the classification results are output.

However, this vulnerability detection system cannot detect data outside the prediction set. In the feature extraction module, we perform PCA dimensionality reduction on the same type of vulnerability data based on the labels. When entering new data that does not exist in the prediction set, we cannot perform PCA on it and make predictions.

VII. Analysis, Discussion and Conclusion

For access control, arithmetic overflow, denial service, reentrancy, time manipulation, and unchecked low calls vulnerabilities, we propose a general feature extraction method,



Current Detection Data

☐	id	Bytecode	LightGBM Prediction Type	RF Prediction Type	XGBoost Prediction Type	AdaBoost Prediction Type	SVM Prediction Type	create_time	操作
☐	91	0x6060604052341560…	Access Control	Access Control	Access Control	Access Control	Arithmetic	2024-02-07 21:23:05	✖

All Detection Data

☐	id	Bytecode	LightGBM Prediction Type	RF Prediction Type	XGBoost Prediction Type	AdaBoost Prediction Type	SVM Prediction Type	create_time	操作
☐	91	0x6060604052341560…	Access Control	Access Control	Access Control	Access Control	Arithmetic	2024-02-07 21:23:05	✖
☐	90	0x6060604052361561…	Arithmetic	Arithmetic	Arithmetic	Arithmetic	Arithmetic	2024-02-07 21:20:47	✖
☐	89	0x6080604052600436…	Reentrancy	Reentrancy	Reentrancy	Denial Service	Arithmetic	2024-02-07 21:19:06	✖
☐	88	0x6060604052361561…	Unchecked Low Calls	Unchecked Low Calls	Unchecked Low Calls	Unchecked Low Calls	Arithmetic	2024-02-07 21:18:11	✖
☐	86	0x6060604052600436…	Front Running	Front Running	Front Running	Front Running	Arithmetic	2024-02-07 21:16:52	✖
☐	84	0x6080604052348015…	Access Control	Access Control	Access Control	Denial Service	Arithmetic	2024-02-07 21:15:25	✖
☐	83	0x6060604052600436…	Arithmetic	Arithmetic	Arithmetic	Arithmetic	Arithmetic	2024-02-07 21:14:14	✖
☐	82	0x6060604052361561…	Arithmetic	Arithmetic	Arithmetic	Arithmetic	Arithmetic	2024-02-07 21:14:01	✖
☐	81	0x6060604052361561…	Denial Service	Denial Service	Denial Service	Denial Service	Arithmetic	2024-02-07 21:13:39	✖

Fig. 5: *Implementation of Smart Contract Vulnerability Detection*

namely the opcode features and edge features. As the opcode denotes the implementation logic of smart contract and includes information about the basic code blocks, we extract the opcode features in each basic block and the features of edges between all basic blocks from the control flow graph. Since Word2vec learns from the context-word matrix, the word vector of context co-occurrence features is obtained. Compared with the previous Embedding method, the effect is better and the method dimension is less. PCA decreases the dimension of high-dimensional data, maximizes the retention of information in the original data and speeds up the algorithm. Therefore, the opcode slicing feature can effectively describe the static smart contract.

The MLCD scheme is very efficient in detecting vulnerabilities in smart contracts. First, in the feature extraction phase, the opcode features and edge features are processed by PCA. Thus, the method simplifies the entered data and decreases the dimension. Second, the main idea of supervised learning is to use algorithms to learn function mappings from inputs to outputs and to predict output variables as new variables are entered. During the training phase, the proposed method gets the best model by constantly imitating and upgrading the mapping function's parameters. During the prediction stage, this MLCD can directly predict whether there are loopholes in new samples by the model obtained in the training stage.

There are three main limitations in this work. Firstly, this study focuses on detecting vulnerabilities in existing smart contracts. However, the proposed method can not detect unknown vulnerabilities or undefined new vulnerabilities. This paper focuses on seven kinds of smart contract vulnerabilities, but the feature extraction method is applicable to all smart contract opcodes, and if a new smart contract vulnerability emerges, this method can still be used to extract the opcode features of the vulnerability. Secondly, the method relies on the authenticity of the labels of Smartbugs Wild dataset detect vulnerabilities. Thirdly, we propose that the vulnerability detection system is unable to detect data outside the prediction set. Since in the feature extraction module, we perform PCA dimensionality reduction on the same type of vulnerability data based on the labels. When new data that does not exist in a dataset is entered, then we cannot perform PCA on it and make predictions.

In the coming work, we work on above limitations, and study new vulnerabilities of smart contracts and propose feature extraction methods that can better characterize the vulnerabilities.

References

[1] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, p. 21260.

[3] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[4] M. Zhang, M. Yang, G. Shen. "SSBAS-FA: A secure sealed-bid e-auction scheme with fair arbitration based on time-released blockchain". *Journal of System Archiecture*, vol 129, no.102619, 2022.

[5] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," vol. 151, no. 2014, 2014, pp. 1–32.

[6] A. Yasin and L. Liu, "An online identity and smart contract management system," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2016, pp. 192–198.

[7] M. Zhang, M. Yang, G. Shen, Z. Xia, Y. Wang. "A verifiable and privacy-preserving cloud mining pool selection scheme in blockchain of things". *Information Sciences*, vol. 623, pp. 293-310, 2023.

[8] E. Wall and G. Malm, "Using blockchain technology and smart contracts to create a distributed securities depository," 2016.

[9] Z. Wan, Z. Guan, and X. Cheng, "Pride: A private and decentralized usage-based insurance using blockchain," 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)*. IEEE, 2018, pp. 1349–1354.

[10] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.

[11] "The dao," 2016, [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability.

[12] "The parity multisig bug," 2017, [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[14] B. Mueller, "A framework for bug hunting on the ethereum blockchain," 2017.

[15] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[16] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," *arXiv preprint arXiv:2209.05872*, 2022.

[17] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[18] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[19] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[20] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "Contractguard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2019.

[21] B. Jiang, Y. Liu, and 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.

[22] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 65–68.

[23] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[24] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.

[25] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2020, pp. 3283–3290, main track. [Online]. Available: https://doi.org/10.24963/ijcai.2020/454

[26] W. Jie, Q. Chen, J. Wang, A. S. V. Koe, J. Li, P. Huang, Y. Wu, and Y. Wang, "A novel extended multimodal ai framework towards vulnerability detection in smart contracts," *Information Sciences*, vol. 636, p. 118907, 2023.

[27] Y. Zhai, J. Yang and M. Zhang. "An efficient vulnerability detection system for smart contracts using gated graph neural network" ACISP'24, LNCS, 2024

[28] L. M. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Ieee, 2018, pp. 1545–1550.

[29] H. Sam, "Batch overflow bug on ethereum erc20 token contracts and safemath[eb/ol]," 2022-5-25, [Online]. Available: https://blog.matryx.ai/batch-overflow-bug-on-ethereum-erc20-token-contracts-and-safemath-f9ebcc137434.

[30] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[31] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[32] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[33] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[34] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[35] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[36] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[37] D. Chen, L. Feng, Y. Fan, S. Shang, and Z. Wei, "Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention," *J. Syst. Softw.*, vol. 202, p. 111705, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:252236350

[38] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "Cbgru: A detection method of smart contract vulnerability based on a hybrid model," *Sensors*, vol. 22, no. 9, p. 3577, 2022.

[39] C. Ma, S. Liu, and G. Xu, "Hgat: smart contract vulnerability detection method based on hierarchical graph attention network," *Journal of Cloud Computing*, vol. 12, pp. 1–13, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259214756