DOI:10.1587/transinf.2024EDL8004

Publicized:2024/05/27

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

Smart Contract Timestamp Vulnerability Detection Based on Code Homogeneity

Weizhi WANG[†], **Lei XIA**[††], **Zhuo ZHANG**[†††a)], ***Nonmembers*, *and* Xiankai MENG**[††††], ***Member***

SUMMARY Smart contracts, as a form of digital protocol, are computer programs designed for the automatic execution, control, and recording of contractual terms. They permit transactions to be conducted without the need for an intermediary. However, the economic property of smart contracts makes their vulnerabilities susceptible to hacking attacks, leading to significant losses. In this paper, we introduce a smart contract *timestamp* vulnerability detection technique HomoDec based on code homogeneity. The core idea of this technique involves comparing the homogeneity between the code of the test smart contract and the existing smart contract vulnerability codes in the database to determine whether the tested code has a *timestamp* vulnerability. Specifically, HomoDec first explores how to vectorize smart contracts reasonably and efficiently, representing smart contract code as a high-dimensional vector containing features of code vulnerabilities. Subsequently, it investigates methods to determine the homogeneity between the test codes and the ones in vulnerability code base, enabling the detection of potential *timestamp* vulnerabilities in smart contract code.

key words: *Smart Contract, vulnerability detection, code homogeneity*

1. Introduction

Over the last decade, as blockchain technology and digital cryptocurrencies have become widely adopted, the emerging platforms, notably Bitcoin [1] and Ethereum [2], have gained increasing recognition and development. Smart contracts [3] stand out as a pivotal technological innovation, utilizing a consensus mechanism that leverages user confirmations on the blockchain to ensure reliability. This ambitious initiative has garnered market recognition, leading to the release of numerous smart contracts on the Ethereum platform. Smart contracts are commonly crafted using a higher-level language known as Solidity [3], which bears similarities to JavaScript and C++. Following the coding process, they are subsequently compiled into EVM (Ethereum Virtual Machine) bytecode. A fundamental principle within the Ethereum ecosystem dictates that all transactions, purchases, and sales are irreversible and immutable. Despite this principle, for the sake of ease in deployment and development, many contracts undergo modifications based on the original code. Unfortunately, this practice opens up opportunities for hackers to exploit vulnerabilities. Therefore, it is of great significance to effectively detect smart contract code vulnerabilities before their deployment.

Various effective detection techniques have been developed by researchers. These include static analysis-based approaches, formal verification-based methods, fuzzing-based strategies, and symbolic execution-based methods [4]. Traditional detection methods heavily rely on expert knowledge, necessitating the manual summarization of rules and patterns for identifying vulnerabilities beforehand, which are susceptible to vulnerabilities and may be vulnerable to sophisticated tactics employed by clever attackers, surpassing known rules. Consequently, the applicability of these methods is constrained. Detecting code homogeneity is a crucial aspect of both software maintenance and development. This process involves identifying similarities between different pieces of code, and its applications are wide-ranging and essential in various domains. Some notable applications include detecting candidate libraries, program comprehension, malware detection, plagiarism and copyright infringement detection, context-based inconsistency detection and refactoring opportunities [5]. Various tools and methods have been developed by researchers to address code homogeneity detection challenges, which demonstrate the diversity of approaches in tackling the code homogeneity detection problem, ranging from textual comparison to higher-level structural analysis. The continual development of such tools and techniques is crucial for maintaining and improving the quality and security of software systems.

In view of this, we investigate more on how to represent smart contracts efficiently in order to conduct code homogeneity detection and propose a vulnerability detection method HomoDec for *timestamp* vulnerabilities. Specifically, we utilize a pre-trained model to represent the smart comtract code as a high-dimensional vector incorporating features of *timestamp* vulnerabilities. Then, HomoDec performs similarity detection by these constructed high-dimensional vectors to detect *timestamp* vulnerabilities in test smart contract codes. In order to verify the effectiveness of HomoDec to existing smart contract vulnerability detection techniques, we design and perform a large-scale experimental study. We choose the source codes from the widely used SmartBugs Wild Dataset [6] and 9 state-of-the-art smart contract vulnerability detection techniques to conduct a comparison. The experimental results verify the effectiveness of our proposed method HomoDec.

[†]The University of Queensland, St Lucia QLD 4072, Australia.
[††]No.83 Army Joint and Truma Disease Treatment Centre of PLA, Xinxiang 453000, China.
[†††]School of Computer Science and Engineering, Xian University of Technology, Xian 710000, China.
[††††]College of Computer and Information Engineering, Shanghai Polytechnic University, Shanghai 200127, China.
a) E-mail: zz8477@126.com (Corresponding author)

2. Background

Smart contracts, as a type of decentralized application written in high-level languages, compiled into bytecode, and executed on the blockchain, inevitably face various security threats closely associated with the runtime environment [7]. Ethereum relies on blockchain as its fundamental supporting technology. It supports the execution and invocation of smart contracts through the Ethereum Virtual Machine (EVM) [8]. Much work has been done to systematically study the types of vulnerabilities in smart contracts, with notable examples such as the work by Zhang et al. [9]. They classified smart contract vulnerabilities into 9 categories based on the IEEE software anomaly standard, including data type vulnerabilities, description type vulnerabilities, environment type vulnerabilities, interaction type vulnerabilities, logic type vulnerabilities, performance type vulnerabilities, security type vulnerabilities, and standard type vulnerabilities, which were collected from multiple sources. Additionally, Ni et al. [10] categorized smart contract vulnerabilities based on their operational mechanisms, dividing them into three major categories: at the high-level language level, at the virtual machine level, and at the blockchain level.

Different studies have different classifications for smart contract vulnerabilities, with occurrences of these vulnerabilities involving issues such as non-compliant control, characteristics of the programming language itself, and unreasonable use of variables or keywords in programming. Besides affecting normal functionality, these vulnerabilities can also pose significant risks on the financial front. The *timestamp* vulnerability in smart contract is a relatively serious vulnerability introduced at the blockchain level, mainly related to the characteristics of the blockchain itself [11]. Once the *timestamp* vulnerability is maliciously exploited by miners, it can lead to very serious consequences on the financial front. Specifically, *timestamp* vulnerability refers to using strict block timestamps in smart contract code to make decisions about behavior control. When a timestamp dependency exists in the code, miners can construct malicious timestamps within the specified timestamp range to intentionally bypass designed restrictions, thus carrying out malicious actions and causing severe consequences. Fig. 1 shows an example of a smart contract with a *timestamp* vulnerability. In line 8, the timestamp *block.timestamp* is assigned to the variable *number*; in line 28, the variable *winNum* depends on the timestamp (*blockhash* and *number*); in line 29, *winNum* is used as a condition, allowing miners to calculate timestamps advantageous to themselves in advance and set timestamps during mining to delay or expedite user self-destruct operations. If miners accelerate user self-destruction, all cryptocurrencies held by the user will be frozen, resulting in monetary losses.

```
1 contract Lottery
2 {
3         mapping (address => uint) usersBet;
4         mapping (uint => address) users;
5         uint nbUsers = 0;
6         uint totalBets = 0;
7         address owner;
8         number=block.timestamp;
9         function Lottery()
10       {
11              owner = msg.sender;
12        }
13        function Bet() public payable
14        {
15               if(msg.value > 0){
16                    if(usersBet[msg.sender] == 0){
17                            users[nbUsers] =
 ↪  msg.sender;
18                            nbUsers += 1;}
19                     usersBet[msg.sender] +=
 ↪  msg.value;
20        totalBets = totalBets + msg.value;
21                }
22         }
23         function EndLottery() public
24         {
25                if(msg.sender == owner)
26                {
27                       uint sum = 0;
28           uint winNum
 ↪  =uint(block.blockhash(number-1)%totalBets+1);
29           for(uint i = 0; i < nbUsers; i++)
30                       {
31                                 sum
 ↪  +=usersBet[users[i]];
32                                 if(sum >= winNum){
33
 ↪  selfdestruct(users[i]);
34                                    return;}
35                        }
36                 }
37         }
38}
```

Fig. 1 An Example of a Contract with Timestamp Vulnerability.

3. Approach

The workflow of HomoDec includes three steps, which are code preprocessing, code feature construction, and code homogeneity detection based on similarity of code representation vectors.

Code preprocessing. We adhere to the contract structure hierarchy specification of Solidity to ensure that code preprocessing can represent the complete semantic information of contract vulnerabilities and preserve the complete structure hierarchy. By using code preprocessing techniques, we can effectively eliminate unused code and functions in the contract, thereby reducing the interference of data unrelated to vulnerabilities on the model. At the same time, this technique can represent as much semantic information as possible with minimal data length and minimize the impact of information loss caused by trimming on model performance. To achieve this, we have constructed a code transformation tree. The purpose is to follow the unified standards defined by the specification tree when trimming smart contract code. This tree is used to transform the input source code for better feature extraction. The code transformation tree is illustrated in Figure 2, and its functionality is to categorize all smart contract keywords and label them as (”Punctuation”, ”Keyword”, ”Aa-Az,0-9”, ”Operators”, ”User-defined”) using a classification method. For instance, punctuation symbol ”;” is labeled as ”Punctuation”, operator ”=” is labeled as ”Operators”, keyword ”public” is labeled as ”Keyword”, alphanumeric ”a” is labeled as ”Aa-Zz,0-9” and user-defined type ”balanceOf” is labeled as ”User-defined”.

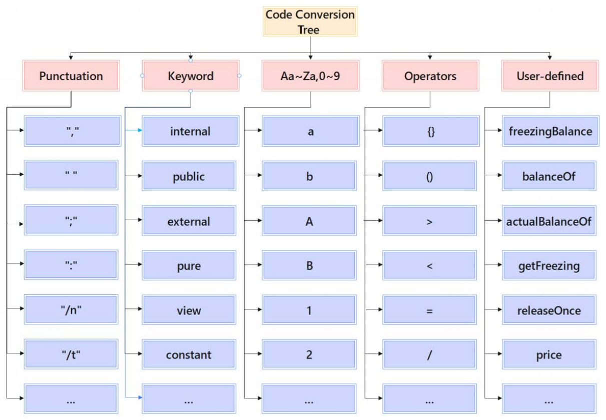


Fig. 2 Code transformation tree.

Code feature construction. After code pre-processing, we construct representation vectors for smart contract code in order to enable the model to capture syntax and semantic information more comprehensively. This allows the model to feature representations for *timestamp* vulnerabilities. Our insight is that the potential capabilities of pre-trained models may offer a new perspective probably benefiting representation construction for smart contracts. Consequently, we tailor a pre-trained model named CodeBert [12] specifically for code of smart contract. As is illustrated in Fig. 3, We follow CodeBert [12], which is based on Transformer neural architecture [13] for programming language, and use a Linear layer to output the result. Specifically, We first concatenate the token sequence and position sequence of tokens into a sequence $I = \{[CLS], T, [SEP], PT\}$, where $[CLS]$ is a special token in front of the two sets and $[SEP]$ is a special notation to split the token sequence T and the position sequence of tokens PT. Our second insight is that the *timestamp* vulnerabilities are caused by propagation of variables. Thus, the second step is to extract variable sequence set and position sequence set of variables for the *timestamp* vulnerabilities. We parse the source code into an Abstract Syntax Tree (AST), then extract data flow relationships from the AST, and finally transform them into crucial data flow graph according to critical information, i.e., timestamp (*block.timestamp, block.number, now*) assignment statement. The AST encompasses syntax details of the code, with terminals (leaves) serving the purpose of identifying the variable sequence. We then concatenate the variable sequence and position sequence of variables into another sequence $I' = \{[CLS], V, [SEP], PV\}$. Finally, the two sequences I and I' are concatenated together to be the input into the model. The input vector goes through masked multi-head attention layers, layer normalization layers, several transformer layers and linear layers to generate the output, which is the representation of the vulnerable code.

Code homogeneity detection. In this stage, we compare the similarity of the code representation vectors generated by the last step to determine the homogeneity between the test codes and the ones in vulnerability code base and detect *timestamp* vulnerabilities. We utilize cosine similarity calculation formula to calculate the similarity between two code representations. In equation 1, n is the dimension of the representation vectors, a means vector of the test code and b denotes vector of the code in vulnerability code base. We conduct similarity measurements on the contracts in the extracted dataset and observe that the similarity values of most smart contract codes are concentrated between 0.4 and 0.7. Zhou et al. [14] detected code cloning phenomena based on fuzzy hashing, and their results align with those of this study. Therefore, this paper considers code to be tested, with a similarity greater than 0.7 to the code in vulnerability code base, as containing *timestamp* vulnerabilities.

$$Similarity = \frac{\sum_{i=0}^{n} (a_i \times b_i)}{\sqrt{\sum_{i=0}^{n} (a_i)^2} \times \sqrt{\sum_{i=0}^{n} (b_i)^2}} \tag{1}$$

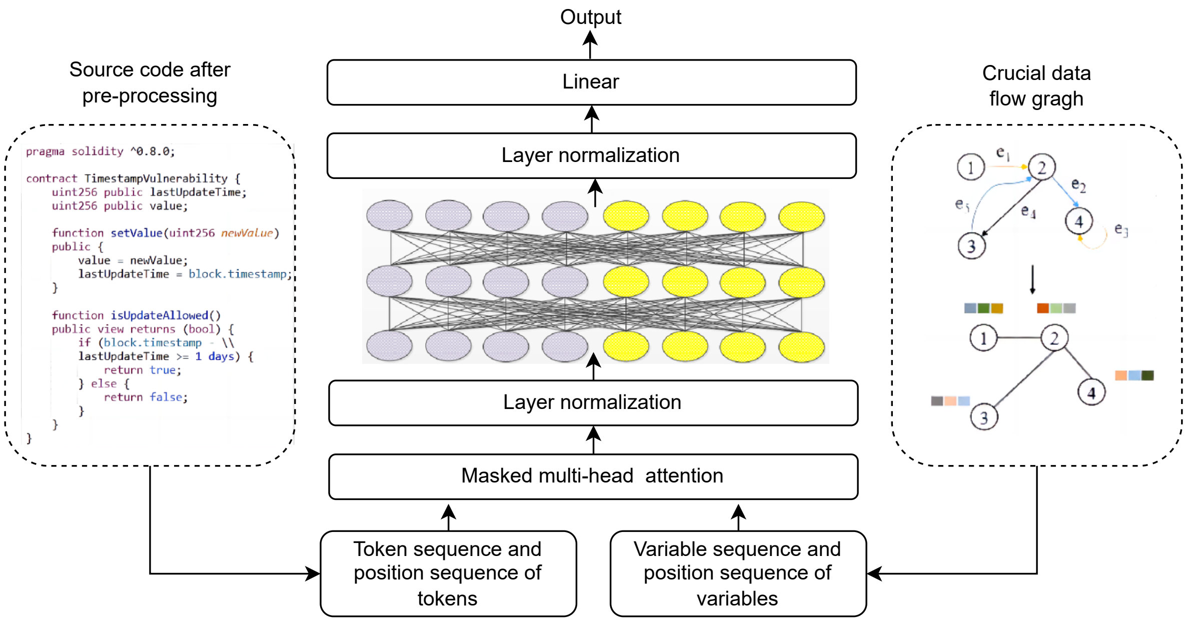
4. An experimental study

4.1 Experimental Setup

We utilize the SmartBugs Wild Dataset [6] as our benchmark, a recently published extensive dataset focusing on vulnerabilities in smart contracts written in the Solidity language. It encompasses 47,398 distinct and authentic .sol files, comprising approximately 200,000 contracts in total. It’s noteworthy that each .sol file may contain one or more contracts. These contracts cover the fields of finance, supply chain, Internet of things, healthcare, digital identity authentication, etc. In our experimental study, we randomly choose 30% of the vulnerable contract as vulnerability code base and 70% as the test codes. we compare HomoDec with 9 state-of-the-art smart contract vulnerability detection methods, which include Manticore [15], Osiris [16], Mythril [17], Oyente [18], SmartCheck [19], GCN [20], Vanilla-RNN [20], LSTM [20] and GRU [20]. The experimental setup comprises a system equipped with an Intel I7-9700 CPU and 64GB of RAM, featuring a NVIDIA TITAN X Pascal GPU with a 12GB capacity. The operating system utilized is Ubuntu 18.04, and data analysis is carried out using MATLAB R2016b.

4.2 Evaluation Metrics

Regarding the assessment criteria, we employ the commonly


Output
Linear
Layer normalization
Layer normalization
Masked multi-head attention
Token sequence and position sequence of tokens
Variable sequence and position sequence of variables
Source code after pre-processing
Crucial data flow gragh
e1
e2
e3
e4
e5
1
2
3
4

```
pragma solidity ^0.8.0;

contract TimestampVulnerability {
    uint256 public lastUpdateTime;
    uint256 public value;

    function setValue(uint256 newValue)
    public {
        value = newValue;
        lastUpdateTime = block.timestamp;
    }

    function isUpdateAllowed()
    public view returns (bool) {
        if (block.timestamp - \\
        lastUpdateTime >= 1 days) {
            return true;
        } else {
            return false;
        }
    }
}
```

Fig. 3 Workflow of code feature construction.

Table 1 Performance comparison of the involved approaches in terms of Recall, Precision and F1 score.

Method	Timestamp		
	Recall(%)	Precision(%)	F1(%)
Manticore	50	57	50
Osiris	51	53	52
Mythril	51	56	51
Oyente	52	55	53
SmartCheck	50	74	51
GCN	76	68	72
Vanilla-RNN	45	52	46
LSTM	59	50	54
GRU	59	49	54
HomoDec	**84**	**78**	**79**

utilized metrics of ”Precision,” ”Recall,” and ”F1-score” [19] as shown in equation 2, equation 3 and equation 4. We opt for the ”macro” approach in our evaluation, which entails calculating metric values separately for contracts with and without vulnerabilities. Subsequently, the average values across these two categories are computed to derive the final result. This methodology enables us to gauge the overall performance of our approach.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{2}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3}$$

$$F1 - score = 2 \times Precision \times \frac{Recall}{Precision + Recall} \tag{4}$$

4.3 Experimental Results

From the quantitative results of Table 1, we have the following observations. First, we found that the existing tools have not yet achieved satisfactory results in *timestamp* vulnerability detection, *e.g.* among baseline approaches, the highest Recall is 76% on *GCN* while the lowest Recall is 45% on *Vanilla-RNN*, the average Recall is only 54.78%. the highest precision is 74% on *SmartCheck* while the lowest precision is 49% on *GRU*, the average precision is only 57.11%. The highest F1-score is 72% on *GCN* while the lowest F1-score is 46% on *Vanilla-RNN*, the average precision is only 53.67%. Secondly, HomoDec outperforms existing approaches to a large extent. More specifically, HomoDec achieves a precision of 78%, improving the state-of-the-art up to 59.18%. With respect to the recall, it improves the state-of-the-art up to 86.67%. In addition, the F1 score of HomoDec is 79% higher than the maximum of the existing techniques, showing that HomoDec achieves a significant improvement with respect to the overall performance.

5. Threats to Validity

The experimental study employs the widely recognized SmartBugs Wild Dataset as the representative dataset for smart contract vulnerability detection, as established in prior work [6]. However, it is essential to acknowledge the presence of numerous unknown and intricate factors in real-world development scenarios. The method proposed in this paper may not comprehensively cover or be universally applicable to all situations. Consequently, future efforts will involve validating the effectiveness of HomoDec using a more diverse set of real smart contract programs, thereby reinforcing the robustness of the experimental results.

To address potential implementation bugs in various comparison methods and HomoDec, we meticulously implemented them based on publicly available source code and previous research. Subsequently, we verified the correctness of these methods using handcrafted test cases to mitigate the risk of errors. In evaluating the effectiveness of various *timestamp* vulnerability detection methods, the experimental study relies on metrics such as Recall, Precision, and F1-score. Given the widespread acceptance and use of these evaluation measurements, the associated validity threats can be considered negligible.

This paper represents a preliminary exploration of smart contract vulnerability detection based on code homogeneity. Due to the substantial workload involved, we only delved into detecting *timestamp* vulnerabilities in the experimental study. Subsequent efforts will focus on exploring other types of vulnerabilities.

6. Conclusion

This paper introduces an automated approach named HomoDec, designed for the detection of *timestamp* vulnerabilities in smart contracts. HomoDec leverages pre-training techniques and examines the code homogeneity of smart contracts. In contrast to existing methods, our approach not only takes into account value dependencies among program variables but also prioritizes critical information relevant to vulnerabilities. Furthermore, we investigate the viability of employing pre-trained models for vulnerability detection. Through extensive experiments, we demonstrate that our approach surpasses state-of-the-art methods in performance. This work represents a significant advancement in showcasing the potential effectiveness of code homogeneity detection approaches for smart contract vulnerability detection tasks. Future research can explore the following aspects: leveraging larger-scale and more diverse datasets for validation; employing advanced and robust model architectures and optimization methods to enhance annotation performance; and further refining and tuning model parameters to optimize the generation effectiveness of representation vectors.

Acknowledgments

This work is partially supported by China Postdoctoral Science Foundation (Certificate Number:2023M732594) and Characteristic Innovation Project of Ordinary University in Guangdong Province (Certificate Number:2023KTSCX193)

References

[1] J. Brito and A. Castillo, Bitcoin: A primer for policymakers, Mercatus Center at George Mason University, 2013.
[2] S. Bhatia and S. Tyagi, “Ethereum,” Blockchain for Business: How It Works and Creates Value, pp.77–96, 2021.
[3] M. Wohrer and U. Zdun, “Smart contracts: security patterns in the ethereum ecosystem and solidity,” 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp.2–8, IEEE, 2018.
[4] S.S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.N. Lee, “Systematic review of security vulnerabilities in ethereum blockchain smart contract,” IEEE Access, vol.10, pp.6605–6621, 2022.
[5] Q.U. Ain, W.H. Butt, M.W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” IEEE access, vol.7, pp.86121–86144, 2019.
[6] J.F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, “Smartbugs: A framework to analyze solidity smart contracts,” 2020.
[7] 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.
[8] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” Ethereum project yellow paper, vol.151, no.2014, pp.1–32, 2014.
[9] P. Zhang, F. Xiao, and X. Luo, “A framework and dataset for bugs in ethereum smart contracts,” 2020 IEEE international conference on software maintenance and evolution (ICSME), pp.139–150, IEEE, 2020.
[10] Y. Ni, C. Zhang, and T. Yin, “A survey of smart contract vulnerability research,” Journal of Cyber Security, vol.5, no.3, pp.78–99, 2020.
[11] L.M. Bach, B. Mihaljevic, and M. Zagar, “Comparative analysis of blockchain consensus algorithms,” 2018 41st international convention on information and communication technology, electronics and microelectronics (MIPRO), pp.1545–1550, Ieee, 2018.
[12] 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.
[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol.30, 2017.
[14] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” Proceedings of the second ACM conference on Data and Application Security and Privacy, pp.317–326, 2012.
[15] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.1186–1189, IEEE, 2019.
[16] C.F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” Proceedings of the 34th annual computer security applications conference, pp.664–676, 2018.
[17] B. Mueller, “Mythril-reversing and bug hunting framework for the ethereum blockchain,” 2017.
[18] L. Luu, D.H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp.254–269, 2016.
[19] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, pp.9–16, 2018.
[20] 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.