Static Analysis of Integer Overflow of Smart Contracts in Ethereum

Enmei Lai
School of Computer Science and Technology
Chongqing University of Posts and Telecommunications, Chongqing, China
laienmei96@163.com

Wenjun Luo
School of Cyber Security and Information Law
Chongqing University of Posts and Telecommunications, Chongqing, China
luowj@cqupt.edu.cn

ABSTRACT
In recent years, vulnerabilities of smart contracts have frequently break out. In particular, integer overflow of smart contracts, a high-risk vulnerability, has caused huge financial losses. However, most tools currently fail to detect integer overflow in smart contracts. In this paper, we summarize 11 types of integer overflow features for Solidity smart contracts in Ethereum and abstractly define 83 corresponding XPath patterns. And we design an extensible static analysis tool to detect common integer overflow vulnerabilities of Solidity smart contracts in Ethereum through the defined XPath patterns. To evaluate our tool, we tested 7,000 verified Solidity smart contracts and found that there were 430 smart contracts with vulnerabilities of integer overflow. Experimental results show that there are still high-risk vulnerabilities of integer overflow in verified smart contracts.

CCS Concepts
•Security and privacy→Software security engineering; •Software and its engineering → Software testing and debugging

Keywords
smart contract; integer overflow; XPath; vulnerability detection

1. INTRODUCTION
The concept of smart contract was first proposed by Nick Szabo in 1995 [1], which allows for trusted transactions without a third party. The essence of smart contract is the computer program [2]. It runs on the blockchain, greatly enriching the function of the blockchain, making the blockchain not only a distributed ledger database, but also able to complete a certain degree of business processing. The Ethereum smart contract runs on the EVM and is finally deployed on the blockchain, so once deployed successfully, it cannot be modified and will be stored in the blockchain [3]. If there is a security vulnerability in the smart contract, the attacker will use it, which is likely to lead to a devastating disaster [4].

ICCSP 2020, January 10–12, 2020, Nanjing, China.
ACM ISBN 978-1-4503-7744-7/20/01…$15.00.
DOI: https://doi.org/10.1145/3377644.3377650

In recent years, the security issues of smart contracts have frequently emerged. For example, the DAO attack in June 2016 caused a loss of $60 million. In July 2017, the vulnerability of Parity Wallet led to $30 million loss and the freezing of $150 million worth of Ether. And the vulnerability in BEC smart contract caused the worth of tokens to be almost zero in April 2018. Each attack caused huge losses. Therefore, the security of smart contracts faces enormous challenges [5,7]. At present, relevant research has been carried out at home and abroad, and some tools for detecting security vulnerabilities in smart contracts have been proposed. However, most tools cannot detect integer overflow vulnerabilities and charge. Integer overflow is a common high-risk vulnerability, which caused the BEC attack with a huge loss.

This paper studies and analyzes the integer overflow of Solidity smart contract in Ethereum (referred to as smart contract). On this basis, we summarize 11 kinds of integer overflow features, and abstractly represent as 83 corresponding XPath patterns. Then, an extensible static detection tool for integer overflow of smart contract is designed based on the idea of SmartCheck tool [6]. Our tool can detect the common smart contract integer overflow vulnerability with strong expansibility and high detection efficiency.

2. THE CHARACTERISTICS OF INTEGER OVERFLOW FOR SOLIDITY SMART CONTRACT IN ETHEREUM

2.1 Basic Principles of Integer Overflow
The integer data type[1] of the Ethereum Smart Contract has a fixed size, and its step size is incremented by 8 bits, i.e., $uint8$ to $uint256$, $int8$ to $int256$. Each integer type has a range, so each integer variable can only store numbers within the range of its data types. If this range is exceeded, it will cause an overflow. As shown in Figure 1, a $uint8$ type can only store numbers between 0 and $2^8 - 1$. If you store 256 into a $uint8$ type, the 256 will eventually become zero. If you store -1 into the $uint8$ type, -1 will eventually become the maximum value of 255.

In general, the types of integer overflow in smart contracts are divided into multiplication overflow, addition overflow, and subtraction overflow [8]. This paper summarizes integer overflow features based on integer overflow vulnerability that exposed and the principle of integer overflow. The specific features are described below.

[1] https://solidity.readthedocs.io.

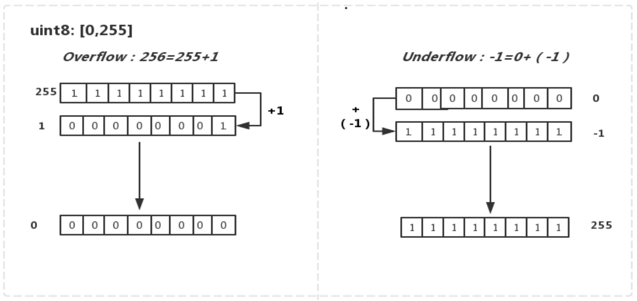


Figure 1. Basic principles of integer overflow

2.2 Multiplication Overflow

Multiplication overflow is caused by a multiplication operator without checking whether the multiplication operation is out of range. This study summarizes three different types of multiplication overflow features.

(1) The source codes of smart contract contain the statement that integer variable is equal to the integer parameter or $msg.value$ multiplied by another arbitrary form of integer data, and do not contain any form of statement that checks whether its operation is out of range. One of the forms of feature codes for this type is $uint256\, amount = len * \_value$, where $\_value$ is an integer parameter or $msg.value$. None of the types of features in this paper contain any form of statement that checks whether its operation is out of range. The features contain an integer parameter or $msg.value$, because the integer parameter can be manipulated manually when the function is called. The attacker can construct a large parameter value so that the results of the arithmetic operation exceed the range. Similarly, $msg.value$ is controllable. All types of feature codes in this paper are placed in here.[2]

(2) The source codes of smart contract contain the statement that the integer parameter is multiplied by another integer variable whose result is less than or equal to the integer state variable or the balance of the contract or an account address. The integer variable contains a state variable or a local variable, or source codes contain the reverse statement above. One of the forms of the feature codes is $require\,(tokenLimit >= \_amount * p)$. $\_amount$ is an integer parameter. $tokenLimit$ and p are the integer variable.

(3) The source codes of smart contract include the statement that the integer variable multiply equal to integer parameter, or that the state variable of the mapping type multiply equal to integer parameter. The mapping type includes mapping address types to integer and address types to mapping types, where the address type can be an address type parameter, variable or $msg.sender$. One of the forms of feature codes for this type is $mount *= \_value$, where $\_value$ is the integer parameter.

2.3 Addition Overflow

Based on the principle of integer overflow, there are four types of addition overflow features.

(1) The source codes of smart contract contain a statement that the variable is equal to the integer parameter plus another integer variable other than itself. One of the forms of the feature codes is $amount = p + \_value$, where $\_value$ is an integer parameter.

(2) The source codes of smart contract contain the statement that the integer variable or the state variable of mapping type is equal to itself plus the integer parameter or the variable whose value is equal to $msg.value$ or integer parameters multiplied by integer variable. $balances[target] = balances[target] + newToken$ is one of the feature codes forms. $newToken$ is an integer parameter, or $newToken$ is equal to the result of $msg.value * p$, or $newToken$ is equal to the result of $\_value * p$, where $\_value$ is an integer parameter.

(3) The source codes of smart contract include the statement that the integer variable or the state variable of the mapping type plus equal to integer parameter. One of the forms of its feature codes is $allowed[target][msg.sender] += newToken$.

(4) The source codes of smart contract contain the statement that the integer parameter or the variable whose value is equal to $msg.value$ or integer parameters multiplied by integer variable plus another integer variable whose result is less than or equal to the integer state variable or the balance of the contract or an account address. Or source codes contain the reverse statement above. $require(this.balance >= total + newToken)$ is one of the feature codes forms. $this.balance$ can also be the variable or $address.balance$.

2.4 Subtraction Overflow

According to the similar method described above, four different types of subtraction overflow characteristics are obtained.

(1) The three types of subtraction overflow features are similar to the first three types of addition overflow features by simply replacing the addition operator with the subtraction operator. But the form of checking its range is inconsistent.

(2) The loop body of the smart contract source codes contains the statement that the integer variable or the state variable of the mapping type minus equal to the number or an integer constant.

In general, the integer overflow features of smart contracts include addition or subtraction or multiplication operator and integer variables. one of the integer variables can be manipulated manually, such as parameters, $msg.value$. And there are no statements that check if their operations are out of range.

3. STATIC ANALYSIS OF INTEGER OVERFLOW IN SMART CONTRACTS

For the analysis of Section 2, this paper abstractly defines the corresponding XPath patterns according to the 11 types of integer overflow features summarized above. Based on the idea of SmartCheck [6], this paper designs an optimized and extensible static detection tool for integer overflow of Solidity smart contract in Ethereum based on XPath patterns defined.

As shown in Figure 2, the main architecture of the tool is divided into three major modules, including: (1) ANTLRV4; (2) matching detector; (3) XPath patterns.

First of all, we define the XPath patterns abstractly based on the vulnerability characteristics above to form a rule library. When the user detects source codes of the smart contract, the tool performs lexical analysis and syntax analysis on the source codes, and generates abstract syntax tree, and then converts it into the XML-based intermediate representation. Then XPath expression is extracted by traversing the rule library to query and locate the matching nodes in the XML-based intermediate representation. Finally, the numbers of rows in the source codes are relocated to form the vulnerability analysis report.

[2] https://github.com/L-PLUM/ExperimentData.

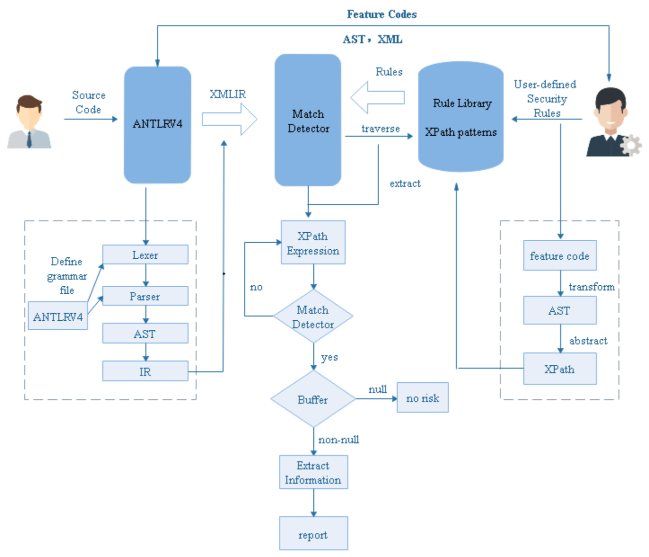


Figure 2. Architecture of the tool

3.1 ANTLRV4

ANTLRV4[3] is a grammar analysis generator based on LL(*) grammar that generates lexical analysis programs and syntax analysis programs through user-defined grammar rules. At the same time, ANTLRV4 can also generate abstract syntax trees and intermediate representations. In Figure 2, given the user-defined grammar rules that recognize the Solidity language, the ANTLRV4 tool will generate a lexical analyzer and parser for Solidity and generate the abstract syntax tree. The abstract syntax tree is traversed by the listener and converted into the XML-based intermediate representation.

3.2 Matching Detector

The tool uses custom XPath patterns to detect integer overflow vulnerabilities in smart contracts. When detected, the source codes of smart contract are first converted to XML-based intermediate representation. At the same time, each rule is traversed in the rule library and the XPath expression is extracted in each rule. And then each XPath expression is executed to query and locate the matching nodes in the XML-based intermediate representation. If the matching node is detected, put it in the temporary storage area. Finally, the storage area is traversed, the vulnerability information of the matching nodes is obtained, and the line numbers are returned to form a vulnerability report.

3.3 XPath Patterns

Self-defined rules determine the types of smart contract security vulnerabilities detected. In order to facilitate matching when the vulnerability is detected, the rule library based on XML data format is also designed in this paper. The rule library contains a number of rules, and each rule includes many XPath patterns. Each rule is a *Rule* node, which contains a child *RuleId* node and another child *Patterns* node. And each child *Patterns* node contains a plurality of child *Pattern* nodes. Each *Pattern* node contains the attribute *patternId* node, the child *Categories* node that represents language type, the child *Severity* node that represents risk level, and the child *XPath* node.

[3] http://www.antlr.org.

In this paper, the integer overflows correspond to three *Rule* nodes. It can be distinguished according to the *RuleId* node whether it belongs to multiplication overflow, addition overflow or subtraction overflow. In order to detect the integer overflow vulnerability more accurately, this paper abstractly represents 11 different types of integer overflow features as 83 different XPath patterns. There are 12 XPath patterns of multiplication overflow, 38 XPath patterns of addition overflow, and 33 XPath patterns of subtraction overflow. Each XPath pattern corresponds to a *Pattern* node and a different *patternId* node. The *patternId* in this paper is a six-character string consisting of overflow type and the number. For example, the *patternId* of the addition overflow is *add*101, *add*102, in order. When smart contract is detected, the expression in the *XPath* node is extracted to implement vulnerability detection. This paper puts these 83 XPath patterns in the above URL, including the rule library, test samples, feature codes, abstract syntax tree and the experimental test contracts. The following describes one of the XPath patterns.

3.3.1 *Case Study of XPath Pattern*

This paper selects the XPath pattern with *patternId* as *sub*102 as a case. According to the analysis in section 2, the core features of *sub*102 are like line 5 in Table 1, but not limited to this form.

Table 1. Core features of Sub102

1	contract Sub102{
2	uint256 amount;
3	function t(uint256 _value) public {
4	// require(amount >= _value);
5	amount = amount - _value; …} …}

First, the feature codes of sub102 are transformed into abstract syntax tree and XML intermediate representation. Figure 3 is only the core part of abstract syntax tree.

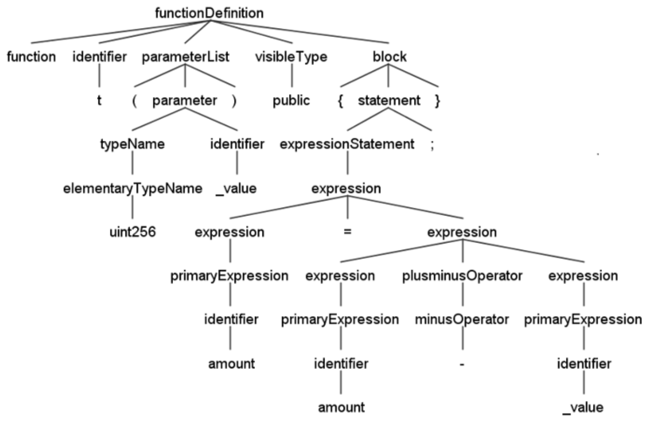
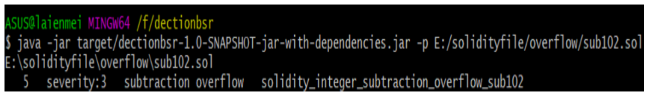


Figure 3. Abstract syntax tree

Then, the XPath pattern of *sub*102 is abstractly defined according to Figure 3 and overflow features. The nodes of XPath expression correspond to nodes of the abstract syntax tree in Table 2.
In Table 2, lines 1 to 5 indicate that the descendant nodes of the *expressionStatement* node contain the *expression* node whose descendant text node (referring to *amount*) is equal to the text node of the ancestor *variableDeclaration* node or the text node of the ancestor *stateVariableDeclaration* node, and contain an *expression* node whose descendant node is the *minusOperator*

node, and contain the node whose text node is equal to itself, and contain the node whose text node is equal to the integer parameter. This abstractly represents line 5 in Table 1. Similarly, lines 6 to 8 of Table 2 abstractly represent line 4 in Table 1. Generally, XPath expressions only contain the names of rule nodes, and abstractly represent feature codes, without involving specific source codes of smart contracts. At the same time, the forms of checking operations are diverse. Line 4 in Table 1 is only one of them.



```
ASUS@laienmei MINGW64 /f/dectionbsr
$ java -jar target/dectionbsr-1.0-SNAPSHOT-jar-with-dependencies.jar -p E:/solidityfile/overflow/sub102.sol
E:\solidityfile\overflow\sub102.sol
   5   severity:3   subtraction overflow   solidity_integer_subtraction_overflow_sub102
```

Figure 4. Results of the detection

Finally, in order to verify the correctness of the XPath pattern above, we firstly test the above XPath expressions through the XMLSpy2013 tool, and detect the corresponding smart contracts. Figure 4 shows the results of the detection. It can be seen that the expected vulnerability can be detected. In order to improve the accuracy of detection, this paper finally tests a large number of smart contracts, and further optimizes the XPath patterns.

Table 2. Core part of the XPath pattern of sub102

1	<XPath> //expressionStatement[expression[text()="="] and
2	expression/expression[1]/primaryExpression/identifier[text()[1]=(ancestor::functionDefinition//variableDeclaration[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/identifier) or text()[1]=(ancestor::sourceUnit//stateVariableDeclaration[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/identifier)] and
3	expression/expression[2]/plusminusOperator/minusOperator and
4	expression/expression[2]/expression[1]/primaryExpression/identifier[text()[1]=(parent::*/parent::*/parent::*/preceding-sibling::expression/primaryExpression/identifier)] and
5	expression/expression[2]/expression[2]/primaryExpression/identifier[text()[1]=(ancestor::functionDefinition/parameterList/parameter[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/idntifier)]]
6	[not(ancestor::functionDefinition//expression[text()[1] = "&gt;" or text()[1] = "&lt;" or text()[1] = "&lt;=" or text()[1] = "&gt;="]
7	[expression/primaryExpression/identifier[text()[1]= (ancestor::functionDefinition//expressionStatement/expression[expression/plusminusOperator/minusOperator]/expression[2]/expression[1]/primaryExpression/identifier) and (text()[1]=(ancestor::functionDefinition//variableDeclaration[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/identifier) or text()[1]= (ancestor::sourceUnit//stateVariableDeclaration[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/identifier))]]
8	[expression/primaryExpression/identifier[text()[1]= (ancestor::functionDefinition/parameterList/parameter[typeName/elementaryTypeName[matches(text()[1],"uint|int")]]/identifier) and text()[1]=(ancestor::functionDefinition//expressionStatement/expression[expression/plusminusOperator/minusOperator]/expression[2] /expression[2]/primaryExpression/identifier)]])] </XPath>

4. EXPERIMENT RESULTS

4.1 Experiment Setup

In this experiment, a command line tool is developed to detect integer overflow vulnerabilities for Solidity smart contracts. It runs in the following environments: Intel i7-7500 CPU, 16GB of memory, maven, JDK 1.8, ANTLR 4.7. we used JSoup crawler to crawl 2,500 source codes of the smart contracts as datasets on Etherscan. Due to the update of Etherscan, only the latest 500 smart contracts were displayed. In order to test more smart contracts, this paper also crawled 1500 validated source codes of the smart contracts from Ropsten, Kovan and Rinkeby as test datasets.

4.2 Experiment Results and Analysis

In order to evaluate our tool, this paper used it to detect the source codes of 7,000 smart contracts with different contract addresses. In order to improve the accuracy of detection and reduce the false positive rate and false negative rate, this paper further optimized the XPath patterns based on the analysis of the detection results. Through the multiple optimization and improvement of the XPath patterns, the accuracy of the detection results is almost 100%. There may be some unknown form of integer overflow types, so there may be some false negatives. The accuracy rate of detection here is for the integer overflow types given in this paper. Finally, this paper used the optimized tool to detect the source codes of 7,000 smart contracts, and made analysis and statistics on the test results.

Table 3. Statistics of detection results

vulnerability Type	Findings	% of all
Mul Overflow	109	1.56%
Add Overflow	218	3.11%
Sub Overflow	103	1.47%

As can be seen from Table 3, the tool detected 109 smart contracts with multiplication overflow vulnerabilities, accounting for 1.56% of the total number of smart contracts tested. There are 218 smart contracts with addition overflow vulnerabilities, accounting for 3.11% of the total number of smart contracts tested. There are 103 contracts with subtraction overflow vulnerabilities, accounting for 1.47% of the total number of smart contracts tested. A total of 430 smart contracts with integer overflow vulnerabilities were detected, indicating that there are still integer overflow vulnerabilities in the verified smart contracts. The integer overflow vulnerability is a high-risk vulnerability, and there are potentially huge risks for the verified smart contracts.

In this paper, we also compare our tool with other smart contract vulnerability detection tools, but most tools currently fail to detect integer overflow vulnerabilities. Table 4 lists three tools that currently cannot detect integer overflow vulnerabilities, such as SmartCheck, Securify, and Oyente, as well as some unlisted tools. The VaaS platform of Chengdu Chain Security can detect integer overflow vulnerabilities. But it cannot complete the detection for complex smart contracts. In addition, the VaaS platform is limited to free use only four times a day, and is not suitable for mass detection. However, our tool supports single source file detection and large-scale source file detection, and the detection efficiency is high.

Table 4. The tools of failing to detect integer overflows

vulnerability Type	Smart Check	Securify	Oyente	Our tool
integer Overflow	×	×	×	√

4.2.1 *Case Analysis of Optimizing XPath Patterns*

Table 3 shows the detection results after optimizing the XPath patterns, but before the optimization, there were false positives. As shown in Table 5, there are the core codes for the case of false positives before optimizing.

In line 6 of Table 5, before XPath patterns optimization, the tool detects an error. But there is no integer overflow vulnerability in this line, which is a case of false positives. This paper originally defined the XPath patterns without considering the case of assigning $allowed[\_from][msg.sender]$ to another variable. Therefore, in view of this feature, we optimize it on the basis of the original, adding the judgment of this situation, that is, excluding the expression in the form of line 5 in Table 5. According to the characteristics of this false positive situation, this paper also similarly optimizes other similar XPath patterns. At the same time, after multiple tests and optimizations, this paper has optimized all false positives, making the correct rate of detection 100%.

Table 5. Case of false positives before optimizing

1	contract C {
2	mapping(address=>mapping(address=>uint256)) public allowed;
3	function t(address _from, uint256 _v) public {
4	uint256 allowance = allowed[_from][msg.sender];
5	require(allowance >= _v);
6	allowed[_from][msg.sender] -= _v; … } …}

5. RELATED WORK

At present, researchers at home and abroad have proposed some solutions for the security of smart contracts. Luu et al. [9] introduced a symbolic execution tool, called Oyente, to detect potential security vulnerabilities, but the scalability of this tool is poor. Liu Chao et al. [10] introduced an analyzer based on fuzzy test, called ReGuard. It can dynamically recognize reentrancy vulnerabilities based on the results of runtime tracing. However, it only detects reentrancy vulnerabilities and is not easy to extend. Bo et al. [11] developed a new fuzzy detector, ContractFuzzer, which generates fuzzy input according to the ABI specification of the smart contract, and then forms a security vulnerability report based on the logs recorded by the EVM. But there are also certain false positives and false negatives. It is also not easy to form fuzzy input. Tsankov et al. [12] introduced Securify, a symbol-based abstraction tool that demonstrates whether smart contract behavior is safe or unsafe for a given attribute. However, the time of detection is too long. Tikhomirov et al. [6] proposed SmartCheck, a static analysis tool that can detect smart contract security vulnerabilities. However, SmartCheck cannot detect integer overflow vulnerabilities and report some unnecessary information.

6. CONCLUSION

In this paper, we summarize 11 different types of integer overflow features and abstractly define them as 83 corresponding XPath patterns by studying and analyzing the integer overflow of the Solidity smart contract in Ethereum. On this basis, an extensible static detection tool was designed to implement the detection of integer overflow of Ethereum smart contract. We tested our tool on 7,000 verified smart contracts and detected 430 smart contracts with vulnerabilities of integer overflow. The experimental results show that there are still integer overflow vulnerabilities in the verified smart contracts. Our tool has the characteristics of high detection efficiency and strong expansibility. However, some unknown integer overflow vulnerabilities cannot be detected.

In future work, we will study more vulnerabilities types of the Solidity smart contract in Ethereum and add more detection patterns. Furthermore, we will study the security vulnerabilities of smart contracts for other languages and expand our tool to support smart contracts detection in multiple languages.

7. ACKNOWLEDGMENTS

We thank anonymous reviewers for their comments on the paper. We thank Sihao Xue for useful discussions and feedback on the early version of the paper.

8. REFERENCES

[1] Szabo N. 1996. Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought.

[2] Bhargavan K, Delignat-Lavaud A, Fournet C, et al. 2016. Formal verification of smart contracts. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16. 91–96

[3] Hirai Y. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. International Conference on Financial Cryptography and Data Security. Springer, Cham, 520-535.

[4] Chen Ting, Li Xiaqi, Luo Xiapu, et al. 2017. Underoptimized smart contracts devour your money. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).442–446.

[5] Atzei N, Bartoleti M, and Cimoli T. 2017. A Survey of Atacks on Ethereum Smart Contracts (SoK). In International Conference on Principles of Security and Trust. Springer, 164–186.

[6] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, et al. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In Proceedings of the 1st ACM/IEEE International Workshop on Emerging Trends in Software Engineering for Blockchain. ACM, New York, NY, USA, 8 pages.

[7] Alexander M, Markus F. 2018. Security vulnerabilities in ethereum smart contracts. In Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services. ACM, 375-380.

[8] Torres C, Schütte J, and State R. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In 2018 Annual Computer Security Applications Conference. ACM, New York, NY, USA, 13.

[9] Luu L, Chu D H, Olickel H, et al. 2016. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 254–269.

[10] Chao L, Han L, Zhong C, et al. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In Proceedings of the 40th International Conference on Software Engineering Companion. IEEE Press. 65-68.

[11] Bo J, Ye L, and W.K. C. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering. Montpellier, France,10.

[12] Tsankov P, Dan A, Cohen D, et al. 2018. Securify: Practical Security Analysis of Smart Contracts. arXiv preprint arXiv:1806.01143 (2018).