A New Smart Contract Anomaly Detection Method by Fusing Opcode and Source Code Features for Blockchain Services

Li Duan, *Member, IEEE,* Liu Yang, Chunhong Liu, *Member, IEEE,* Wei Ni, *Senior Member, IEEE,* and Wei Wang, *Member, IEEE*

***Abstract*—Digital assets involved in smart contracts are on the rise. Security vulnerabilities in smart contracts have resulted in significant losses for the blockchain community. Existing smart contract vulnerability detection techniques have been typically single-purposed and focused only on the source code or opcode of contracts. This paper presents a new smart contract vulnerability detection method, which extracts features from different levels of smart contracts to train machine learning models for effective detection of vulnerabilities. Specifically, we propose to extract 2-gram features from the opcodes of smart contracts and token features from the source code using a pre-trained CodeBERT model, thereby capturing the semantic information of smart contracts at different levels. The 2-gram and token features are separately aggregated and then fused and input into machine-learning models to mine the vulnerability features of contracts. Over 10,266 smart contracts are used to verify the proposed method. Widespread reentrancy, timestamp dependence, and transaction-ordering dependence vulnerabilities are considered. Experiments show the fused features can help significantly improve smart contract vulnerability detection compared to the single-level features. The detection accuracy is as high as 98%, 98% and 94% for the three vulnerabilities, respectively. The average detection time is 0.99 second per contract, indicating the proposed method is suitable for automatic batch detection of vulnerabilities in smart contracts.**

***Index Terms*—Blockchain, Smart Contract, Vulnerability Detection, Machine Learning, Feature Extraction.**

I. Introduction

SMART contracts have been increasingly developed in conjunction with blockchains since the emergence of the latter. Blockchains help smart contracts to address the trust problem of smart contracts, and provide a trusted automatic execution environment [1] and technical guarantee. Recently, smart contracts have been deployed on many blockchain platforms, including Ethereum, EOS, VAT Chain and more. Since January 1, 2022, Ethereum has verified a total of 63,446 smart contracts. However, many smart contract vulnerabilities remain to be addressed.

L. Duan, L. Yang, and W. Wang are with the Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China (email: duanli@bjtu.edu.cn; liuy3990@163.com; wangwei1@bjtu.edu.cn).

L. Duan is with the Guangxi Key Laboratory of Cryptography and Information Security, Guilin, Guangxi.

C. H. Liu is with the Department of Computer and Information Engineering, Henan Normal University, Xinxiang 453007, China (email: LCH@htu.edu.cn).

W. Ni is with the Data61, CSIRO, Marsfield, NSW 2122, Australia (email: Wei.Ni@data61.csiro.au).

The corresponding authors are W. Wang and C. H. Liu.

With the immutability characteristic of blockchain, smart contracts cannot be modified once they are deployed [2]. In the process of code design, coding loopholes, design defects, and other problems inevitably occur. Any defective codes may have disastrous consequences because smart contracts are often applied to manage large amounts of tokenized assets and access rights between different entities [3]. For example, the DAO is that hackers stole 3.6 million Ethers in 2016 by attacking the reentrancy vulnerability [4]. In 2017, $30 million worth of Ether was stolen from the Parity Multisig wallet due to the Delegatecall vulnerability [5]. Ethereum lost more than $1.3 billion in security incidents in 2021 alone. On October 15, 2021, Sophos reported that the crypto-fraud app CryptoRom stole $1.4 million by "super-signature service" and Apple's Developer Enterprise program. All these cause significant losses to users.

The reasons underlying smart contract security vulnerabilities are as follows: 1) In the face of the new programming languages and tools, developers cannot fully understand its operation logic, such as the future state or environment of the contract. 2) Unlike traditional applications, smart contracts cannot be modified or updated once they are deployed due to the immutability of blockchains, even if a vulnerability is found during the operation. 3) Smart contracts as important controllers of digital currency are tempting for hackers. Compared with other software, contracts exposed to open environments are more vulnerable to malicious attacks. Effective contract vulnerability detection methods are crucial to address the security of smart contracts.

With the recent expansion of smart contracts and the continuous increase of their functions, the interactions between contracts have become increasingly complex. The demand for efficient smart contract vulnerability detection is growing. The critical challenges of smart contract vulnerability detection include automating vulnerability detection, especially in support of batch processing of smart contracts in numbers, and improving the accuracy and efficiency of vulnerability detection.

Existing methods for rule-based smart contract vulnerability detection have been primarily conducted by static analysis or dynamic execution [6]–[8], which usually rely on experts to define vulnerability rules. For the formal validation tool VaaS [6] and the fuzzy detection tool ContractFuzzer [8], the average detection time of each smart contract is as long as 159.4 seconds and 352.2 seconds, respectively. The interme-

diate representation tool, Slither [9], and the symbolic execution tool, Oyente [7], have relatively high accuracy among existing detection methods. However, Oyente and Slither were designed to take their inputs, i.e., smart contracts, one by one, and would require additional programming efforts to create an interface for inputting batches of smart contracts. Moreover, the processing speeds of Oyente and Slither could be too slow to support batch processing. It takes Oyente an average of 18.48 seconds per contract [10] and it takes Slither about 5 seconds per contract [11].

Additionally, researchers have begun to apply machine learning to detect smart contract vulnerabilities. Studies based on natural language processing have shown that by analyzing the semantic and syntactic relationships of contracts [12]–[15], vulnerability detection can be implemented. Graph neural networks (GNNs) and clone detection have also been used for vulnerability detection [16]. By exploiting GNNs, the authors of [17] combined graph features with expert knowledge, while the authors of [18] analyzed the bytecode node features and semantic features. In [19], a comprehensive review of cross-chain systems was provided, covering security threats and attack paths and highlighting the importance of vulnerability detections.

From natural language processing to graph neural networks, the methods for the feature extraction of smart contracts have been used to extract contract context, data flow and control dependency. Studies based on machine learning have brought new vitality to the vulnerability detection of smart contracts. However, the existing vulnerability detection methods focus on a particular level of contract features, making the methods ineffective in feature representation. The loss of contract information makes part of the feature statistics obsolete, leading to the failures of the model in accurately detecting vulnerabilities [20]. The accuracy and efficiency are compromised accordingly. For this reason, the existing methods for smart contract vulnerability detection are unable to rise to the current challenges.

In this paper, we propose a new vulnerability detection method by fusing opcode and source code. Different from existing studies focusing on feature analysis and model optimization of smart contracts at a particular level, the feature representations of different levels, e.g., opcode and source code, are extracted from smart contracts and fused to improve the accuracy of vulnerability detection. A machine learning-based vulnerability detection model is built to train and defect the fused features. The fused features comprehensively capture contract information benefiting machine learning models to learn vulnerability patterns correctly and effectively.

Seven machine learning models are built, including K-Nearest Neighbor (KNN) [21], Support Vector Machine (SVM) [22], Convolutional Neural Network (CNN) [23], Random Forest (RF) [24], Multi-Layer Perceptron (MLP) [25], Recurrent Neural Network (RNN) [26] and Long and Short-Term Memory Network (LSTM) [27]. They are compared under different input features, i.e., fused opcode and source code features, or individual features. The RF model with fused feature input is identified for its superior accuracy.

The key contributions of this paper are as follows.

- We propose a new smart contract vulnerability detection method that fuses opcode features with source code features. The opcode and source code features of smart contracts are extracted and fused to improve the detection of vulnerabilities in smart contracts.
- Smart contract opcodes are comprehensively analyzed. The new 2-gram features of opcodes are extracted to capture the semantics of smart contracts at a lower level. Token features are extracted from the source code through CodeBERT [28]. The two types of features are fused to form the smart contract feature matrix as the input of machine learning-based detection models.
- Three key vulnerabilities of smart contracts are detected, namely, reentrancy vulnerability, timestamp dependence vulnerability, and transaction-ordering dependence vulnerability. A balanced dataset is created based on the Ethereum dataset and used to select the machine learning models and gauge the efficiency of the proposed method.

Extensive extensive experiments show that the macro-F1 is as high as 93.3% and the mAP is 96.7% under the proposed method with RF as the vulnerability detection model and fused opcode and source code features as the input to the model. The average detection time of each smart contract is as short as 0.99 second, considerably better than those of the mainstream vulnerability detection tools, Oyente [7] and Securify [29].

The rest of this paper is organized as follows. The related studies are reviewed in Section II. Section III introduces Ethereum and smart contracts. The three vulnerabilities we study are analyzed in Section IV. In Section V, the proposed smart contract vulnerability detection framework is described, and the new feature extraction is elaborated on. In Section VI, experiments and performance evaluation are provided, followed by conclusions in Section VII.

II. Related Work

At present, smart contract vulnerability detection methods can be mainly divided into two categories: rule-based smart contract vulnerability detection and machine learning-based vulnerability detection.

A. Rule-based smart contract vulnerability detection

Rule-based refers to the matching of contract behaviors with limited vulnerability patterns identified in advance. The existing work on rule-based smart contract vulnerability detection defines vulnerability rules manually, which makes it easy for attackers to bypass the inherent rules of the system. Thus bringing great risks to the contracts.

The existing vulnerability detection methods consist primarily of formal validation methods, symbolic execution methods, fuzzy detection methods, and intermediate representation methods. VaaS [6] was the first automatic formal verification platform based on formal verification, which supports blockchain smart contracts, such as EOS and Ethereum. It adopts a "one-click" mode to locate contract risks and indicates the reasons, hence effectively detecting routine risks. Oyente [7] is a vulnerability detection tool based on symbolic execution, which takes contract bytecode as input. However, it

may not cover all execution paths, resulting in false negatives [12]. Chen *et al.* [10] proposed DefectChecker, which also utilized symbolic execution to detect smart contract defects. ContractFuzzer [8] was the first fuzzy detection framework for detecting security vulnerabilities in Ethereum smart contracts. Offline EVM testing tools and online fuzzy detection tools are involved in support of more detection types. However, ContractFuzzer does not support arbitrary assertions in the Solidity code, as differences are likely to be caused by specification semantics, rather than exploration capability [30]. The formal semantics of the EVM designed by the F* framework and the K framework provides strong formal verification guarantees, but they are still semi-automated [17]. The source code of a contract can be converted into an intermediate representation opcode with an effective semantic expression captured. Slither [9] takes an abstract syntax tree (AST) obtained from the source code compilation of a smart contract as the input. In addition to the detection results, Slither can also output code optimization suggestions.

Manual rules designed by experts based on individuals' experience may not provide consistency and accuracy [31]. Some tools also take a long time to detect. For the formal validation tool, VaaS [6], and the fuzzy detection tool, ContractFuzzer [8], the average detection time of a smart contract can be as long as 159.4 seconds and 352.2 seconds, respectively. By contrast, we aim to train a vulnerability detection model to learn the characteristics of contracts, as opposed to manually defining the rules. This contributes to the efficiency of vulnerability detection.

B. Machine learning-based vulnerability detection

Existing machine learning-based studies have primarily focused on vulnerability detection based on natural language processing and non-Euclidean graph.

1) Vulnerability detection based on natural language processing: Natural language processing considers contract codes as texts and analyzes their semantic and syntactic relationships and the dependence between their control flow and data flow. Through appropriate model learning and target setting, vulnerability detection and code similarity detection can be effectively implemented. Specifically, source codes and compiled opcodes can be standardized in the stage of text processing. For example, Wang *et al.* [12] proposed ContractWard, which parses bytecodes into opcodes and then extracts the binary features from the opcodes as the input to machine learning models. ContractWard retains the opcode information and semantic features, but can only detect predefined vulnerabilities. Li *et al.* [13] proposed VulDeePecker, which slices target programs into code snippets and transforms the code snippets into vectors. Huang *et al.* [14] proposed a Multi-Task learning model, which extracts contract features using a neural network based on the attention mechanism. However, the model adopted a hard parameter-sharing method, which may compromise the expressiveness of the model and reduce the generalization ability of the model [32].

Some other methods have been developed based on bytecodes. For example, considering the differences between programming language and natural language, the authors of Eth2Vec [15] analyzed EVM bytecodes to create the JSON files of different levels as input to add syntax-relevant information. The current construction of Eth2Vec is in an unsupervised setting. Yet, the supervised setting could be more suitable for the classification of natural language processing than the unsupervised setting, according to [33].

2) Vulnerability detection based on non-Euclidean graph: The methods based on non-Euclidean graphs have a strong impact on smart contract vulnerability detection. Liu *et al.* [17] proposed a vulnerability detection method based on a GNN, which converts the control flow and data flow of a source code into a contract graph. However, expert knowledge was used to determine labels, which can be subjective to some extent and difficult to generalize to other vulnerabilities. Using GNNs, Zhao *et al.* [18] analyzed node features and semantic features from bytecodes to implement vulnerability detection. Wu *et al.* [34] proposed Peculiar, which is a technology based on crucial data flow graphs and pre-training techniques and can better identify reentrancy vulnerabilities. However, these two approaches focused on only bytecodes or source codes and did not support joint consideration.

Graph convolutional networks (GCNs) [17] can be used to obtain graph embedding or achieve graph classification. Xu *et al.* [35] verified that a GNN's recognition ability is almost as powerful as a Weisfeiler-Lehman graph kernel (WL) test in distinguishing graph structures. Apart from being directly used for classification, approximate graph matching can be adopted based on generated graphs. For example, the authors of [16] proposed CCGraph to perform kernel calculation on graphs by improving WL and then calculate the similarity between graphs to achieve clone detection. CCGraph is a PDG-based clone detection work, but there are no open-source or standard benchmark datasets for this method. Huang *et al.* [36] encoded code graphs into quantitatively comparable vectors, and vulnerability was tested through code similarity. They verified dangerous contracts by deploying them on a private blockchain, providing a similar environment to the real blockchain and observing the results. However, this is not automated.

All these studies bring new vitality to smart contract vulnerability detection. However, existing research on machine learning-based contract vulnerability detection is still in its infancy. The vulnerability types targeted are also relatively limited. Moreover, the existing methods focus on the feature analysis and model optimization of smart contracts at a specific level, making the feature statistics incomplete in representation. In contrast, we jointly consider the feature representation of different levels in our proposed vulnerability detection method.

III. Background

In this section, Ethereum smart contracts, their source codes, bytecodes and opcodes are introduced in brief. Ethereum smart contracts are considered here for their proliferation.

A. Ethereum Smart Contract

Ethereum is the largest platform that provides a development and execution environment for smart contracts, with

many application programming interfaces (APIs). A large number of distributed applications have been developed and deployed on the Ethereum blockchain as smart contracts, e.g., using the Solidity language. EVM, Ethereum's core structure for Turing-complete computing, is a stack-based virtual machine that executes smart contract bytecode. When a smart contract is called, EVM performs actions and stack operations according to the corresponding instructions of the smart contract. During the operation, EVM interacts with the storage space.

In the Ethereum platform, a smart contract is an immutable piece of program code deployed on the blockchain. The life cycle of a smart contract consists of five phases, namely, design, development, deployment, invocation and destruction. As it is immutable, there is no maintenance phase, which is different from a traditional code. First, demand analyses are conducted, including outline design and detailed design of the contract. After determining its overall structure, programmers begin to develop smart contracts. Then, a satisfying contract will be deployed to the blockchain. The deployment of a contract is achieved by sending a transaction to the blockchain. The transaction contains information, such as bytecode, gas value and the empty receiver. The address returned by a miner after packaging the transaction into the blockchain is the contract address, which is the unique identification of the contract.

The call and access of a contract will require the use of the contract address and the application binary interface (ABI). There are two types of calls, sending transactions and messages. The originator of the transaction is primarily an external account, and a message refers to a message call between internal accounts (contract accounts). Once a transaction is initiated, it will be broadcast to the blockchain network. However, the message is the transmission of internal parameters. Thus, transactions are recorded in the blockchain, while messages are not. In order to avoid the pressure of endless user calls and malicious codes on the network, Ethereum introduced a Gas mechanism to limit the number of computing resources used to execute smart contracts. By making the contract caller pay the corresponding price to increase the cost of attacking, the safe operation of a contract is assured to a certain extent.

B. Smart Contract Source Code, Bytecode, and Opcode

A smart contract can be written in Solidity, LLL, Serpent, and other languages, among which Solidity is popular and has been extensively used [37]. Solidity is a contract-oriented, high-level, programming language designed to develop smart contracts run on the Ethereum Virtual Machine (EVM). A source code written in Solidity needs to be compiled into an EVM bytecode to run on the EVM. In this paper, we focus on the Solidity language.

The language updates rapidly, so the same keyword may mean differently in different versions. An appropriate compiler version is needed according to the contract. A bytecode as the only executable code on EVM is made up of hexadecimal numbers. Smart contracts are stored in the Ethereum network in the form of bytecode. To better understand the semantic relationships in contracts, bytecode is parsed into an intermediate representation opcode with a higher semantic expression by de-compilation. Opcode is more readable than bytecode.

There are 125 opcodes specified in Ethereum Yellow Book [38], corresponding to different state transition rules of bytecode instructions. There are ten functions, including stop and arithmetic operation, comparison and bit logic operation, SHA3 operation, and so on. Fig. 1 shows the relationship among source code, bytecode and opcode in various forms of smart contracts, where each level has its presentation.

In this paper, we use opcodes and source codes to detect the vulnerabilities of smart contracts. According to the Ethereum Yellow Book [38], opcodes provide one-to-one correspondence with bytecodes, while opcodes have better readability. On the other hand, the source codes of smart contracts have been widely used for vulnerability detection. There are several publicly available smart contract datasets, including contract addresses, source codes, and opcodes, such as Smart Contract Dataset [17] and Ponzi Contract Dataset. The information on the latest 500 verified smart contracts on Ethereum, including their source codes, is also updated in real-time at https://etherscan.io/contractsVerified.

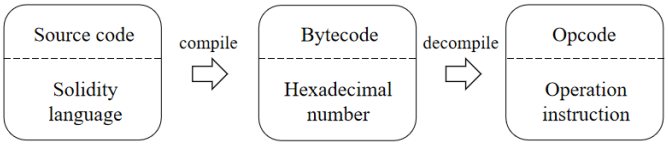


Fig. 1: The relationship between the source code, bytecode, and opcode of smart contracts.

IV. Vulnerability Analysis

In this paper, three vulnerabilities, namely, reentrancy vulnerability, timestamp dependence vulnerability, and transaction-ordering dependence vulnerability, are considered for the following reasons: 1) These three vulnerabilities are among the ten most frequent types of Ethereum smart contract vulnerabilities and have caused huge losses to Ethereum. The DAO incident in 2016 alone caused a loss of nearly $60 million. 2) The three vulnerabilities have a great impact on smart contracts. According to [15], about 5,013 of 307,396 Ethereum smart contract functions contain call.Value, and about 4,833 have block.timestamp. These functions are susceptible to the reentrency vulnerability and timestamp dependence vulnerability. Devastating front-end attacks could also occur in the face of transaction-ordering dependence vulnerability. To this end, it is important to ensure that transaction information is presented in an appropriate form at reasonable stages by leveraging a combination of Gas restrictions and disclosure schemes.

A. Reentrancy Vulnerability

The key risk associated with this vulnerability is that it calls external contracts, which may take over the control process of a smart contract. In a reentrancy attack, a malicious contract calls back the contract before the first call of the function is complete. This can cause different calls of the

function to interact in unexpected ways. Before introducing vulnerabilities, the fallback function needs to be called, which is a special nameless function in a contract. If the contract call does not match a function identifier or data is not provided, such as a transfer function with only Ether but no data, then the fallback function is automatically called. When the victim contract transfers funds to the attacker (who is an authorized but misbehaved user of the contract), the contract will automatically call the attacker's fallback function F. As shown in Fig. 2, the attacker has deposited 10 Ethers in the account of the victim contract in advance. When the attacker withdraws funds from the victim contract, the withdraw function is called, and the contract begins to transfer funds to the attacker. Then, the fallback function F of the attacker is automatically called. The withdraw function of the victim contract will be repeatedly called in the fallback function F. The attacker can take advantage of this status defect of the victim contract to make multiple transfer operations, until the contract account is empty or the fallback function F reaches its limit.

The above-mentioned status defect refers to that the state variable is set when a contract transfers funds normally, and then the state variable does not change when the attacker repeatedly calls the transfer function of the contract. In this case, the contract believes that a normal transfer transaction is ongoing, resulting in its account loss. To this end, a key operation to prevent the reentrancy vulnerability is to make sure that all operations that can change state variables take place before any external call, thereby making the code that executes an external call the last action of the code segment.

Victim contract

```
contract A{
    mapping (address => unit) private userBalance;

    function deposit() payable{
        userBalance[msg.sender]+=msg.value;
    }
    function withdraw() public{
        uint amount = userBalance[msg.sender];
        require(msg.sender.call.value(amount)());
        userBalance[msg.sender] = 0;
    }
}
```

Attacker contract

```
contract B{
    address bank_ add=01f3x...32;
    function attack(){
        bank_ add.deposit.value(10)();
        bank_ add.withdraw();
    }
    function () payable{
        if(count ++< 10)
        bank_ add.withdraw();
    }
}
```

Fig. 2: An example of reentrancy vulnerability.

B. Timestamp Dependence Vulnerability

In this vulnerability, the execution of a smart contract depends on the timestamp of the current block. If the timestamps are different, the execution results of the contract will be different. The timestamp obtained in a smart contract depends on the local time of a miner node, and a miner can manipulate its timestamp by up to 900 seconds. It is possible for the miners to obtain illegal gains by maliciously modifying block timestamps. The basis of determining the vulnerability is whether the timestamp is used in a judgment condition for critical operations in a contract. A simplified example is shown in Figure 3, which is like a simple lottery. Only one transaction per block can bet 10 Ethers for a chance to get a contract balance. According to the logic of the contract, the odds of playing the lottery are 1 in 15. A miner can be incentivized to tamper with the timestamp of such a block to receive an Ether reward in the contract.

A game contract

```
contract Game{
    uint public pastBlockTime;
    constructor() public payable{}
    function () external payable{
        require(msg.value == 10 ether);
        require(now != pastBlockTime);
        pastBlockTime = now;
        if(now % 15 ==0){
            msg.sender.transfer(address(this).balance);}
    }
}
```

Fig. 3: An example of the timestamp dependence vulnerability.

C. Transaction-Ordering Dependence Vulnerability

The cause of this vulnerability is that the transaction order is not maintained in accordance with the submission order. In a period of time, multiple transactions are broadcast and agreed to be packaged in the same block by miners. New blocks are confirmed about every 17 seconds. The miners check the received transactions after the block is confirmed and determine the block transaction order based on the size of the gas value, leading to a different order of the transactions being executed from the order of the transactions packaged. When the transactions are sent to Ethereum, the messages are forwarded to each node. The users running Ethereum nodes know about transactions ahead of time. When a malicious user runs an Ethereum node, it is easy to attack transactions.

An example is a bounty contract that rewards the first person who solves a problem, as shown in Fig. 4. Since a bounty contract only contains the constructor and fallback function, the fallback function will be automatically executed when the contract is called. If the caller of the contract is the owner, the reward will be updated and the previously set reward value will be returned to the caller. Otherwise, a user solves the problem, and the answer submitted by the user is judged. If the answer$<$diff, the answer is confirmed and the user is rewarded. Here, diff is the pre-defined difficulty of the problem.

However, something unexpected may happen. For example, a smart user solves the problem and submits the answer to the network. At this point, a malicious user running the Ethereum node sees the answer to the problem in the network and submits it to the network with a higher gas value. The miner gives priority to the malicious user's transaction, according to the transaction rules. As a consequence, the malicious user is rewarded, not the original user who solved the problem. In other words, the malicious user can steal solutions and replicate its transactions at a high cost to preempt the original solutions.

V. Proposed Smart Contract Vulnerability Detection Framework

In this section, we delineate the proposed vulnerability detection method, which extracts contract features from opcodes and source codes and detects smart contract vulnerabilities using machine learning techniques. To better understand the smart contract features, bytecodes are compiled into an intermediate representation of opcodes in our method. The

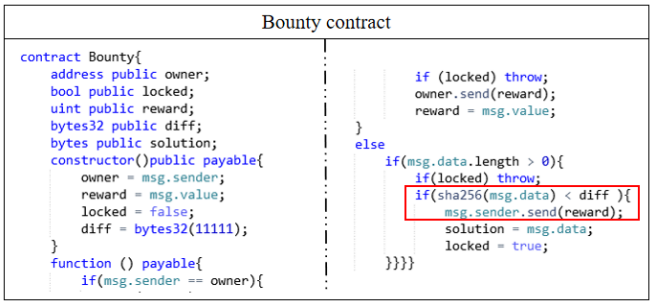


Fig. 4: An example of transaction-ordering dependence vulnerability.

opcodes contain a large number of assembly instructions and high semantic expressions, helping analyze the logical relationship between smart contract codes at a lower level. The vulnerability information is often reflected by the abnormal state transition rules of assembly instructions.

An important aspect of the proposed method is to capture the semantics of consecutive opcode instructions. The 2-gram features of the opcodes are extracted to capture the semantics of consecutive opcode instructions. There are also a lot of similar features in the contracts due to the similarity of smart contract code language. The risk statements of contracts with vulnerabilities can coincide. To reflect the keywords and other key information in smart contracts, CodeBERT is used to extract token features from the contract source codes. Our method feeds the 2-gram features of the opcodes, the token features of the source codes, and the opcode instructions into a machine-learning model for vulnerability detection.

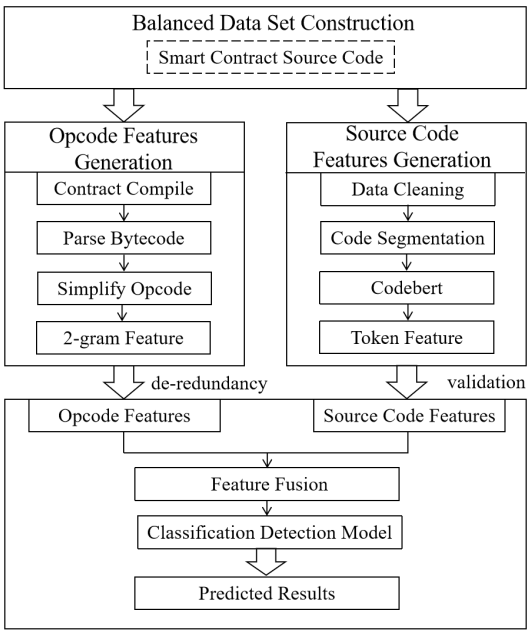


Fig. 5: The proposed smart contract vulnerability detection framework.

As shown in Fig. 5, the proposed method consists of three stages: 1) Opcode feature generation. According to the version of each smart contract, the corresponding solc compiler is selected to generate its bytecode. Then, the bytecode is parsed into an opcode for comprehensive analysis. The 2-gram feature of the opcode is extracted. 2) Source code feature generation. After the smart contract source code is cleaned, the token features are extracted using CodeBERT. The obtained token segments are used as the source code features of the contracts. 3) Vulnerability prediction. To train the detection models, the opcode and the source code features are fused to generate the feature matrix. The prediction results of contracts are output.

A. Opcode Feature Extraction

Opcodes are used as the features of smart contract vulnerability detection, derived from smart contract source codes or bytecodes. Since the dataset obtained contains contract source codes, a series of data pre-processing is performed, including contract source code compilation, bytecode de-compilation, opcode simplification, etc.

Contract compilation. The local solc compilers are used for batch processing of smart contracts. During compilation, the following may take place: 1) Compilation fails; 2) bytecode generated by compilation is empty; and 3) the compiled code contains non-hexadecimal characters.

Bytecode de-compilation. For Ethereum smart contracts, the Solidity source code, bytecode generated by the solc compiler, and de-compiled opcode yield one-to-one correspondence. The contract bytecodes generated by the local compiler are de-compiled into opcodes by the assemble_hex function in the pyevmasm library, as illustrated in Fig. 6.

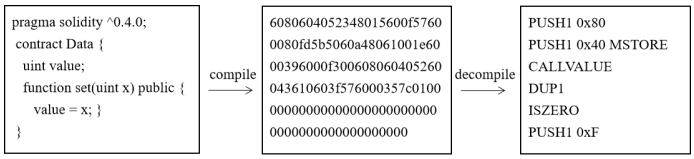


Fig. 6: A concrete example.

Opcode simplification. Given the variety of opcodes, the direct use of feature dimension can be too large, which is likely to cause dimension disaster and is not conducive to embodying the features of vulnerabilities. Therefore, we simplify the smart contract opcodes based on word meaning analysis. The simplification rules are as follows: 1) Operands are removed. Based on the rules and key instructions [38], there is an operand after each push instruction in the opcodes parsed by bytecodes. The operands have little significance to the features of the contracts and can be removed. 2) Opcodes that are functionally similar are grouped.

For example, the opcodes pushed onto the stack, e.g., PUSH1, PUSH2, PUSH3, etc., are grouped to be represented by PUSH. Both BLOCKHASH and TIMESTAMP belong to the class of Block Information with the only difference in the type of information [38]. DIFFICULTY, GASLIMIT, and COINBASE also belong to the class of Block Information. To this end, we group them into one group, CONSTANT1, to reduce the number of instructions with little loss of semantics. Likewise, logical operation opcodes, arithmetic comparison opcodes, and address-dependent opcodes are separately

TABLE I: Opcode simplification rules.

Simplified opcode	Original opcode
PUSH	PUSH1-PUSH32
DUP	DUP1-DUP16
LOG	LOG0-LOG4
SWAP	SWAP1-SWAP16
CONSTANT1	BLOCKHASH, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, COINBASE
ARITHMETIC	ADD, MUL, SUB, DIV, SDIV, SMOD, MOD, ADDMOD, MULMOD, EXP,SHR,SHL,SAR
CONSTANT2	ADDRESS, ORIGIN, CALLER
COMPARISON	SLT, SGT, LT, GT
LOGIC_OP	AND, OR, XOR, NOT

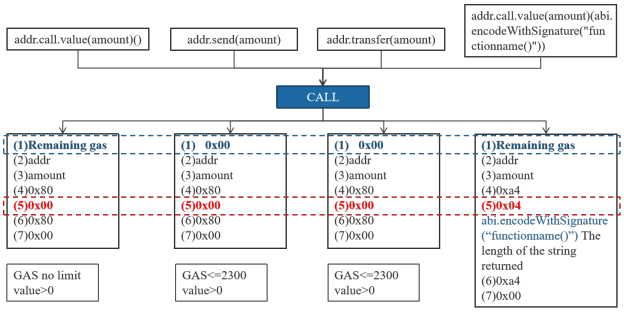


Fig. 7: Corresponding parameters of the transfer instruction.

grouped into three different classes. As a result, the 125 smart contract opcodes specified by Ethereum Yellow Paper [38] are grouped into 51 groups, with rules specified in Table I. Based on the rules and key instructions, this grouping method reduces complexity and prevents semantic losses.

Feature definition and extraction. There are logical relationships between smart contract source codes that persist after the codes are compiled into opcodes. The opcode is different from the source code in features expression. For example, there are generally three transfer methods in smart contracts, namely, transfer(), send() and call(). All these transfer methods will be converted into CALL instructions when compiled into opcodes. The parameters of the CALL are "gas," "to," "value," "in offset," " in size," "out offset," and "out size" in sequence.

The specific instruction parameters are shown in Fig. 7, where Call.value(α)(ρ) indicates that the function represented by ρ is called and the amount α is transferred. When the second parameter α is empty, no function is executed, then the fallback function will be executed after the call fails. As reflected in the opcode, an attacker can use the fallback function to call the victim contract, if the fifth parameter of the call instruction in size is empty. In addition, the call() method has no limit on the gas value, but there is a limit of GAS$<$2300 for transfer() and send(). This allows the attacker to call back the victim contract multiple times through the fallback mechanism when using call() and the second parameter is empty. Attackers may steal the contract account until the gas value is 0; i.e., a reentrancy vulnerability occurs. The key opcodes in the reentrancy vulnerability include CALL and GAS, and the corresponding functions are shown in Table II.

TABLE II: Opcodes and their functions.

Key opcode	Function
CALL	Call methods in other contracts and can send Ethers.
GAS	Capture the remaining GAS.

Based on the key opcodes listed in Table II, the reentrancy vulnerability detection rules are defined as follows: 1) Transferring money first and then modifying state variables. The state variables related to the transfer statement are not modified before the transfer is executed. For example, balances[msg.sender]-=_amount is executed after a contract transfer. 2) A call() transfer operation exists. When the call() method exists in a contract, we assess whether the value is greater than 0 and whether the gas value is sufficient by checking the stack parameters Gas and value of the CALL instruction. 3) The status variable does not change before and after a transfer, which means after a contract transfer, the account balance record changes but the status variable does not change. Specifically, we determine whether the state variable corresponding to the SSTORE opcode after the transfer operation is the same as the state variable used before the transfer operation, i.e., by assessing the SSTORE stack parameters. If the parameters have not changed, an attacker has repeatedly used the state variable before modification, causing a reentrancy vulnerability.

On Ethereum, block timestamps are important to serve as a time variable in a conditional statement of contracts. A smart contract exhibits timestamp dependence vulnerabilities. The contract is vulnerable to malicious attacks, when a timestamp is used in the conditional statement of the contract and the transfer functions, transfer(), send(), and call(), are called. To meet favorable conditions and attack a contract, a malicious miner may modify the timestamp (by up to 900 seconds), as mentioned in Section IV-B. The key opcodes in the timestamp dependence vulnerability include TIMESTAMP, NUMBER, SELFDESTRUCT, CALL, GAS, etc. The corresponding functions are shown in Table III.

TABLE III: Opcodes and their functions.

Key opcode	Function
TIMESTAMP	The timestamp of the current block, in seconds.
NUMBER	Current Block number.
SELFDESTRUCT	Suspend execution and register accounts for.
CALL	Call methods in other contracts and can send Ethers.
GAS	Capture the remaining GAS.

The detection rules of timestamp dependence vulnerabilities are defined as follows: 1) Timestamp is used as a variable in the conditional statement, and 2) the outcome of a conditional statement can affect the execution of key operations, such as transfer and selfdestruct. When the keywords, such as block.number, now, and block.timestamp, appear in the source code, the corresponding smart contract opcodes will contain TIMESTAMP and NUMBER. The transfer operations in the source code of the contract will be converted into assembly instructions, such as CALL and SELFDESTRUCT. In Fig. 8, a timestamp dependence vulnerability is taken as an example. On the condition of the if statement, the obtained timestamp is modulo computed and judged, which is reflected in the opcode

as two iszeros. The function of the first iszero is to assess whether the operation result is "0." If it is "0," then "1" is pushed onto the stack. The latter iszero evaluates the results of the push. Through two iszeros, the conditional statement is explained, helping understand the relationship between the source code and opcode and indicating the logical relationship between opcodes is not omissible.

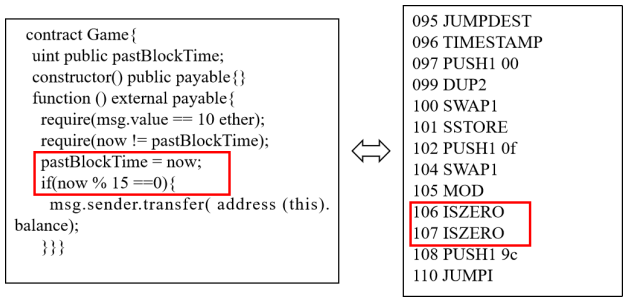


Fig. 8: The relationship between source code and opcode.

TABLE IV: Opcodes and their functions.

Key opcode	Function
ISZERO	Check whether the result is zero.
EQ&LT	Used for comparison.
JUMPI	Jump as the condition is true.
GAS	Capture the remaining GAS.
SWAP	Exchange items information in stack.
CALL	Call methods in other contracts and can send Ethers.
CALLVALUE	Get deposited value by the instruction or transaction.

Transaction-ordering dependence vulnerabilities often appear in lucrative contracts, such as bounty contracts. When a reward problem is solved by a user (or in other words, the caller of a contract provides the answer to the problem), the answer will be forwarded to each Ethereum node. A malicious user who runs Ethereum nodes can steal information and submit the answer with a higher gas value. Moreover, the order in which the miners execute transactions in a block depends on the gas values. The higher a gas value is, the higher its execution order. Therefore, the transaction of a malicious attacker will be preferentially executed to get an undeserved reward. In a bounty contract, each name and answer of the user who solves a question are recorded when a contract is called. The answer is automatically verified, as soon as the preset condition is met. When a malicious user steals answers, the reward is incorrectly issued and a transaction-ordering dependence vulnerability occurs.

The key opcodes in the transaction-ordering dependence vulnerability include ISZERO, EQ&LT, JUMPI, GAS, SWAP, CALL, CALLVALUE, etc. The corresponding functions are shown in Table IV. The detection rules of transaction-ordering dependence vulnerability are defined as follows: 1) Multiple comparisons of stack storage information exist in a contract, such as iszero, EQ, LT, and other opcodes. In a bounty contract, the authentication and answer comparison are both included to execute the contract. 2) When meeting a certain condition (i.e., specified in a conditional statement), the execution of statements jumps to operations, such as transfer and assignment. For example, the jump instruction is triggered when a conditional statement is met. The instruction JUMPI can be followed by CALLVALUE and CALL instructions, exposing possible vulnerabilities to potential attacks.

To capture the semantic feature of contract opcodes, 2-gram features are extracted from the simplified opcodes. Consider the Ethereum smart contracts. Each contract can be represented by 2-gram features with 2,401 dimensions, accounting for abundant opcode features. The feature forms of the opcodes are shown in Table V.

TABLE V: Opcode feature form.

Initial opcode	Simplified opcode	2-gram opcode
PUSH1 0x3	PUSH	(PUSH, SSTLOGIC_OPE)
SSTORE	SSTLOGIC_OPE	(SSTLOGIC_OPE,
CALLVALUE	CALLVALUE	CALLVALUE)
DUP1	DUP	(CALLVALUE ,DUP)
ISZERO	ISZERO	(DUP, ISZERO)
PUSH2 0x1f	PUSH	(ISZERO, PUSH)
JUMPI	JUMPI	(PUSH, JUMPI)

B. Source Feature Extraction

The Solidity programming language contains a wealth of semantic relationships that show the importance of keywords and various user identifiers in the source code of a contract. Simplifying operations, such as removing operands, may cause some information losses of a user when the source codes are converted to opcodes. For this reason, the characteristics of source codes are also analyzed and the token features are extracted to retain contract information in our method.

In the source codes of smart contracts, statements are made up of individual token segments. When processing source codes, the first step is to split contiguous characters into independent token segments. When extracting the source code features, we apply the pre-trained model CodeBERT [28]. CodeBERT was developed with a transformer-based neural architecture, which can handle both programming language (PL) and natural language (NL) and capture semantic connections between NL and PL. It uses both bimodal data of NL-PL pairs and unimodel data to improve the tokens for model training. A potential alternative to CodeBERT is lexical analysis [39], which performs natural language processing on smart contract source codes and builds a lexical analyzer for the Solidity language. However, the classification of tokens in the lexical analysis may not be complete. Particularly, user-defined identifiers, strings, and other features may not be obtained. Instead, simply classified word attributes are provided. For this reason, CodeBERT is applied to extract full source features. Algorithm 1 summarizes the token feature extraction based on CodeBERT.

The number of codes in different smart contracts can differ significantly, ranging from dozens of lines to thousands of lines. It can be challenging to count all token features, and this process can also result in data redundancy. However, as with traditional programs, smart contract codes can exhibit similarities in coding [40]. In other words, different contracts bear similar features, which can be exploited to substantially

Algorithm 1 CodeBERT source code feature extraction

Input: A smart contract source code file **TC**
Onput: A smart contract source code feature file **CSV**
1: Read smart contract source code files in the dataset **TC**;
2: Perform data cleaning, e.g., removing comment lines and comments;
3: Remove null characters, convert tokens to token IDs, according to the convert_tokens_to_ids function of the AutoTokenize module in CodeBERT;
4: Count the token IDs that exist in the dataset;
5: Count the occurences of each token in every contract, and record no-show tokens;
6: Write all tokens, including no-show tokens, and the numbers of their occurences in every contract into the source code feature file **CSV**;
7: **return CSV**;

reduce the number of token features. Moreover, when writing smart contract codes, developers often annotate the codes with comments. Therefore, data cleaning is needed to remove unnecessary fields, including comments and annotations, before extracting the source code features of smart contracts.

Using CodeBERT to convert all token segments in contract source codes and count the occurrences of each token ID as the source code feature of smart contracts can obtain the source code information of smart contracts. Moreover, the features in the contracts often overlap to a great extent due to the similarity of smart contract codes. When detecting a new contract, the token segments collected from a large amount of data can reflect new contract features. This token feature extraction method provides robustness and is applicable to the vulnerability detection of unknown contracts.

C. Construction of Fused Feature Matrix

Next, the opcode features and the source code features are fused by splicing the number of features extracted. We construct a feature vector as the input to machine learning models for smart contract vulnerability detection. Each feature vector corresponds to a smart contract. The feature dimension of the contract is equal to the number of columns.

For the i-th smart contract, the feature vector, denoted by $\mathbf{f}_i$, is given by

$$\mathbf{f}_i = [b_{i,1} \cdots b_{i,N_F}], \tag{1}$$

where N_F is the total number of features per smart contract; and $b_{i,j}$ specifies the number of occurrences of the j-th feature in the i-th smart contract, which can be either an opcode feature or a source code feature, and is given by

$$b_{i,j} = c_{i,j}/c_{i,all}, \tag{2}$$

where $c_{i,j}$ is the number of occurrences of the j-th feature in the i-th smart contract, and $c_{i,all}$ is the total number of occurrences of all features in the i-th smart contract. Take Ethereum smart contracts for an example. A 3,747-dimensional feature vector can be obtained for a smart contract after fusing the source codes and opcodes, including 1,346 dimensions for source code features and 2,401 for opcode features.

We can further compress the features to suppress some less important and less frequent features, e.g., by setting a threshold for the minimum number of occurrences of an opcode feature. For example, by setting the threshold to 5, the dimension of the opcode features is reduced from 2,401 to 570 in the Ethereum smart contracts. As a result, the dimension of the overall feature vectors is reduced from 3,747 to 1,916 (=1,346+570).

D. Detection models

Seven classification machine learning models are built for vulnerability detection:

• KNN: The core idea is to determine the category of a sample, according to the category of the nearest one or several samples, e.g., through k-nearest neighbor queries. We set the number of neighbors to $k = 5$.

• SVM: When data samples are linearly separable, the most suitable classification hyperplanes are found by maximizing the gap between classes. We use a liner kernel function, and train the SVM model using the default hinge loss function. The hyperparameters, i.e., the regularization parameter (=2) and gamma value (=10), are obtained through extensive tests.

• CNN: Different from traditional neural networks, a CNN contains a feature extractor consisting of convolutional layers and pooling layers. It avoids explicit feature extraction and learns features directly from training data. Our CNN model contains a convolutional layer with a kernel size of three, a pooling layer with a pool size of two, and two fully connected layers. We adopt the ReLU activation function, Adam optimizer, and a "binary_crossentropy" loss function.

• RF: An RF model constructs multiple independent decision trees. We set the number of trees to 100. The number of samples required to split an internal node is set to be no smaller than two for the RF model training. RF is easy to implement, computationally efficient, and effective in classification tasks. The RF model is trained using stochastic gradient descent (SGD).

• MLP: We build an MLP model comprising a hidden layer with 100 hidden units, and adopt Linear activation functions and a cross-entropy loss function.

• RNN: This model deals with time-series problems. Its output depends on the current and the previous time slices, thereby effectively mining semantic information in data. Our RNN model contains three hidden layers. The size of feature dimensions in the hidden layer tensor is 64. We adopt a nonlinear activation function $\tanh()$, Adam optimizer, and cross-entropy loss function.

• LSTM: A gate mechanism is introduced to control the circulation and loss of features, helping a neural network better learn semantic information about long sentences. LSTM solves the long-term dependence problem of RNN. We set the dimension of the output layer to 50, and adopt the $\tanh()$ activation function, Adam optimizer, and mean absolute error (MAE) loss function.

VI. Experiment and Performance Evaluation

In this section, the proposed method is experimentally evaluated on the Ethereum smart contract dataset. Comparison

studies are carried out between the method and the status quo.

A. Experimental Settings

The experimental settings include the construction of the dataset, performance indicators, and benchmark algorithms.

1) Dataset Construction: To verify the proposed vulnerability detection method, a reliably labeled dataset is needed. To efficiently label the dataset, existing vulnerability detection tools are considered, including Oyente [7], Slither [9], and DefectChecker [10]. Oyente is based on symbolic execution and has been widely applied to smart contract security [31]. Slither is a static analysis framework for smart contracts. It uses abstract syntax trees from source code compilation as input and can detect more than 70 vulnerabilities and draw topology diagrams of contract inheritance and invocation relationships. DefectChecker is another effective and more recent tool, which can detect eight contract defects from smart contract bytecodes [10]. We test Oyente, Slither, and DefectChecker on an annotated smart contract dataset provided in [17]. As shown in Fig. 9, the detection accuracy is 67.2%, 68%, and 55.3% under DefectChecker, Slither, and Oyente, respectively.

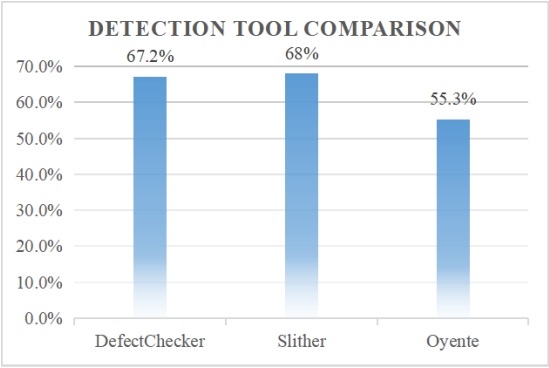


Fig. 9: The comparison of different detection tools.

The Ethereum dataset[1] published by Messi-Q on GitHub is used to create the dataset for assessment of the proposed vulnerability detection method. We selected different solc compilers for batch processing, according to the smart contract versions. Any contracts that failed the detection of the three tools or did not have version numbers were removed, resulting in 10,266 normally compiled contract files. The remaining contracts were classified according to the types of vulnerabilities, and written into CSV files. Since different tools detect different types of vulnerabilities, the detection results of the three tools need to be merged by: 1) obtaining the public contract list for the detection results of Slither, Oyente, and DefectChecker, and 2) checking the test results of public contracts and discarding the contracts with inconsistent test results regarding the same vulnerability. At last, we inspected the results closely and corrected false positives to ensure the correctness of the labels.

The features of different contracts exhibit similarities. Take Fig. 10 as an example. (block.timestamp).add performs the add operation on the block timestamp, which helps with the detection of a possible timestamp vulnerability in the contract. It is observed that there are 7.2% of smart contracts with these particular statements in the considered Ethereum dataset. The results of feature extraction show that for source codes of 20 lines to nearly a thousand lines, the features of a smart contract can be represented by 1,346 token IDs.

```
function airdrop() private view returns(bool){
  uint256 seed = uint256(keccak256(abi.encodePacked(
    (block.timestamp).add
    (block.difficulty).add
    ((uint256(keccak256(abi.encodePacked(block.coinbase))))/(now)).add
    (block.gaslimit).add
    ((uint256(keccak256(abi.encodePacked(msg.sender))))/(now)).
    add(block.number))));
  if((seed - ((seed / 1000) * 1000)) < airDropTracker_)
    return(true);
  else
    return(false); }
```

Fig. 10: An example of timestamp dependence vulnerability.

TABLE VI: Smart contract dataset vulnerability information.

Vulnerability type	Contract number		
	Vulnerability contract	Normal contract	Total number
Reenterancy	74	10192	10266
Timestamp	188	10078	10266
Order	570	9696	10266

Table VI shows the information of the resulting smart contract dataset that was used to evaluate the proposed smart contract vulnerability detection method[2]. The dataset is divided between a training set (80%) and a test set (20%), and utilized to train and test the proposed method.

2) Performance Indicators: The accuracy, precision, recall, F1-score, macro-F1 and mAP serve as the evaluation indexes of the models considered. Accuracy represents the ratio of samples correctly detected by a model in the total number of predictions. Precision assesses the ratio of samples whose true and predicted values are both positive in all predicted positive classes. Recall represents the ratio of samples whose true and predicted values are both positive in the true class. F1-score and macro-F1 are used to measure the performance of the classifier. F1-score is a metric for evaluating binary classifiers, while macro-F1 evaluates the recognition performance of a model for multiple vulnerabilities and it assigns the same weight to each vulnerability during evaluation. mAP evaluates the average accuracy of a model regarding different types of vulnerabilities, where AP gives the mean accuracy of each vulnerability type. Specifically, mAP is defined as

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i. \tag{3}$$

The Receiver Operating Characteristic (ROC) curve is also used to evaluate the proposed vulnerability detection method, where the x-axis indicates the false alarm rate (FPR) and the y-axis indicates the true positive rate (TPR). The area under the ROC curve (AUC) reflects the classification performance of a machine learning model. The closer the ROC curve is to the

[1]https://github.com/Messi-Q/Smart-Contract-Dataset

[2]The dataset can be found at https://github.com/Sheep-L/boke.

upper left, the stronger vulnerability identification capability the model has.

B. Feature Extraction and Simplification

Some of the features do not occur often in the feature vectors of smart contracts and have never occurred in any contract labeled with vulnerabilities. We can remove the features and hence shorten the feature vectors. By inputting the shortened feature vector, not only the efficiency of the machine learning models is improved, but also the accuracy is increased. For example, a 2,401-dimensional opcode feature vector can be reduced to 570-dimensional in the Ethereum dataset. As a result, the dimension of a feature vector is reduced from originally 3,747 to 1,916, as described in Section V-C.

In order to verify the correctness of this method, the datasets before and after the dimension reduction are trained separately and the test results are compared. As shown in Table VII, removing less important features has a marginal impact on the results of vulnerability classification and detection, while the detection efficiency is improved substantially.

In terms of source code extraction, apart from CodeBERT, the lexical analyzer is a potential alternative to CodeBERT and is tested and compared with CodeBERT. This is done by replacing CodeBERT with the lexical analyzer in the proposed vulnerability detection method. The source code features obtained by the lexical analysis or CodeBERT, are fused with the opcode features. Then, the feature vectors are generated and input into the machine learning models. The classification results are shown in Table VIII. Clearly, CodeBERT has better macro-F1 than lexical analysis. Using CodeBERT to convert all token segments in the source codes and count them, can offer richer source code feature information than lexical analysis. This validates our selection of CodeBERT for extracting source code features for further research.

C. Experimental Design and Detection Results

We proceed to compare the proposed fused opcode and source code features with the separate opcode features or source code features, by inputting them separately into the machine learning models and assessing the detection accuracies of the models. CodeBERT is used for source code feature extraction. The 1,916-dimensional feature vectors (consisting of 1,346 source code features and 570 opcode features per contract) are input into the machine learning model.

Table IX shows the detection results of the seven machine learning models regarding reentrancy, timestamp dependence, and transaction-ordering dependence, where three different forms of the feature are considered: opcode feature, source code feature, and fused feature. It is observed that fusing the source code and opcode features improves the classification results (i.e., macro-F1 and mAP) of SVM, CNN, RF, MLP, RNN and LSTM, compared to only using the source code features or opcode features. It is also observed that RF consistently outperforms the other models. Specifically, macro-F1 is 87.3% and mAP is 92% under RF, much better than the rest of the models. The corresponding ROC curve of the RF model is shown in Fig. 11. The conclusion drawn is that the RF model with the input of fused features provides an effective smart contract vulnerability detection tool.

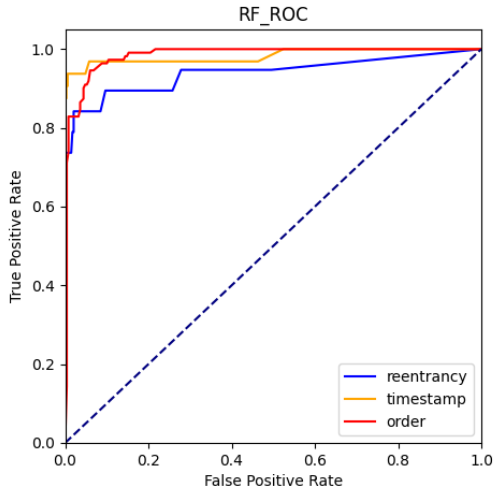


Fig. 11: The ROC plot for the RF model with the input of fused opcode and source code features.

It is interesting to note that the fusion of different features does not benefit the classification result of the KNN model, while benefiting all other models. The reason is that by fusing the different features into a vector, the dimension of the feature vector increases substantially. However, the KNN classifies the samples based on the distance between the feature vectors. As a result, the increased dimension of the feature vectors leads to increased distances among different contracts and increased dispersion of the contracts in the high-dimensional feature space, thus resulting in a failed detection of the KNN model.

Table IX also shows that the detection using source code features is better than that using opcode features. The CNN model benefits the most by replacing opcode features with source code features, with an improvement of 11% in macro-F1 and an improvement of 10% in mAP.

Tables X, XI, and XIV summarize the detection results of the considered models using opcode features, source code features, and fused features. It is observed that by adding the token features extracted from the source codes using CodeBERT, the classification models, i.e., KNN, SVM, CNN, MLP, RNN and LSTM, are considerably improved in accuracy, precision, recall, and F1-score for detecting each of the three considered vulnerabilities. Among them, the SVM model can benefit the most from the integration of the source code features into the opcode features using the SVM model, as also evident from the ROC curves of the SVM model shown in Fig. 12. It is further confirmed that all performance indicators of the RF model improve for the detection of reentrancy vulnerabilities, timestamp dependence vulnerabilities, and transaction-ordering dependence vulnerabilities. By further assessing macro-F1 and mAP, we can conclude that RF offers the best vulnerability detection.

The detection time of the proposed vulnerability detection method is shown in Table XIII, where the RF model with the

TABLE VII: Comparison of detection results before and after opcode redundancy suppression.

Detection model	F1-score			macro-F1	F1-score(de-redundancy)			macro-F1
	reentrancy	timestamp	order		reentrancy	timestamp	order	
KNN	0.59	0.56	0.67	0.606	0.57	0.55	0.70	0.606
SVM	0.69	0.67	0.74	0.70	0.68	0.67	0.71	0.686
CNN	0.70	0.62	0.74	0.686	0.67	0.71	0.76	0.713
RF	0.81	0.83	0.85	0.83	0.81	0.90	0.85	0.853
MLP	0.70	0.66	0.70	0.686	0.66	0.70	0.74	0.736

TABLE VIII: Comparison of detection results of different feature representation methods.

Feature model	Contract number				
	KNN	SVM	CNN	RF	MLP
Opcode + Lexical Analysis	0.650	0.756	0.760	0.876	0.780
Opcode+CodeBERT	0.713	0.826	0.863	0.873	0.826

TABLE IX: Vulnerability detection results of different types of features.

Feature type	Opcode feature		Source code feature		Fused feature	
Model	macro-F1	mAP	macro-F1	mAP	macro-F1	mAP
KNN	0.606	0.61	0.79	0.76	0.713	0.693
SVM	0.686	0.67	0.75	0.73	0.826	0.853
CNN	0.713	0.713	0.823	0.813	0.863	0.876
RF	0.883	0.933	0.85	0.916	**0.933**	**0.967**
MLP	0.736	0.70	0.803	0.803	0.823	0.85
RNN	0.763	0.77	0.82	0.793	0.853	0.83
LSTM	0.816	0.876	0.807	0.863	0.848	0.942

input of fused features is used to detect the three considered vulnerabilities. By contrast, commonly-used vulnerability detection tools, such as Oyente [7], and Securify [20], execute on the basis of symbols. For example, Oyente needs to traverse all executable paths, which are often time-consuming. It was reported that Oyente requires about 18.48 seconds [41] and Securify requires 18 seconds [12] to detect a contract. A more recent technique, named Smartcheck, takes about 10 seconds to detect a contract [42]. Moreover, the formal validation tool, VaaS, and the fuzzy detection tool, ContractFuzzer, also incur longer average detection times of 159.4 seconds [6] and 352.2 seconds [8] per contract, respectively. The proposed method is superior to the existing techniques in detection efficiency.

D. Comparison with Status Quo

We compare the proposed method with two recent detection methods, namely, Smartcheck [43], Contractward [12], and GCN [17]. Slither is also considered since it is a popular and effective vulnerability detection tool [44]. As discussed in Section II, ContractWard analyzes and constructs contract features from opcodes and can detect the three vulnerabilities studied. Smartcheck, an extensible static analysis tool, translates Solidity source codes into an XML-based intermediate representation and checks the representation against XPath patterns [43]. GCN classifies the three considered types of smart contract vulnerabilities by combining GNN and expert knowledge [17]. Table XIV shows that the proposed vulnerability detection model based on the RF model and fused opcode and source code features achieves substantially higher accuracy and F1-score than the three state-of-the-art

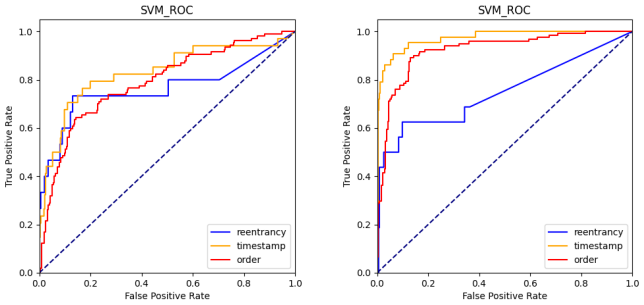


Fig. 12: Comparison of the SVM model before and after adding the source code token features to the opcode features.

approaches. As also noticed, Slither does not perform well, even though the test dataset was annotated using Slither, Oyente and DefectChecker (followed by manual inspection and correction), as described in Section VI-A1. This is because Slither fails to detect many false negatives that were manually corrected in the test set.

VII. Conclusion and Future Work

In this paper, we have proposed an automated vulnerability detection method that fuses opcode features and source code features to facilitate machine learning-based detection of smart contract vulnerabilities. Our model has been trained to efficiently detect widespread smart contract vulnerabilities, including reentrancy vulnerability, timestamp dependence vulnerability, and transaction-ordering dependence vulnerability. Tested on a real-world dataset, the macro-F1 and mAP of the method regarding the three vulnerabilities are as high as 93.3% and 96.7%, respectively. The experiments have also shown that the proposed method has high efficiency and substantial gains over existing techniques that focused on single-level feature representation and model optimization.

Like many machine learning-based algorithms, the accuracy and efficiency of the proposed vulnerability detection method depend heavily on the acquisition and quality (e.g., representativeness) of the data used for model training. Existing studies typically require both vulnerability detection tools, such as Slither, and manual inspection to annotate their training data. Given the typically poor accuracy of the existing tools, manual inspection is indispensable and crucial. However, manual inspection is laborious, slow, and susceptible to human errors. Another critical challenge arises from the generalization of the proposed method to other possible vulnerabilities, particularly those never seen or known. In this case, generative adversarial networks can be potentially used to produce adversarial samples, and adversarial training can be potentially

TABLE X: Opcode feature vulnerability detection results.

Classification	reentrancy				timestamp				order				macro-F1	mAP
Model	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc		
KNN	0.56	0.70	0.57	0.85	0.57	0.67	0.55	0.71	0.70	0.73	0.70	0.72	0.606	0.61
SVM	0.65	0.71	0.68	0.94	0.65	0.80	0.67	0.82	0.71	0.74	0.71	0.74	0.686	0.67
CNN	0.70	0.68	0.69	0.95	0.69	0.72	0.70	0.89	0.75	0.76	0.76	0.79	0.713	0.713
RF	0.93	0.79	0.85	0.98	0.97	0.85	0.90	0.96	0.90	0.90	0.90	0.91	0.883	0.933
MLP	0.69	0.69	0.69	0.95	0.69	0.74	0.71	0.89	0.72	0.73	0.72	0.76	0.736	0.70
RNN	0.74	0.78	0.76	0.95	0.82	0.75	0.78	0.92	0.75	0.78	0.75	0.77	0.763	0.77
LSTM	0.92	0.70	0.77	0.97	0.84	0.78	0.81	0.94	0.87	0.87	0.87	0.89	0.816	0.876

TABLE XI: Source code feature vulnerability detection results.

Classification	reentrancy				timestamp				order				macro-F1	mAP
Model	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc		
KNN	0.65	0.75	0.69	0.93	0.79	0.89	0.83	0.91	0.84	0.86	0.85	0.86	0.79	0.76
SVM	0.61	0.71	0.63	0.91	0.76	0.86	0.80	0.90	0.82	0.83	0.82	0.84	0.75	0.73
CNN	0.71	0.74	0.72	0.95	0.87	0.90	0.89	0.95	0.86	0.86	0.86	0.86	0.823	0.813
RF	0.92	0.72	0.78	0.97	0.94	0.86	0.89	0.96	0.89	0.88	0.88	0.90	0.85	0.916
MLP	0.74	0.71	0.72	0.96	0.83	0.88	0.85	0.93	0.84	0.85	0.84	0.86	0.803	0.803
RNN	0.74	0.87	0.79	0.95	0.78	0.82	0.80	0.91	0.86	0.88	0.87	0.89	0.82	0.793
LSTM	0.90	0.67	0.73	0.97	0.82	0.82	0.82	0.94	0.87	0.87	0.87	0.89	0.807	0.863

TABLE XII: Fused feature vulnerability detection results.

Classification	reentrancy				timestamp				order				macro-F1	mAP
Model	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc	Preci	Recall	F1	Acc		
KNN	0.63	0.69	0.65	0.925	0.72	0.84	0.76	0.87	0.73	0.75	0.73	0.75	0.713	0.693
SVM	0.82	0.68	0.73	0.96	0.91	0.91	0.91	0.96	0.83	0.84	0.84	0.85	0.826	0.853
CNN	0.88	0.81	0.84	0.97	0.87	0.87	0.87	0.94	0.88	0.88	0.88	0.89	0.863	0.876
RF	0.99	0.87	0.92	0.98	0.99	0.94	0.96	0.98	0.93	0.92	0.92	0.94	0.933	0.967
MLP	0.81	0.71	0.75	0.96	0.88	0.85	0.86	0.94	0.86	0.86	0.86	0.88	0.823	0.85
RNN	0.76	0.88	0.81	0.96	0.86	0.92	0.89	0.95	0.87	0.86	0.86	0.89	0.853	0.83
LSTM	0.88	0.70	0.75	0.965	0.91	0.915	0.91	0.965	0.91	0.87	0.885	0.895	0.848	0.942

TABLE XIII: detection time.

	Our method				average speed
	preprocessing	source feature	opcode feature	prediction	
time(s)	0.025	0.024	0.035	0.015	0.99

applied to improve the generalization ability of our model. In our future work, we will collect more vulnerability data of different vulnerability types, generate adversarial samples for unknown vulnerabilities, and train the proposed machine learning-based method for its generalization ability to other types of vulnerabilities, including those unseen or unknown.

VIII. Acknowledgment

This work was supported in part by the Fundamental Research Funds for the Central Universities of China under Grant 2020JBZ104, the Beijing Natural Science Foundation under Grant No.4212008, Guangxi Key Laboratory of Cryptography and Information Security under Grant No. GCIS201915, the Open Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China under Grant No.ISECCA-202101 and in part by the National Natural Science Foundation of China, under Grant 62272031, U21A20463 and U22B2027.

References

[1] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.

[2] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[3] M. S. Chishti, F. Sufyan, and A. Banerjee, "Decentralized on-chain data access via smart contracts in ethereum blockchain," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 174–187, 2021.

[4] M. Del Castillo, "The dao attacked: Code issue leads to $60 million ether theft," *Saatavissa (viitattu 13.2. 2017)*, vol. 3, 2016.

[5] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[6] T. Feng, X. Yu, Y. Chai, and Y. Liu, "Smart contract model for complex reality transaction," *International Journal of Crowd Science*, vol. 3, no. 2, pp. 184–197, 2019.

[7] 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.

[8] 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.

[9] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[10] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2021.

[11] 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.

[12] 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.

TABLE XIV: Comparison between the proposed method and the state-of-the-art vulnerability detection methods.

Vulnerability	reentrancy			timestamp			order		
detection method	Preci	Recall	F1-score	Preci	Recall	F1-score	Preci	Recall	F1-score
Our method	0.99	0.87	0.92	0.99	0.94	0.96	0.93	0.92	0.92
ContractWard [12]	0.99	0.79	0.86	0.97	0.86	0.91	0.91	0.90	0.90
Slither [9]	0.70	0.75	0.68	0.69	0.72	0.65	-	-	-
SmartCheck [43]	0.71	0.79	0.75	0.73	0.80	0.74	-	-	-
CGE [17]	0.85	0.87	0.86	0.87	0.88	0.87	-	-	-

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[14] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, vol. 22, no. 5, p. 1829, 2022.

[15] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, 2021, pp. 47–59.

[16] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: a pdg-based code clone detector with approximate graph matching," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 931–942.

[17] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[18] B. ZHAO, C. SHANGGUAN, X. PENG, Y. AN, J. TONG, and Y. Anqi, "Smart contract bytecode vulnerability detection method based on semantic-aware graph neural network," *Engineering Science and Technology*, pp. 49–55, 2022.

[19] L. Duan, Y. Sun, W. Ni, W. Ding, J. Liu, and W. Wang, "Attacks against cross-chain systems and defense approaches: A contemporary survey," *IEEE/CAA Journal of Automatica Sinica*, 2023.

[20] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[21] K. Taunk, S. De, S. Verma, and A. Swetapadma, "A brief review of nearest neighbor algorithm for learning and classification," in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*. IEEE, 2019, pp. 1255–1260.

[22] D. A. Pisner and D. M. Schnyer, "Support vector machine," in *Machine learning*. Elsevier, 2020, pp. 101–121.

[23] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[24] M. Belgiu and L. Drăguţ, "Random forest in remote sensing: A review of applications and future directions," *ISPRS journal of photogrammetry and remote sensing*, vol. 114, pp. 24–31, 2016.

[25] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit *et al.*, "Mlp-mixer: An all-mlp architecture for vision," *Advances in neural information processing systems*, vol. 34, pp. 24 261–24 272, 2021.

[26] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, no. 3. Makuhari, 2010, pp. 1045–1048.

[27] G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, pp. 5929–5955, 2020.

[28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[29] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[30] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[31] J. Ma, Z. Zhao, X. Yi, J. Chen, L. Hong, and E. H. Chi, "Modeling task relationships in multi-task learning with multi-gate mixture-of-experts," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1930–1939.

[32] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[33] 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.

[34] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 378–389.

[35] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[36] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.

[37] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

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

[39] A. Rajput, "Natural language processing, sentiment analysis, and clinical analytics," in *Innovation in health informatics*. Elsevier, 2020, pp. 79–97.

[40] E. Fedorenko, A. Ivanova, R. Dhamala, and M. U. Bers, "The language of programming: a cognitive perspective," *Trends in cognitive sciences*, vol. 23, no. 7, pp. 525–528, 2019.

[41] F. Hill, K. Cho, and A. Korhonen, "Learning distributed representations of sentences from unlabelled data," *arXiv preprint arXiv:1602.03483*, 2016.

[42] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2021.

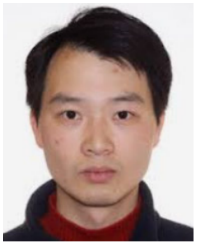
[43] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[44] Z. Pan, T. Hu, C. Qian, and B. Li, "Redefender: A tool for detecting reentrancy vulnerabilities in smart contracts effectively," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 915–925.

Li Duan received her Ph.D degree in Computer Science and Technology from Beijing University of Posts and Telecommunications, Beijing, China, in 2016. She is currently an Assistant Professor with the School of Computer and Information Technology, Beijing Jiaotong University. She was a research fellow of Nanyang Technological University and University of Science and Technology Beijing. Her research interests are services computing and internet of thing, Data security and privacy protection, and Blockchain security and applications. She received the National Natural Science Foundation of China, the Postdoctoral Fund, and the Basic Scientific Research Project.

Liu Yang received her B.S. degree from Henan Normal University of China in 2022. She is currently pursuing the M.S. degree at the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. Her research interests are network security, blockchain security and smart contract defect detection.

Chunhong Liu received the PhD degree in computer science and technology from Beijing University of Posts and Telecommunications (BUPT) in 2018. She is currently an associate professor in Department of Computer and Information Engineering at Henan Normal University, China. Her major research interests include intelligent service computing, machine learning, Data security and Blockchain security and applications.



Wei Ni (M'09-SM'15) received the B.E. and Ph.D. degrees in Communication Science and Engineering from Fudan University, Shanghai, China, in 2000 and 2005, respectively. Currently, he is a Principal Research Scientist at CSIRO, Sydney, Australia. He is also a Conjoint Professor at the University of New South Wales, an Adjunct Professor at the University of Technology Sydney, and an Honorary Professor at Macquarie University. He was a Postdoctoral Research Fellow at Shanghai Jiaotong University from 2005 to 2008; Deputy Project Manager at the Bell Labs, Alcatel/Alcatel-Lucent from 2005 to 2008; and Senior Researcher at Devices R&D, Nokia from 2008 to 2009. He has authored seven book chapters, more than 280 journal papers, more than 100 conference papers, 26 patents, and ten standard proposals accepted by IEEE. His research interests include machine learning, online learning, stochastic optimization, and their applications to the security, integrity and efficiency of network systems.

Dr. Ni serves as an Editor for IEEE Transactions on Vehicular Technology since 2022, IEEE Transactions on Wireless Communications since 2018, and Cambridge Press New Research Directions: Cyber-Physical Systems since 2022. He served as the Chair of the IEEE Vehicular Technology Society New South Wales Chapter from 2020 to 2022, the Secretary and then the Vice-Chair of the Chapter from 2015 to 2019, Track Chair for VTC-Spring 2017, Track Co-chair for IEEE VTC-Spring 2016, Publication Chair for BodyNet 2015, and Student Travel Grant Chair for WPMC 2014.

Wei Wang is currently a full professor and chairs the Department of Information Security, Beijing Jiaotong University, China. He earned his Ph.D. degree in control science and engineering under the supervision of Prof. Xiaohong Guan from Xi'an Jiaotong University, in 2006. He was a postdoctoral researcher in University of Trento, Italy, during 2005-2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, during 2007-2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, during 2009-2011. He visited INRIA, ETH, NTNU, CNR, New York University Polytechnic, and KAUST. He has authored or co-authored over 100 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer and network security. He was invited to serve on the Editorial Board of Computers & Security Journal.