Smart Contract Vulnerability Detection Based on Clustering Opcode Instructions

Xiguo Gu, Huiwen Yang, Shifan Liu, Zhanqi Cui*
Computer School, Beijing Information Science and Technology University, Beijing, China
Email:{xiguo_gu, yhw_yagol, pawn2017, czq}@bistu.edu.cn

Abstract

Smart contracts are programs running on the blockchain. In recent years, due to the continuous occurrence of smart contract security accidents, how to effectively detect vulnerabilities in smart contracts has received extensive attention. Machine learning-based vulnerability detection techniques have the advantage of not requiring expert rules. However, existing approaches have limitations in identifying vulnerabilities caused by version updates of smart contract compilers. In this paper, we propose OC-Detector, a smart contract vulnerabilities detection approach based on opcode instruction clustering. OC-Detector learns the characteristics of opcode instructions to cluster them and replaces opcode instructions belonging to the same cluster with the cluster number. After that, the similarity is calculated against the contract in the vulnerability database to identify vulnerabilities. Experimental results demonstrate that OC-Detector improves the F_1 value of detecting vulnerabilities from 0.04 to 0.40 compared to DC-Hunter, Securify, SmartCheck, and Osiris. Additionally, compared to DC-Hunter, F_1 value is improved by 0.27 when detecting vulnerabilities in smart contracts compiled by different version compilers.

***Index Terms*—Ethereum, Smart Contracts, Slicing, Word embedding, Clustering.**

I. Introduction

The blockchain is a distributed database with characteristics of decentralization, immutability, traceability, and joint maintenance by multiple parties[1]. Smart contracts are programs that run on the blockchain, which help developers apply blockchain techniques to several fields, such as finance, education, and the internet of things. Ethereum is the most popular blockchain platform, which uses the Ethereum Virtual Machine (EVM) to execute smart contracts.

While smart contracts have given rise to a variety of applications, its stored digital assets make it vulnerable to numerous attacks. For instance, in 2016, a security vulnerability in the DAO contracts led to the loss of more than 3.6 million Ethereum tokens, with a value of about $60 million. In this case, the attacker repeatedly reentered the transfer function to steal Ethereum tokens through a reentrance vulnerability in the DAO contract[2].

Therefore, it is necessary to study effective vulnerability detection techniques to mitigate losses caused by vulnerabilities in smart contracts. Due to the rapid development of machine learning technology, machine learning-based vulnerability detection techniques for smart contracts have gained considerable attention in recent years. Specially, it is often challenging to obtain the source code of smart contracts, while contract bytecode can be directly acquired from Ethereum platforms. Hence, bytecode-based techniques for detecting vulnerabilities in smart contracts are more practical. Han et al.[3] leverage program slicing techniques, track the data flow and extract slices from contract bytecode. They also used graph embedding techniques to capture more structural information to improve the performance of detecting smart contract vulnerabilities. However, the Solidity language, which is used to write smart contract code in Ethereum, is still in the evolutionary stage. For example, 11 versions were released in 2021. Depending on the version of Solidity compilers, the same statement in the smart contract may have different opcode instructions. Therefore, it is difficult to detect vulnerabilities by using machine learning-based approaches when the model is trained on other versions of smart contracts.

To solve the above problems, we present OC-Detector, a smart contract vulnerability detection approach based on clustering opcode instructions. First, OC-Detector transforms the bytecode of smart contracts into opcode instructions, which are then embedded into vectors via word embedding techniques. Second, OC-Detector utilizes a clustering algorithm to classify vectors into several clusters based on their semantic information. After that, we construct a dataset containing 4 types of vulnerabilities by selecting representative contracts, including reentrance, timestamp-dependencies, access control, and unchecked call return values. Finally, OC-Detector slices the target contracts and represents the opcode instructions belonging to the same cluster uniformly based on the results of clustering and detects vulnerabilities by comparing the similarity between the target contract and the dataset. Experimental results demonstrate that OC-Detector is more effective than other existing approaches, including DC-Hunter[3], Security[4], SmartCheck[5] and Osiris[6]. The F_1 value is improved by 0.04 to 0.40. Additionally, compared with DC-Hunter, the F_1 value is improved by 0.27 when detecting vulnerabilities

* Zhanqi Cui is the corresponding author.
DOI reference number: 10.18293/SEKE2023-183

in smart contracts compiled by different version compilers.

To summarize, our main contributions are as follows:

- We propose OC-Detector, a smart contract vulnerability detection approach based on clustering opcode instructions. OC-Detector can alleviate false negatives and false positives caused by inconsistent compiled opcode instructions by different version compilers.
- Experiments are conducted to compare OC-Detector with DC-Hunter, Securify, SmartCheck, and Osiris. Experimental results shows that OC-Detector outperforms existing approaches on a large-scale dataset.

The remainder of the paper is structured as follows: Section II provides an example to illustrate the motivation of this research. Section III presents details of the approach. Section IV describes the experiments and evaluations to validate the effectiveness of the OC-Detector. Section V analyzes threats of validity. Section VI reviews related work. Section VII concludes the paper and discusses future research directions.

II. The Motivation Example

In practical applications, opcode instructions play a vital role in the execution of programs, and their variations can result in significant differences in program behaviors. Due to the frequent iteration of compiler versions, opcode instructions generated by different versions of compilers can vary significantly. Using existing vulnerability detection techniques to detect smart contract vulnerabilities can may lead to false positives and false negatives due to inconsistencies of opcode instructions. Figure 1 shows a vulnerable contract that is compiled into various opcode instructions by different versions of compilers.

Figure 1 shows "PERSONAL_BANK[1]", an example of a smart contract with a vulnerability, in which the vulnerability is located at line 10. The contract may execute a recursive call at line 10, which triggers a vulnerability that could cause damage to the contract caller. In line 5 of the contract, different version compilers will compile the contract to different opcode instructions. For instance, the 0.4.11 version Solidity compiler generates opcode instructions as "ISZERO, ISZERO, PUSH, JUMPI, INVALID, JUMPDEST, PUSH", whereas the 0.4.26 version generates "ISZERO, DUP1, ISZERO, PUSH, JUMPI, PUSH, DUP1, INVALID". When using our replicated DC-Hunter to detect opcode instructions compiled with version 0.4.26 compiler using opcode instructions generated by version 0.4.11 compiler, it fails to detect the vulnerability on line 10 of the smart contract. While by clustering the opcode instructions using OC-Detector, the opcode instructions with similar features in different versions can be clustered into the same cluster, and to make a uniform representation. This effectively eliminates the differences of opcode instruction sequences generated by the two versions of compiler. As a result, the vulnerability in the contract can be successfully detected without the influence of opcode instructions generated by different version compilers.

[1]PERSONAL_BANK: https://github.com/chen2233/smartbugs/blob/master/dataset/reentrancy/0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol

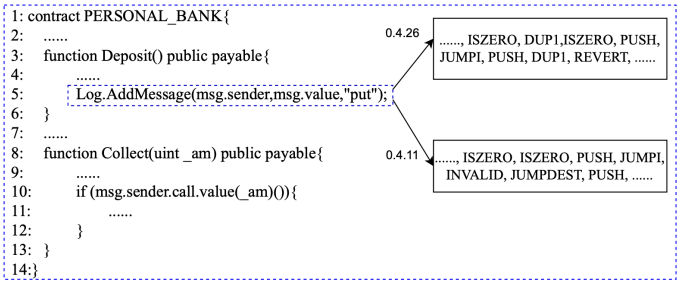

Fig. 1: A motivation example of smart contract.

III. Smart Contract Vulnerability Detection Based on Clustering Opcode Instructions

Figure 2 presents the proposed approach for detecting smart contract vulnerabilities based on clustering opcode instructions. First, OC-Detector converts the bytecode of a smart contract into opcode instructions, then embeds them into vectors by using word embedding models. Second, OC-Detector utilizes clustering algorithms to classify vectors of opcode instructions into several clusters based on their semantic information. After that, representative contracts are selected to construct a vulnerability dataset. The opcode instructions that belong to the same cluster in the vulnerability dataset are uniformly represented. Finally, the target contracts are sliced and the opcode instructions belonging to the same cluster are uniformly represented. The vulnerability is then detected by comparing the similarity of the target contract with those in the vulnerability dataset.

A. Opcode instruction vectorization.

Due to the difficulty of obtaining the source code of smart contracts, it is challenging to learn contract features directly from the source code. Therefore, we utilize smart contract opcode instructions to learn contract features and translate the opcode instructions into vectors.

When learning contract features, we compile the bytecode of the smart contracts and transforming them into opcode instructions. The opcode instructions are then normalized by removing the operands and applying a delimiter to separate different instructions. Next, a word embedding model is trained with the normalized opcode instruction sequences, to learn the features of opcode instructions. This step enables the extraction of meaningful features that can be used to identify similarities or differences between smart contracts and provide insights into their functionalities and potential vulnerabilities.

B. Opcode instructions clustering.

There are a total of 143 opcode instructions available in smart contracts[2]. While some opcode instructions have similar functions, they may differ slightly in their symbolic representation. For instance, PUSH1, PUSH2 and PUSH3 are all instructions used for pushing values onto the stack, but they differ in the number of bytes they push. Furthermore, it

[2] Ethervm: https://www.ethervm.io/

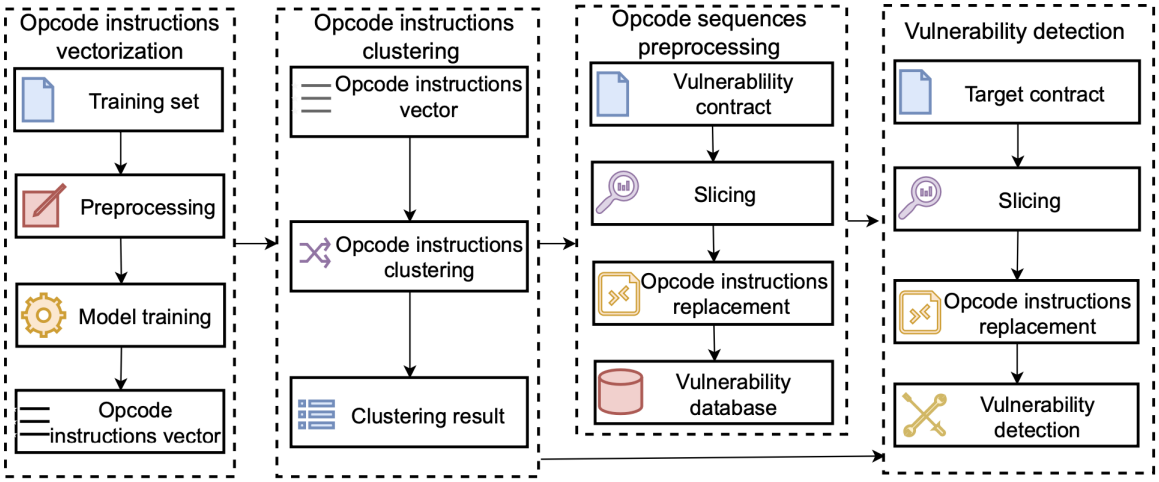

Fig. 2: Framework of OC-Detector.

is imperative to note that due to different version of Solidity compilers, the opcode instructions generated for the same contract may vary. As shown in the motivation example.

To solve the above problem, OC-Detector clusters opcodes with similar functionality into the same cluster. In the clustering process, k opcode instructions are randomly selected as initial cluster centers. The distance between each opcode instruction and each cluster center is calculated, and the opcode instructions are assigned to the cluster center closest to it. For each opcode instruction assigned, the cluster centers are recalculated based on existing opcode instructions in the clusters. This process is repeated until all opcode instructions are assigned to the corresponding clusters.

C. Opcode sequences preprocessing.

Using the opcode instruction sequences generated by contract compilation for similarity calculation can be interfered by a mass of irrelevant instructions, resulting in performance degradation of detecting vulnerabilities.

To solve this problem, OC-Detector slices the opcode sequence to reduce the interference of noise based on analyzing the dependency between opcode instructions and external data. During the opcode instructions slicing process, smart contracts containing representative vulnerabilities are first manually selected and transformed into opcode instruction sequences through preprocessing. The data dependency relationship between opcode instructions and external data in the opcode instruction sequences is then analyzed. Instructions that introduce external data and instructions that use external data are sliced according to the data dependency relationship. Finally, the sliced opcode instructions belonging to the same cluster are replaced with the cluster number, and the opcode instruction sequence is added to the vulnerability dataset.

D. Vulnerability detection.

To effectively detect vulnerabilities in smart contracts, OC-Detector utilizes similarity calculation to detect vulnerabilities in smart contracts. To calculate the similarity, Levenstein[3] distance is used to calculate the similarity between the opcode instruction sequences of two smart contract. Specifically, OC-Detector calculates the similarity between the target contract and each contract in the vulnerability dataset one by one. If the similarity exceeds a threshold p, OC-Detector considers the contract to be vulnerable and output the type of vulnerability.

IV. Experimnts and Evaluations

All experiments in this paper are run on a computer with i7-6700H CPU and 16GB of memory, and the development and running environment is Ubuntu 18.04 and Python 3.9.

The following three research questions are designed to verify the effectiveness of OC-Detector.

- RQ1: How does the performance of OC-Detector compare to other existing tools?

To evaluate the performance, OC-Detector is compared with DC-Hunter, SmartCheck[4], Securify[5] and Osiris[6]. SmartCheck, Securify and Osiris are open source tools, while DC-Hunter is not open-source, but it shares similarities with OC-Detector, so we implemented it according to the descriptions of the paper. In addition, to verify the effectiveness of OC-Detector in alleviating false positives and false negatives caused by inconsistent opcode instructions due to different versions of Solidity compilers, we compare OC-Detector with DC-Hunter which is similar to our approach.

- RQ2: How does the Word2Vec model affect the performance of OC-Detector?

To verify the impact of the word embedding model trained by opcode instruction sequence for vectorizing opcode instructions, we replace the Word2Vec model trained by opcode instruction sequence with the default Word2Vec model (denoted as OC-Detector*) and compare it with OC-Detector.

[3] Levenstein: https://github.com/ztane/python-Levenshtein
[4] SmartCheck: https://github.com/smartdec/smartcheck
[5] Securify: https://github.com/eth-sri/securify
[6] Osiris: https://github.com/christoftorres/Osiris

- RQ3: How does the clustering opcode instructions affect the performance of OC-Detector?

To verify the impact of the clustering algorithm used in OC-Detector, we directly calculate the similarity between the target contract and the contract in vulnerability dataset without clustering (denoted as OC-Detector w/o clustering) and compare it with OC-Detector.

A. Experimental dataset.

The experimental objects utilized two widely used datasets, include Xblock[7] and Smartbugs[8], which have been used in prior smart contract vulnerability detection studies[7][8]. The Xblock dataset comprises a total of 149,363 smart contracts, from which 10,000 contracts were randomly selected for training the Word2Vec model. Another 5,000 contracts were randomly chosen as the target contracts to evaluate the effectiveness of OC-Detector. Since the Xblock dataset lacks vulnerability information, we utilized Slither[9] to analyze the 5,000 target contracts and annotate their vulnerability labels. Empirical studies have shown that Slither is the most effective static analysis tool for detecting vulnerabilities of smart contracts[10][11]. Therefore, we chose Slither to annotate the vulnerability information of the dataset. The Smartbugs dataset contains 143 smart contracts that have vulnerability information. To construct the vulnerability dataset, we manually selected 20 smart contracts from this dataset, include 5 reentrant vulnerabilities, 5 timestamp-dependent vulnerabilities, 5 access control vulnerabilities, and 5 unchecked CALL return value vulnerabilities.

B. Evaluation metrics

We utilize standard metrics, include precision, recall, and F_1 to evaluate the effectiveness of OC-Detector. Additionally, we utilize Silhouette Coefficient[12] to evaluates the clustering effect of the K-means algorithm[13]. These evaluation metrics are frequently utilized in vulnerability detection approaches[14][15] and clustering algorithms[16].

C. Parameter settings.

To determine proper values of the parameters, we randomly select 500 contracts from the Xblock dataset and applying various vulnerability detection approaches. We measured the True Positive Rate and False Negative Rate of each approach by different similarity thresholds. Through these experiments, the optimal performance of OC-Detector, DC-Hunter, OC-Detector*, and OC-Detector w/o clustering are achieved with threshold values of 0.40, 0.50, 0.40, and 0.50, respectively. In subsequent experiments, we evaluated the vulnerability detection performance of each approach using these threshold values.

To determine the optimal number of clusters for clustering opcode instructions with different dimensions, we trained Word2Vec models to vectorize opcode instructions into different dimensions, and clustered the opcode instruction vectors with different dimensions. We calculated Silhouette Coefficient to determine the optimal number of clusters. The experimental results demonstrate that for opcode instruction vectors with 25 dimensions, the optimal number of clusters is 4, while for opcode instruction vectors with 256 dimensions, the optimal number of clusters is 20.

After that, we randomly selected 500 contracts from the Xblock dataset to detect potential vulnerabilities. The results indicate that the optimal vulnerability detection performance was achieved when opcode instructions were clustered into 4 clusters, Thus, the opcode instructions will be clustered into 4 clusters in subsequent experiments to maximize the effectiveness of detecting vulnerabilities.

[7] Xblock: http://xblock.pro/ethereum

[8] Smartbugs: https://github.com/smartbugs/smartbugs

TABLE I: The performance of different approaches to detect smart contract vulnerabilities.

Approach	Avg Time(s)	Precision	Recall	F_1
DC-Hunter	6.16	0.44	0.10	0.16
Securify	23.79	0.46	0.23	0.31
SmartCheck	70.86	0.54	0.51	0.52
Osiris	21.18	0.23	0.14	0.17
OC-Detector	2.31	0.59	0.53	0.56

D. Experimental results and analysis.

1) RQ1: How does the performance of OC-Detector compare to other existing tools?

To answer this question, we compare the vulnerability detection performance with other approaches, such as DC-Hunter, Security, SmartCheck, and Osiris. The experimental results are shown in Table I. As the result show, OC-Detector outperforms other four approaches in terms of precision, recall, F_1, and average time taken to analysis each contract. Specifically, the precision, recall, and F_1 value are improved from 0.05 to 0.36, from 0.02 to 0.43, and from 0.04 to 0.40, respectively. In addition, the average time to analysis one smart contract is reduced from 3.85 to 68.55 seconds, when compared to other four approaches. To validate the effectiveness of OC-Detector in solving the issue of false positives and false negatives caused by inconsistent opcode instructions generated by different versions of Solidity compilers, OC-Detector is compared with DC-Hunter. Since the contracts in the vulnerability dataset are compiled with compilers of version 0.4.25, 0.5.15, 0.7.1, and 0.8.7, we randomly selected 500 smart contracts compiled with compilers other than those above versions from the dataset. Then, these 500 smart contracts are analyzed by OC-Detector and DC-Hunter to detect vulnerabilities. The experimental results are shown in Table II.

As the result show, OC-Detector outperforms DC-Hunter in terms of precision, recall, and F_1. The precision, recall, and F_1 value are improved by 0.14, 0.27, and 0.27, respectively. Furthermore, the average time to analysis one smart contract is reduced by 3.87 seconds.

Answer to RQ1: OC-Detector outperforms other tools, include DC-Hunter, Securify, SmartCheck, and Osiris, in terms of identifying vulnerabilities in smart contracts. Moreover,

TABLE II: The performance comparison of OC-Detector and DC-Hunter with different compliers.

Approach	Avg Time(s)	Precision	Recall	F_1
DC-Hunter	6.25	0.45	0.13	0.20
OC-Detector	2.38	0.59	0.40	0.47

TABLE III: The performance comparison of OC-Detector and OC-Detector*.

Approach	Precision	Recall	F_1
OC-Detector	0.59	0.53	0.56
OC-Detector*	0.41	0.45	0.43

OC-Detector effectively alleviates the problem of false positives and false negatives caused by inconsistent opcode instructions generated by different versions of compilers.

2) RQ2: How does the Word2Vec model affect the performance of OC-Detector?

To answer this question, OC-Detector is compared with OC-Detector* which utilize the glove.twitter.27B.25d[9] model to vectorize opcode instructions instead of the Word2Vec model trained by using smart contract opcode instruction sequences, then calculate the similarity between the target contract and the contract of vulnerability dataset to detect vulnerabilities.

To determine the optimal number of clusters for the opcode instruction vectors generated by the glove.twitter.27B.25d model, we evaluate the results of clustering with different number of clusters by using Silhouette Coefficient. Experimental results show that the optimal number of clusters is 33.

The experimental results are shown in Table III. As the result shows, compared to OC-Detector*, the values of precision, recall, and F_1 value are improved by 0.18, 0.08, and 0.13, respectively.

Answer to RQ2: Smart contract opcode instruction sequences can be used to learn context features of opcode instructions, which improve the performance of smart contract vulnerability detection.

3) RQ3: How does the clustering opcode instructions affect the performance of OC-Detector?

To answer this question, OC-Detector is compared with OC-Detector w/o clustering, which directly computed the similarity between the target contract and the contract in vulnerability dataset without clustering the opcode instructions. The experimental results are shown in Table IV.

As the result show, compared to OC-Detector w/o clustering, the precision, recall, and F_1 value are improved by 0.17, 0.12, and 0.15, respectively.

Answer to RQ3: The clustering algorithm used by OC-Detector helps to eliminate differences in opcode instructions compiled by different versions of compilers and improve the performance of detecting smart contract vulnerabilities.

[9]glove.twitter.27B.25d: https://github.com/stanfordnlp/GloVe

TABLE IV: The performance comparison of OC-Detector and OC-Detector w/o clustering.

Approach	Precision	Recall	F_1
OC-Detector	0.59	0.53	0.56
OC-Detector w/o clustering	0.42	0.41	0.41

V. Threats to Validity

External validity refers to ensure the generality of the experimental results. The contract to be checked in the experiment are taken from Xblock, and the contract in the Xblock dataset are real contract from the Ethernet, which being able to better validate the effectiveness of the approach. The smart contracts in the Smartbugs dataset are used to build the vulnerability dataset. These two datasets are widely used in studies of smart contract vulnerability detection[7][8]. Although the vulnerabilities contained in these two datasets are representative, same performance cannot be ensured when applying OC-Detector on other datasets.

Internal validity is primarily related to factors that affect the correctness of the experiment. In this paper, we implemented the clustering algorithm using the Sklearn[10], trained the Word2vec model using the Genimn[11] and calculated similarity using the Levenshtein library to ensure the correctness of the implementation. Additionally, since DC-Hunter is not open-source, so we reimplemented it according to the descriptions of the paper. Although we have tested and checked results multiple times, the implementation details may differ from the original paper.

Construct validity is primarily concerned with the evaluation metrics used in the experiment. Precision, recall, and F_1 value are used to evaluate the performance of detecting smart contract vulnerabilities in the experiment. These evaluation metrics are widely used in smart contract vulnerability detection approaches[17][18]. In addition, Silhouette Coefficient is used to evaluate the effect of clustering, which is commonly used to in previous studies[19][20].

VI. Related Work

The machine learning-based vulnerability technology has been widely used in smart contracts. Previous research has focused on learning contract semantics and syntax using natural language processing techniques. For example, Wang et al.[21] learned syntax and semantic features from bytecode to detect smart contract vulnerabilities, while Liu et al.[22] learned code features from contract source code to detect vulnerabilities. However, extraneous code generates noise, which affects the performance of learning contract syntax and semantic information.

To solve this problem, some researchers proposed slicing technology to filter key code and reduce interference of code unrelated to vulnerabilities. For example, Han et al.[3]

[10] Sklearn: https://github.com/automl/auto-sklearn

[11] Genimn: https://github.com/RaRe-Technologies/gensim

reduce noise by slicing and use graph embedding algorithms to convert program-dependent graphs into vectors to detect vulnerabilities. This approach has demonstrated that slicing can improve the performance of vulnerability detection.

Recent research has focused on using features learned through natural language processing techniques to detect vulnerabilities by machine learning models. For example, Yang et al.[23] proposed a new self-supervised learning approach for smart contract representation. Cai et al.[24] encode smart contract's function into a graph with sufficient semantic features. They then utilize bidirectional gated graph neural network with a hybrid attention pooling layer to learn the code features, efficiently capturing vulnerability-related features from the graph for vulnerability detection. These approaches have been proven to be effectively to improve the performance of smart contract vulnerability detection.

To improve performance in detecting smart contract vulnerabilities, we learn opcode instruction features after slicing opcode instruction sequences and use clustering algorithms in machine learning to eliminate the differences between different opcode instructions to detect contract vulnerabilities.

VII. Conclusions and Future work

In this paper, we propose a smart contract vulnerability detection approach based on clustering opcode instructions and implement a prototype tool OC-Detector. Experimental results on the Xblock dataset show that OC-Detector outperforms DC-Hunter, Securify, SmartCheck, and Osiris in detecting smart contract vulnerabilities. Specially, mitigates false positive and false negative caused by inconsistent opcode instructions generated by different versions of compilers.

With the wide application of smart contracts, in addition to Solidity, Go, C++, and other programming languages are also commonly used for developing smart contracts, which also affected by typical vulnerabilities such as reentry, overflow and unchecked CALL return value. However, different smart contract programming languages inevitably generate different opcode instructions, which invalidate vulnerability detection approaches based on specific programming languages. As part of our future work, we plan to solve the issue of detecting vulnerabilities of smart contracts written in different programming languages.

Acknowledgement

This work was supported in part by the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project (No. QXTCP C201906).

References

[1] Z. Gao, L. Jiang and X. Xia, et al. Checking smart contracts with structural code embedding[J]. IEEE Transactions on Software Engineering. 2020, 47(12): 2874-2891.

[2] P. Shen, S. Li and M. Huang, et al. A Survey on Safety Regulation Technology of Blockchain Application and Blockchain Ecology[C]//2022 IEEE International Conference on Blockchain. 2022: 494-499.

[3] S. M. Han, B. Liang and J. J. Huang. DC-Hunter: Detecting dangerous smart contracts via bytecode matching[J]. Journal of Cyber Security. 2020, 5(3): 100-112.

[4] P. Tsankov, A. Dan, and D. Drachsler-Cohen, et al. Securify: Practical security analysis of smart contracts[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 67-82.

[5] S. Tikhomirov, E. Voskresenskaya and I. Ivanitskiy, et al. Smartcheck: Static analysis of ethereum smart contracts[C]//Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain. 2018: 9-16.

[6] C F. Torres, J. Schütte and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts[C]//Proceedings of the 34th Annual Computer Security Applications Conference. 2018: 664-676.

[7] J F. Ferreira, P. Cruz and T. Durieux, et al. Smartbugs: A framework to analyze solidity smart contracts[C]//Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 2020: 1349-1352.

[8] H. W. Yang, Z. Q. Cui and X. Chen, et al. Defect Prediction for Solidity Smart Contracts Based on Software Measurement[J]. Journal of Software. 2022, 33(5): 1587-1611.

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

[10] M. Ren, Z. Yin and F. Ma, et al. Empirical evaluation of smart contract testing: What is the best choice?[C]//Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 566-579.

[11] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020: 415-427.

[12] H. Řezanková. Different approaches to the silhouette coefficient calculation in cluster evaluation[C]//21st International Scientific Conference AMSE Applications of Mathematics and Statistics in Economics. 2018: 1-10.

[13] H. B. Tambunan, D. H. Barus and J. Hartono, et al. Electrical peak load clustering analysis using K-means algorithm and silhouette coefficient[C]//2020 International Conference on Technology and Policy in Energy and Electric Power. 2020: 258-262.

[14] Q. Zeng, J. He and G. Zhao, et al. EtherGIS: A Vulnerability Detection Framework for Ethereum Smart Contracts Based on Graph Learning Features[C]//2022 IEEE 46th Annual Computers, Software, and Applications Conference. 2022: 1742-1749.

[15] A. Ghaleb, J. Rubin and K. Pattabiraman. eTainter: detecting gas-related vulnerabilities in smart contracts[C]//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 728-739.

[16] A. M. Bagirov, R. M. Aliguliyev and N Sultanova. Finding compact and well-separated clusters: Clustering using silhouette coefficients[J]. Pattern Recognition, 2023, 135: 109144.

[17] J. Huang, S. Han and W. You, et al. Hunting vulnerable smart contracts via graph embedding based bytecode matching[J]. IEEE Transactions on Information Forensics and Security. 2021, 16: 2144-2156.

[18] H. Wang, G. Ye and Z. Tang, et al. Combining graph-based learning with automated data collection for code vulnerability detection[J]. IEEE Transactions on Information Forensics and Security, 2020, 16: 1943-1958.

[19] P. J. Kaur. Cluster quality based performance evaluation of hierarchical clustering method[C]//2015 1st International Conference on Next Generation Computing Technologies. 2015: 649-653.

[20] R. Hidayati, A. Zubair and A. H. Pratama, et al. Analisis Silhouette Coefficient pada 6 Perhitungan Jarak K-Means Clustering[J]. Techno. Com, 2021, 20(2): 186-197.

[21] W. Wang, J. Song and G. Xu, et al. Contractward: Automated vulnerability detection models for ethereum smart contracts[J]. IEEE Transactions on Network Science and Engineering. 2020, 8(2): 1133-1144.

[22] C. Liu, H. Liu and Z. Cao, et al. Reguard: finding reentrancy bugs in smart contracts[C]//Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. 2018: 65-68.

[23] S. Yang, X. Yang and B. Shen. Self-supervised learning of smart contract representations[C]//Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. 2022: 82-93.

[24] J. Cai, B. Li and J Zhang, et al. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection[J]. Journal of Systems and Software, 2023, 195: 111550.