Block-gram: Mining Knowledgeable Features for Smart Contract Vulnerability Detection

Tao Li[1,3,4], Haolong Wang[2], Yaozheng Fang[2], Zhaolong Jian[2], Zichun Wang[2], and Xueshuo Xie[⋆2,3,4]

[1] Tianjin Key Laboratory of Network and Data Security Technology, Tianjin, China
[2] College of Computer Science, Nankai University, Tianjin, China
[3] Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province
[4] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

Abstract. Effective vulnerability detection of large-scale smart contracts is critical because smart contract attacks frequently bring about tremendous economic loss. However, code analysis requiring traversal paths and learning methods requiring many features training is too time-consuming to detect large-scale on-chain contracts. This paper focuses on improving detection efficiency by reducing the dimension of the features, combined with expert knowledge. We propose a feature extraction method *Block-gram* to form low-dimensional knowledgeable features from the bytecode. We first separate the metadata and convert the runtime code to opcode sequence, dividing the opcode sequence into segments according to some instructions (*jump*, etc.). Then, we mine extensible *Block-gram* features for learning-based model training, consisting of 4-dimensional block features and 8-dimensional attribute features. We evaluate these knowledge-based features using seven state-of-the-art learning algorithms to show that the average detection latency speeds up 25 to 650 times, compared with the features extracted by *N-gram*.

Keywords: Smart Contract, Bytecode, Opcode, Knowledgeable Features, Vulnerability Detection

1 Introduction

Smart contracts are widely deployed on blockchain to implement complex transactions, such as decentralized applications on Ethereum [1]. As of August 9, 2022, the number of smart contracts exceeded 51.1 million on Ethereum[1]. These smart contracts are written in a domain specific language (e.g., *Solidity*), compiled into bytecodes with a compiler, executed many times as opcodes in EVM after being deployed on-chain by the consensus mechanism [2]. Due to running on distributed nodes of blockchain, once vulnerabilities are found, they are difficult

⋆ Corresponding author: xueshuoxie@nankai.edu.cn

[1] https://explore.duneanalytics.com/

to upgrade and repair [3,4]. For example, hackers exploited a re-entrancy vulnerability in the DAO contract to steal 3.6 million ETH[2]. According to SlowMist Hacked, a huge economic loss of more than $10 billion has been caused due to the security issues of smart contracts[3]. With the rapid increase in the number of smart contracts, an efficient smart contract vulnerability detection method is particularly important for the development of blockchain [5].

Nowadays, the mainstream vulnerability detection methods on smart contracts include code analysis and machine learning. For code analysis, we can analyze the types or causes of vulnerabilities with expert knowledge through formal verification, symbolic execution, fuzz testing, etc. [6–10]. But these methods need to traverse more paths of code or complexity mathematical proofs, the detection is time-consuming and labor-intensive. For machine learning, they primarily capture code features by training machine learning models to infer whether it is vulnerable. Some smart contract vulnerability detection algorithms are based on the combination of text information and neural network [5, 11–13] or based on the combination of smart contract graph information and neural network [14,15]. But all of them need a large feature space for training, and the dimensional of features will influence the model performance and detection latency.

In this paper, we focus on detection efficiency as the quick detection requirement goal of large-scale smart contracts and face the following challenges: (1) how to improve detection model performance combined with attribute features of vulnerabilities through expert knowledge; (2) how to reduce the dimension of feature space for machine learning model training? Optimize the detection latency without influencing the model performance. We address the above challenges through two key designs. To tackle the first challenge, we preprocess the bytecode to opcode sequences according to the disassembling rules of Ethereum and divide the opcode sequences into flow graphs through some instructions (*jump*, etc.) for extracting 4-dimensional block features. For the second challenge, we mine other 8-dimensional attributr features from vulnerabilities analysis through expert knowledge, to construct extensible 12-dimensional *Block-gram* features. The *Block-gram* has lower dimensional features than the existing features with thousand dimensions by *N-gram*, and will significantly reduce the detection latency and not influence the model performance. We also evaluate the effectiveness of the *Block-gram* features on seven state-of-the-art machine learning algorithms. In summary, this paper makes the following contributions:

– We mine the *Block-gram* features from bytecode, including block features, and attribute features on smart contract vulnerabilities. The *Block-gram* features are extensible low-dimensional features and will significantly reduce the detection latency without influencing the model performance.
– We introduce expert knowledge when constructing opcode sequence flow graphs and extracting attribute features, and combine vulnerability analysis to improve the performance of the above low-dimensional features.

[2] http://www.coindesk.com/daoattacked-code-issue-leads-60-million-ether-theft, 2016.

[3] https://hacked.slowmist.io/en/

– We validate the efficiency of *Block-gram* features on seven state-of-the-art machine learning algorithms. The evaluation shows that the above low-dimensional features can flexibly support multiple detection algorithms and significantly reduce detection latency.

The rest of this paper is organized as follows. We briefly introduce some concepts about Ethereum virtual machine (EVM) and smart contracts and summarize the related work of smart contract security analysis in Section II. Section III provides a detailed discuss the features extraction method. Experiments and results are presented in Section IV, to discuss the effectiveness, efficiency of the extracted features. Section V concludes the paper.

2 Preliminaries

2.1 Ethereum Smart Contract

Ethereum Virtual Machine. For smart contract running, EVM provides 142 opcodes or bytecodes with 10 functions, such as stop and arithmetic operations, push operations, etc. [16] There are only 256 opcodes at maximum, but some instructions are not defined now, only for future expansion. EVM has a simple stack structure with a maximum stack size of 1024. Each opcode is allocated one byte (for example, *STOP* is *0x00*), pushes or pops a certain number of elements from the stack, and can obtain information about the execution environment, or interact with other blockchains smart contracts [16]. During execution, the bytecode is split into bytes (1 byte equals 2 hex characters). Bytes in region *0x60-0x7f* (*PUSH1-PUSH32*) are treated differently because they contain data that needs to be pushed into the stack. If the call count is over 1024, the call-stack attack may take place. EVM explains how to change the system state given a series of the above instructions and a small part of environmental data.

Smart Contract Compilation. As shown in Fig. 1, the developers write the source code in a high-level language (e.g., *Solidity*). The source codes are compiled into byte arrays encoded by hexadecimal digits with a compiler as bytecodes. [17] Then, the bytecodes are uploaded to EVM with an Ethereum client and can be translated into EVM instructions or opcodes. [18] After the source code of the contract is compiled into EVM bytecode and ABI, it can be deployed using the *Web3.js* interface. For contract deployment, it is essential to execute a transaction, which has no destination address but the data field is EVM bytecode. [19] When processing this transaction, the EVM executes the input data as code. The bytecode is divided into deployment code, runtime code, and metadata. After the contract deployment, EVM will store the runtime code and metadata on the blockchain, and then match their storage addresses to the contract account to complete the deployment of the contract. It would be easier to analyze smart contracts with bytecodes or opcodes, because: (1) bytecodes or opcodes are not had man-made variables that are defined in source codes; (2) bytecodes or opcodes are easy to collect from the public blockchain.

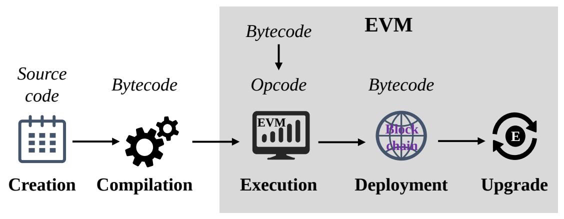

Fig. 1: Smart Contract Compilation Process.

2.2 Smart Contract Security Analysis

Code Analysis. As shown in Table 1, we conclude some state-of-the-art code analysis and learning methods for smart contract vulnerability detection [4]. Most of the traditional smart contract vulnerability detection methods are based on program code analysis and program path analysis, such as formal verification, symbolic execution, fuzz testing, intermediate representation, and so on. Hildenbrandt et al. [6] present KEVM based on formal verification and provide an executable formal specification for EVM's program language using the K framework. Luu et al. [8] use symbolic execution to implement Oyente which traverses smart contract execution paths based on control flow graphs to detect vulnerability. Jiang et al. [9] propose Contractfuzzer that sets up test cases and analyzes smart contract behavior logs to detect vulnerabilities based on fuzz testing. Albert et al. [10] implement Ethir based on intermediate representation and analyze the security properties of bytecode by converting Oyente's control flow graph into a rule-based representation. Due to the need to traverse most paths of the code, the detection is time-consuming and labor-intensive, and difficult to use for large-scale contract detection.

Learning method. We also investigated some neural network-based smart contract vulnerability detection tools. Yu et al. [11] present Deescvhunter that uses a novel notion of Vulnerability Candidate Slice to capture the key of reentrancy vulnerability and time dependence vulnerability. Wang et al. [5] propose ContractWard that can extract bigram features from smart contract opcodes and use multiple machine learning algorithms for vulnerability detection. Mi et al. [13] apply novel feature vector generation techniques from bytecode and metric learning-based deep neural network to detect vulnerability. Zhuang et al. [14] use DR-GCN to convert source code into contract graph and use graph convolutional neural network to build a vulnerability detection model. Based on DR-GCN, TMP considered the time sequence information in the contract graph and used the time sequence graph neural network to build a vulnerability detection model. Zeng et al. [15] use graph neural network and expert knowledge to build the control flow graph with attribute and input graph attribute features into graph neural networks to detect vulnerabilities. However, due to the lack of expert knowledge of the features, most learning methods use high-dimensional feature spaces for training, such as *N-gram* extracting thousands of dimensional features. Due to the high feature space dimension, the detection is still time-

Table 1: Comparisons among smart contract vulnerability detection methods.

Type	Name	Base Model	Detection Source	Platform
Code analysis	KEVM	Formal Verification	bytecode	EVM
	Oyente	Symbolic Execution	bytecode & ETH Condition	EVM
	Contractfuzzer	Fuzz Testing	EVM ABI & EVM Log	EVM
	Ethir	Intermediate Representation	bytecode	EVM
Learning methods	DR-GCN	GCN	source code	ETH & VNT
	TMP	TGNN	source code	ETH & VNT
	Deescvhunter	DNN	source code	ETH
	Eth2vec	DNN	bytecode	ETH
	Escort	DNN	bytecode	ETH
	Rechecker	BLSTM	sourcecode	ETH
	ContractWard	XGBoost	bytecode	ETH
	Vscl	DNN	bytecode	ETH
	EtherGIS	GNN	bytecode	ETH
	SafeSC	LSTM	opcode	ETH

consuming and may not be suitable for batch vulnerability detection. Therefore, combining the expert knowledge used in code analysis with the learning methods is precisely the problem that our work mainly solves.

3 Detailed Design

3.1 Overview

As shown in Fig.2, we present a detailed description of the features extraction method *Block-gram* using smart contract bytecode. We first extract the 4-dimensional bytecode block features from the opcode sequence flow graph according to the disassembling rules of Ethereum. Then, we divide the opcodes into eight categories and count their ratios as 8-dimensional attribute features through six vulnerabilities analysis by expert knowledge. Finally, we use these 12-dimensional *Block-gram* features by seven state-of-the-art machine learning algorithms to detect six vulnerabilities. The *Block-gram* features include in:

– **Rule-based bytecode block features.** We first collect the bytecode of the metadata header of various solidity versions. The metadata in the bytecode is separated from the runtime code by string matching, and convert the runtime code into opcode according to Ethereum disassembling rules. Then we divide the opcode sequence into blocks, determine the jump relationship between blocks through `JUMP` opcode, and generate the opcode sequence flow graph, denoted by the adjacency matrix. We extract the number of nodes, number of edges, maximum out-degree, and maximum in-degree from the adjacency matrix as 4-dimensional bytecode block features. These features can preserve the relationship between different bytecode blocks.
– **Attribute Features with Expert Knowledge.** We first divide the opcodes of Ethereum into eight categories, such as unary arithmetic opcodes, binary arithmetic opcodes, block opcodes, control-flow opcodes, environmental opcodes, system opcodes, stack opcodes, and invalid opcodes. Each type

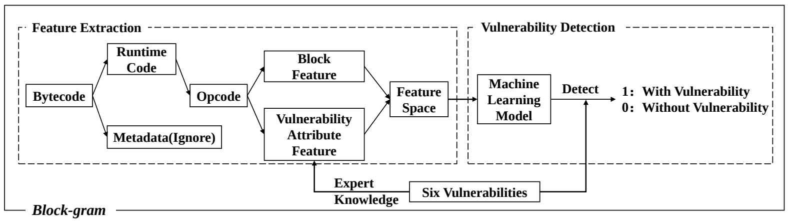

Fig. 2: The *Block-gram* feature extraction method overview.

of opcode corresponds to several smart contract vulnerabilities according to the vulnerability analysis with expert knowledge. We take the scale of each type of opcode and count their ratios as 8-dimensional extensional attribute features. These features can preserve the expert knowledge on vulnerability, and extend with the new vulnerabilities.

3.2 Rule-based Bytecode Block Features

The key to bytecode preprocessing are metadata separation and bytecode conversion. In the metadata separation module, we separate metadata and runtimecode according to metadata header(*e.g.,0x65 'bzzr0' 0x58 0x20 <32 bytes swarm hash>0x00 0x29*). Then we convert the runtimecode to the opcode according to Ethereum conversion rules. The bytecode of each non-PUSH class opcodes is converted into the corresponding opcode(for example, *STOP* is *0x00*, *ADD* is *0x01*). But for the PUSH class opcodes, we get the length of their parameters according to the type of the PUSH opcode, so as to convert the bytecode.

The existing processing methods of opcode sequence are divided into block-sequence method and natural language processing method. Block-sequence method divides the opcode sequence into blocks and edges. Natural language processing method uses *N-gram* method to process the opcode sequence. We choose block-sequence method to process opcode sequence and build adjacency matrix. First, we divide the opcode into blocks according to the jump class instruction, and then determine the edges between blocks according to the jump type at the end of each block. Each blocks has the following boundary.

- **Starting point:** `JUMPDEST`...
- **End point:** `JUMP, JUMP I, STOP, REVERT, RETURN, SELFDESTRUCT, INVALID`

In order to construct the edges between blocks, we divide edges into three categories according to the end point of blocks:

- `JUMP`**:** If there is `PUSHn` before `JUMP`, the parameter of `PUSHn` is the destination address of `JUMP`. If there is not `PUSHn` before `JUMP`, we using stack execution algorithm provided by EtherSlove [20] to calculate the destination address.

- `JUMP I`: `JUMP I` is a conditional jump. True edge's target is the parameter of the PUSH opcode; false edge's target is the offset of the following block. This means that if a basic block ends with JUMP I, then there will be two edges starting from this block.
- `REVERT, SELFDESTRUCT, RETURN, INVALID, STOP`: These opcodes mean the interruption of control flow, so they have no subsequent basic blocks.

When we get the blocks and edges, we can build an opcode sequence flow graph. Because the opcode sequence flow graph is a directed graph, we use the adjacency matrix to represent the opcode sequence flow graph and extract sequence features from the adjacency matrix. Then, we extract four-dimensional smart contract block features according to the adjacency matrix. These features represent the sequence attribute of the opcode.

- **Number of nodes.** The number of rows or columns of the adjacency matrix is the number of nodes. The number of nodes represents how many basic blocks are in the opcode sequence flow graph.
- **Number of edges.** The sum of the elements in the adjacency matrix is the number of edges. The number of edges represent how many basic edges are in the opcode sequence flow graph.
- **Maximum out-degree.** The maximum value of the sum of the elements in row i is the maximum out-degree. Out-degree is the sum of the times when a block of the opcode sequence flow graph is used as the starting point.
- **Maximum in-degree.** The maximum value of the sum of the elements in row j is the maximum in-degree. In-degree is the sum of the times when a block of the opcode sequence flow graph is used as the end point.

3.3 Attribute Features with Expert Knowledge

There are some common vulnerabilities in Ethereum, such as integer overflow vulnerabilities, integer underflow vulnerabilities, callstack depth attack vulnerability, transaction-ordering dependence vulnerability, timestamp dependency vulnerability, and re-entrancy vulnerability. We extract some attribute features by using expert knowledge to analyze six vulnerabilities of the Ethereum smart contracts. The cause of the above vulnerability is associated with some opcodes. We divide the opcodes of Ethereum into eight categories, such as unary arithmetic opcodes, binary arithmetic opcodes, block opcodes, control-flow opcodes, environmental opcodes, system opcodes, stack opcodes, and invalid opcodes. As shown in Table 2, each type of opcode corresponds to several smart contract vulnerabilities according to the vulnerability analysis with expert knowledge. We take the scale of each type of opcode and count their ratios as 8-dimensional extensional features. These features can preserve the expert knowledge on vulnerability, and extend with the new vulnerabilities. We define *count (opcode, i)* as the number of all opcodes of the smart contract i and *count (j, i)* as the number of opcodes corresponding to feature j in the smart contract i. For example, *count (1, 1)* is the number of unary arithmetic opcodes in smart contract 1.

Table 2: *Block-gram* Knowledgeable Features.

Feature		Opcode	Value	Knowledge
Block Feature	node	None	>0	feature of block
	edge		>0	
	maxout		>0	
	maxin		>0	
Attribute Feature	unary-arithmetic ratio	`ISZERO, NOT...`	(0,1)	feature of integer overflow
	binary-arithmetic ratio	`ADD, AND, SHA3...`	(0,1)	
	block ratio	`NUMBER, BLOCKHASH, COINBASE...`	(0,1)	feature of timestamp dependency
	control-flow ratio	`JUMP, JUMP I, JUMPDEST...`	(0,1)	feature of re-entrancy
	environment ratio	`CALLER, CALLDATASIZE...`	(0,1)	feature of TOD
	system ratio	`CALL, RETUREN, REVERT...`	(0,1)	feature of callstack depth attack
	stack ratio	`POP, PUSH, SWAP...`	(0,1)	
	invalid ratio	Others	(0,1)	feature of invalid opcodes

– **Unary and Binary arithmetic opcodes ratio:** In Ethereum, some opcodes are responsible for arithmetic operations, including unary, binary, and ternary arithmetic opcodes. Only these arithmetic opcodes cause integer overflow. In our research on smart contract, we find that almost no contract contain ternary arithmetic codes, so we only consider unary(*e.g.,* `ISZERO, NOT`) and binary(*e.g.,* `ADD, AND, SHA3`) arithmetic opcodes.
– **Block opcodes ratio:** In Ethereum, some opcodes are related to the block information. `BLOCKHASH` shows the hash value of the block, `COINBASE` is the address of the miner. At Ethereum system layer, block information is often used as a seed for generating random numbers. However, the block timestamp, number, and other information are often used by attackers. Block information opcodes are related to timestamp dependency vulnerabilities.
– **Control-flow opcodes ratio:** Some opcodes are used to change the control flow. `JUMP` is an unconditional jump that takes the top element of the stack as the destination address. `JUMP I` is a conditional jump that has the same destination address with `JUMP` if the top element of the stack is not zero; otherwise, EVM will execute the opcodes following `JUMP I`. The re-entrancy vulnerability stems from the attacker's cyclic call changing the control flow. Control flow opcodes are related to this vulnerability.
– **Environmental opcodes ratio:** Some opcodes are responsible for interacting with contracts and message calls and transactions. `ADDRESS` is the address of the currently executed contract. `ORIGIN` is the address of the initiator of the transaction. `CALLER` is the address of the caller of the message. Since the environmental opcodes can obtain transaction information, they are related to Transaction-Ordering Dependence vulnerabilities that also rely on transaction information.
– **System opcodes ratio:** The system opcodes are responsible for calling between smart contracts. `CALL` calls the function in other contracts. `RETURN` returns from the contract calls. `REVERT` reverts transaction and return data.
– **Stack opcodes ratio:** EVM is a stack machine, and all calculations are performed on a data area called the stack. Some opcodes deal with the elements of the stack. `POP` pops the top element of the stack and discards

it. `SWAP1` exchanges the top two members of the stack. They are stack opcodes.System opcodes and stack opcodes act on calls and stack operations, and these opcodes may cause callstack depth attack vulnerabilities.

– **Invalid opcodes ratio.** Invalid opcodes refer to the opcodes irrelevant to the six vulnerabilities detected.

3.4 Low-dimensional Knowledgeable Features

During feature extraction, to make the trained model suitable for all smart contracts, we first select 4-dimensional block features to highlight the relationship between different opcode blocks. The 4-dimensional block features are the number of nodes, the number of edges, the maximum out-degree, and the maximum in-degree of the control flow graph. These 4-dimensional features represent the complexity of the smart contract, emphasizing the role of the features of the smart contract itself in vulnerability detection. Then, we investigate the causes of vulnerabilities in smart contracts and mine 8-dimensional attribute features for opcodes associated with them. For example, unary-arithmetic and binary-arithmetic opcodes modify integers in Ethereum, and improper arithmetic operations can lead to integer overflow vulnerabilities; block information opcodes are closely related to timestamp vulnerabilities and block parameter dependency vulnerabilities; control flow opcodes are related to the re-entrancy vulnerability stems from the attacker's cyclic call changing the control flow. As shown in Table 2, we combined the 4-dimensional block features and 8-dimensional attribute features together for efficient vulnerability detection. When constructing the opcode sequence flow graph, we used Depth-First-Search(DFS) algorithm, and the time complexity is $O(n)$. When constructing the adjacency matrix, the time complexity is $O(n^2)$. Therefore, the time complexity of feature extraction is $O(n + n^2)$.

Features normalization. Due to the difference in feature extraction, the first four-dimensional features are large integers, and the last eight-dimensional features are decimals between 0 and 1. Therefore, if these 12-dimensional features are directly used as the input of machine learning models, some machine learning models (such as K-Nearest Neighbors) will only focus on the first four-dimensional features while ignoring the last eight-dimensional features during training. In addition, some machine learning models also have requirements for the format of input data. To make *Block-gram* features suitable for most mainstream machine learning models, we use linear normalization to process these features. The normalization method is defined in Equation (1).

$$x^*_{(n,f)} = \frac{x_{(n,f)} - \min\limits_{0<i<r} x_{(i,f)}}{\max\limits_{0<i<r} x_{(i,f)} - \min\limits_{0<i<r} x_{(i,f)}} \tag{1}$$

where $x_{(n,f)}$ represents the value of the feature f in the n row.

4 Evaluation

4.1 Experimental Setup

Configuration. We perform experiments on a Windows 10 machine with 12th Gen Intel Core 2.10 GHz CPUs and 32GB RAM and use the GPU of a 1060ti graphics card to train the model and predict the results. To verify the efficiency and the validity of the above 12-dimensional feature, we choose seven state-of-the-art machine learning algorithms, such as eXtreme Gradient Boosting (XGBoost), Random Forest(RF), K-Nearest Neighbors(KNN), Logistic Regression(LR), Decision Tree(DT), Naive Bayes, and Long short-term memory(LSTM), as the training and detection model. We use the sklearn library in python3.6.8 to build the machine learning algorithms. We also select accuracy, recall, F1-score, and latency as the measured metrics of the model.

Datasets. We select 3000 smart contracts as the dataset for performance analysis from Contractward [5], 70% for training, and 30% for testing. The size of the dataset is 62.7MB. There are 871 contracts with vulnerabilities and 2179 contracts without vulnerabilities. The vulnerabilities of the dataset include integer overflow and integer underflow vulnerabilities, callstack depth attack vulnerability, transaction-order dependence vulnerability, timestamp dependency vulnerability, and re-entrancy vulnerability.

4.2 Performance Analysis

Detection performance. As shown in Table 3, in terms of accuracy, the maximum value of the seven models is 82.22% and the minimum value of the seven models is 73.88% when they use *Block-gram* features. When they use features extracted from opcodes by *N-gram*, almost all models' accuracy drops. *Block-gram* features perform better than features extracted by *N-gram* in terms of accuracy. In terms of recall and F1-score, the performance of K-Nearest Neigh-

Table 3: Model Performance by *Block-gram* Features.

Model	Feature	Accuracy(%)	Recall(%)	F1-score(%)	Latency(ms)
XGBoost	*Block-gram*	**82.22**	93.27	88.16	**0.2**
	N-gram	80.4	95.3	87.33	15
Random Forest	*Block-gram*	**81.77**	95.00	88.10	**0.3**
	N-gram	70.55	98.27	82.58	12
K-Nearest Neighbors	*Block-gram*	**75.44**	**84.51**	**83.01**	**0.004**
	N-gram	51.55	36.62	51.77	0.1
Logistic Regression	*Block-gram*	75.44	98.28	85.04	**0.01**
	N-gram	75.88	87.17	83.70	0.4
Decision Tree	*Block-gram*	77.44	84.66	84.20	**0.004**
	N-gram	76.11	83.72	83.27	2.6
Naive Bayes	*Block-gram*	**76.66**	**87.17**	**84.13**	**0.0003**
	N-gram	60.33	58.37	67.63	0.09
LSTM	*Block-gram*	**73.88**	66.28	59.55	**1**
	N-gram	69	75.10	58.42	90

bors and Naive Bayes drops significantly when they use features extracted by *N-gram*. And other measured metrics of K-Nearest Neighbors and Naive Bayes also dropped significantly. The two models poor perform when dealing with *N-gram* features. The reason is that we have considered the real jump relationship when the smart contract runs and the expert knowledge of six vulnerabilities but the *N-gram* method only extracts the combination of the opcode sequence.**Detection latency.** As shown in Fig.3, the detection latency of all models is greatly improved when they use *Block-gram* features. Among the seven models, the detection latency of the decision tree model has reduced 650 times when using *Block-gram* features compared with using *N-gram*. In addition, compared with the traditional method Oyente and other machine learning methods (for example, VSCL and EtherGIS), the latency of using *Block-gram* features is also significantly reduced. Since VSCL and EtherGIS did not publish open-source code, we directly cited the detection delay published in the paper. Their experimental environment is far better than our work. The reason is that the dimension of the feature space extracted by *N-gram* is too high, as high as tens of hundreds of dimensions during the initial extraction. In training, the features will be expanded up to tens of thousands of dimensions. When using *Block-gram* features, the initially extracted feature space is only 12-dimensional. Even if it needs to expand during the training process, it is only 15-dimensional at most. Significant differences in feature space dimensions lead to differences in latency. The experimental results demonstrate that *Block-gram* features are efficient and can be used by mainstream machine learning models.

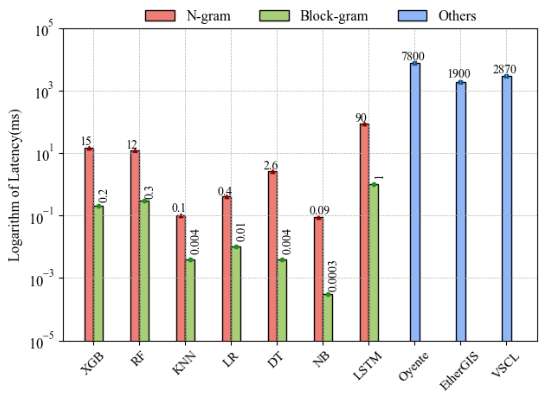

Fig. 3: The Latency of models

5 Conclusion

This paper addressed improving the detection efficiency for the quick detection requirements of large-scale smart contracts. We only used extensible 12-dimensional features mining from bytecode and opcode. The low-dimensional features will speed up the detection time 25 to 650 times without influencing the model performance (*accuracy* etc.). We can also extend these features to support more vulnerability detection and security analysis in the future. Compared with the existing thousand-dimensional feature space, the features improve the detection efficiency and extend the detection range. The evaluation based on seven state-of-the-art learning-based methods has shown the effectiveness of *Block-gram* features and can significantly improve detection efficiency.

Acknowledgment

This work is partially supported by the National Natural Science Foundation (62272248), the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201905, CARCHA202108), the Natural Science Foundation of Tianjin (20JCZDJC00610), CAAI-Huawei MindSpore Open Fund (CAAIXSJLJJ-2021-025A) and Sponsored by Zhejiang Lab (2021KF0AB04).

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. F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "Evm*: from offline detection to online reinforcement for ethereum virtual machine," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 554–558.
3. A. R. Sai, C. Holmes, J. Buckley, and A. L. Gear, "Inheritance software metrics on smart contracts," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 381–385.
4. 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.
5. 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.
6. 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.

7. J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.
8. 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.
9. B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.
10. E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*. Springer, 2018, pp. 513–520.
11. X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "Deescvhunter: A deep learning-based framework for smart contract vulnerability detection," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.
12. K. Gai and M. Qiu, "Reinforcement learning-based content-centric services in mobile sensing," *IEEE Network*, vol. 32, no. 4, pp. 34–39, 2018.
13. F. Mi, Z. Wang, C. Zhao, J. Guo, F. Ahmed, and L. Khan, "Vscl: Automating vulnerability detection in smart contracts with deep learning," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–9.
14. 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.
15. Q. Zeng, J. He, G. Zhao, S. Li, J. Yang, H. Tang, and H. Luo, "Ethergis: A vulnerability detection framework for ethereum smart contracts based on graph learning features," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2022, pp. 1742–1749.
16. T. Li, Y. Fang, Y. Lu, J. Yang, Z. Jian, Z. Wan, and Y. Li, "Smartvm: A smart contract virtual machine for fast on-chain dnn computations," *IEEE Transactions on Parallel and Distributed Systems*, 2022.
17. H. Qiu, M. Qiu, G. Memmi, Z. Ming, and M. Liu, "A dynamic scalable blockchain based communication architecture for iot," in *International Conference on Smart Blockchain*. Springer, 2018, pp. 159–166.
18. K. Gai, Y. Wu, L. Zhu, Z. Zhang, and M. Qiu, "Differential privacy-based blockchain for industrial internet-of-things," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 4156–4165, 2019.
19. Z. Tian, M. Li, M. Qiu, Y. Sun, and S. Su, "Block-def: A secure digital evidence framework using blockchain," *Information Sciences*, vol. 491, pp. 151–165, 2019.
20. F. Contro, M. Crosara, M. Ceccato, and M. Dalla Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 127–137.