.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2020.DOI

Function-level dynamic monitoring and analysis system for smart contract

YI DING[1], CHENSHUO WANG[1], QIONGHUI ZHONG[1], HAISHENG LI[2,3*], JINJING TAN[1], JIE LI[1*]

[1]Beijing Wuzi University, Beijing, China
[2]School of Computer Science and Engineering, Beijing Technology and Business University, Beijing, China
[3]National Engineering Laboratory for Agri-Product Quality Traceability, Beijing, China

*Corresponding author: Jie Li (e-mail: lijiebwu@163.com), Haisheng Li (e-mail: lihsh@th.btbu.edu.cn).

This work was supported by National Key Research and Development Program of China under Grant 2018YFB1402703, Science and Technique General Program of Beijing Municipal Commission of Education under Grant KM201910037003, Research Base Project of Beijing Municipal Social Science Foundation under Grant 18JDGLB026 and 20GLB026, Open Project of National Engineering Laboratory for Agri-product Quality Traceability under Grant AQT-2020-YB5, Project of 2020 "Shipei Plan" of Beijing Wuzi University, Beijing Key Laboratory under Grant BZ0211, and Beijing Intelligent Logistics System Collaborative Innovation Center under Grant PXM2018_014214_000009.

ABSTRACT The close integration of blockchain and smart contract technology has become an important foundation for current trusted applications. High-quality, high-efficiency and high-security codes have become basic requirements for smart contract applications because they are not easy to be modified after being deployed on blockchain. This paper proposes a function-level dynamic monitoring and analysis method for smart contract, and implements a prototype system. The method adds a "shadow stack" and related data structures to virtual machine of testing blockchain platform by analyzing the principle of function management with original stack, then monitors the bytecode after code instrumentation, records the function calling relationships as well as the relevant metrics of time, instruction number and gas consumption. The prototype system identifies contract inefficient behaviors using visualization and intelligent analysis methods, then forms a smart contract optimization closed loop through iterative improvement. Finally, the paper verified the high feasibility and applicability of the monitoring and analyzing method as well as prototype system's performance through experiments.

INDEX TERMS smart contract; blockchain; dynamic monitoring; code instrumentation

I. INTRODUCTION

IN 2009, Satoshi Nakamoto proposed Bitcoin [1] and its implementation technology, which is called Blockchain 1.0 and triggers the vigorous development of blockchain. Nowadays, blockchain has become one of the most popular technologies and been widely applied in many fields such as finance, copyright, logistics and so on [2]. Smart contract was presented by Nick Szabo [3] in 1995, which is earlier than blockchain and has built a close relationship with it naturally. Blockchain promotes smart contract development effectively while the integration of smart contract has become a typical feature of Blockchain 2.0. Based on blockchain, smart contract turns to be open and transparent to all participants and does not rely on third-party platform. Once being deployed, smart contract will not be easy to be modified, leading to higher correctness and security requirements compared to traditional programs [4-5].

Smart contract can be used for the application of asset transfer and it is vulnerable to be invaded. With the growing number of smart contract, security risks are also growing, which may cause irretrievable loss [6]. Researchers discovered that 34,200 of 1 million smart contracts have security risks [7], while the DAO attack in June 2016 lost about 50 million US dollars in damage because of smart contract unreliable design [8]. In addition, the Parity wallet smart contract caused severe accidents: Ethers valued approximate 30 million US dollars were stolen [9]. The critical reason for these is the defects in smart contract codes. Therefore, deep understanding, pre-deployment testing and then optimization of smart contract program is essential work.

The Ethereum platform [10] is a typical representative of Blockchain 2.0 and Solidity has become one of the mainstream smart contract languages. This paper takes Solidity as an exemplary case, and introduces a smart contract function-level dynamic monitoring and analysis system named SC-Mon based on source code instrumentation technology. The

function calling relationships as well as the relevant metrics of time, instruction number and gas consumption can be recorded by the instrumented code running on the test blockchain environment, then displayed through visualization or intelligent analysis in order to assist users to quickly understand program execution behaviors and discover various problems such as performance, operation, and defect. The system can iteratively optimize smart contract before final deployment on productive blockchain by checking the availability, legitimacy and efficiency to reduce the security risk, and strengthen its robustness and credibility.

The major contributions of the paper are described as follows:

(1) A set of source code instrumentation method for smart contract is proposed. The traditional program performance monitoring and analysis methods are applied to smart contract codes to dynamically monitor smart contract at function level. The instrumented code running on test blockchain environment can accurately record the dynamic execution behaviors of the smart contract, which provides a novel technical method for the development of optimization and defect identification of smart contract program.

(2) The SCMon system can acquire rich measurement metrics. Beside traditional function calling relationships and execution time, it also extends the metrics of execution instruction number, the cost of resources, the classification of instruction, etc.

(3) The system provides various visual analysis diagrams to help users understand the execution characteristics and calling relationships of functions. An intelligent interface is also introduced to assist users in automatically identifying inefficient and defective behaviors of smart contract, and then improving them.

II. RELATED WORK

In recent years, researchers have made many efforts to detect security vulnerability in smart contract, and proposed some frameworks and tools based on static or dynamic analysis [11-12].

For the security vulnerability detection in smart contract, Slither, SmartCheck, Serurify, Zeus, and SIF are based on static analysis. Slither [13] is a security analysis framework for smart contract, which could convert Solidity code into an intermediate representation called SlithIR. SlithIR can automatically detect contract's vulnerability, and help user to understand as well as examine smart contract codes. SmartCheck is an extensible analysis tool proposed by Tikhomirov et al. [14], which compares the intermediate form XML converted from smart contract with XPath mode. The tool's purpose is main to detect common problems in Solidity code. Securify proposed by Tsankov et al. [15] extracts the semantic information by analyzing contract's dependency graph, then proves whether the contract is safe or not. In addition, Zeus presented by Kalra et al. [16] introduces a framework that uses abstract interpretation and symbolic model checking to analyze source code of the contract. The tool cannot detect "divide by zero" error. A general framework SIF [17] can query, analyze and detect the abstract syntax tree (AST) of smart contract to generate reliable codes. It could build 7 tools as well to analyze smart contract codes, which help users to do customized operation for smart contract, and then to understand, analyze and improve the codes better. Additionally,the calling relationship graph is static and cannot show dynamic behaviors. Slither, SmartCheck, Securify, and Zeus all detect security vulnerability by analyzing source code. Slither and SmartCheck convert the contract source code into intermediate forms, while Securify and Zeus are to analyze semantic information of the contract. SCMon in this paper employs the technology of source code instrumentation and then deploys it on Ethereum Virtual Machine (EVM) to be executed. The calling relationships between the functions and the relevant metrics can be obtained dynamically to understand the running smart contracts better.

Sereum[18], ECFChecker[19] and EasyFlow[20] are security tools using dynamic analysis technology. Sereum proposed by Michael Rodler et al. can protect existing deployed contracts from re-entry attacks, which is based on runtime monitoring and validation. Grossman et al. studied the decidability of dynamically or statically checking if an object is an Effectively Callback Free (ECF) objects and developed a prototype implementation called ECFChecker. EasyFlow published by Gao et al. utilizes track technology based on taint analysis. It establishes additional stack and space in memory to store the marked tainting data by extending go-ethereum to detect overflow flaws in smart contract. The trace log in JSON can detect whether there are flaws by the log analyzer. The E-EVM proposed by Robert Norvill et al. [21] is a prototype tool that dynamically simulates and visualizes the execution of Solidity contract on EVM. It primarily simulates the operations of compiled bytecodes to help users optimize smart contract and understand how EVM and smart contract work, which covers 88.2% of the codes. Fuchen Ma, Ying Fu et al. [22] presents a method for the problem of contract code vulnerability detection by strengthening EVM. EVM will automatically stop unsafe transactions to prevent economical losses, but the types of covered errors are limited. SCMon is not only for security detection, but also to provide useful information for developers to optimize smart contract by monitoring the dynamic execution behaviors and consumptions, which could support a wider range of applications.

In summary, the smart contract function-level dynamic monitoring and analysis system based on code instrumentation is devoted to improving the execution efficiency and resources usage of smart contract code in addition to vulnerability security detection. The bytecodes compiled from instrumented contract codes are performed on blockchain testing system, and then the dynamic function calling relationships and relevant metrics will be required to help developers and researchers to understand the contract better.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

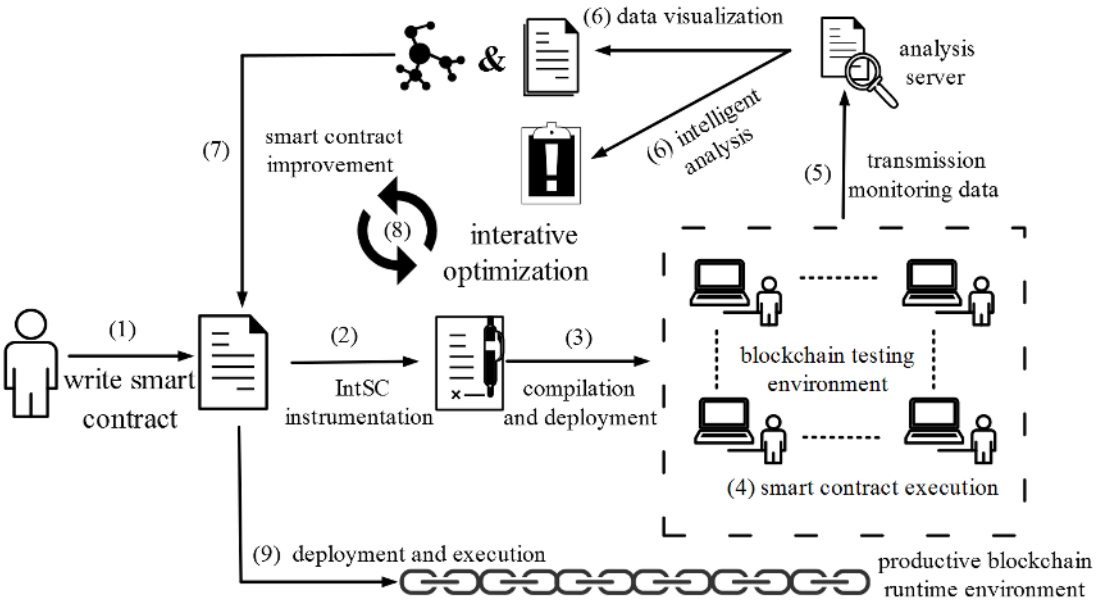

FIGURE 1: The procedure of smart contract optimization loop.

III. SMART CONTRACT MONITORING AND ANALYSIS SYSTEM

This paper presents a smart contract function-level dynamic monitoring and analysis system based on Solidity source code instrumentation. This system can obtain dynamic function calling relationships and relevant metrics through smart contracts running on dedicated monitoring virtual machine, which also provides functions of intelligent automatic analysis and data visualization.

Smart contract is difficult to be upgraded and modified once it is deployed on the blockchain system. The characteristic is conflict with the requirements of iterative optimization. Therefore, smart contract is designed to be deployed on blockchain testing environment in this paper. Then it can be executed, monitored, and iteratively improved. In addition, the blockchain testing environment is often configured similar to productive system for smooth transition. The procedure of smart contract optimization loop is shown as figure 1:

(1)Smart contract developer writes Solidity code;

(2)Smart contract code is instrumented by script IntSC. Then an instrumented Solidity code as well as a file containing all the function names are generated;

(3)The instrumented contract code is compiled to bytecode by SOLC compiler and deployed on blockchain testing environment composed of multiple distributed nodes;

(4)The smart contract is executed on the blockchain testing environment, then the functions' calling relationships and relevant metrics such as instruction execution number, time and gas consumption, are acquired;

(5)The monitoring data is transmitted to the analysis server;

(6)User can visualize the monitoring data and do intelligent analysis on the contract's behaviors;

(7)The smart contract developer optimizes and revises the contract according to the feedback;

(8)The operations above are repeated iteratively until the quality of smart contract satisfies the execution requirements;

(9)The optimized smart contract is deployed on real productive blockchain runtime environment for execution.

The smart contract optimization loop is over.

A. SMART CONTRACT INSTRUMENTATION AND MONITORING MODULE

The research method in this paper is to instrument the source code of smart contract, then monitor and record the dynamic process relied on the EVM mechanism. Figure 2 is the flow chart of monitoring mechanism. The developer utilizes a script IntSC to finish the instrumentation of smart contract source code, and then an instrumented contract code and a file containing all the function names of the contract are generated. The instrumented contract is compiled and deployed on the blockchain testing environment. The bytecode, transaction data, and files containing function names are executed on EVM of the testing environment. At the same time, a "shadow stack" and relevant data structures, which simulate the original EVM stack structure and execution policy, are created to record function names and related data such as function dependency relationship, execution time, instruction number, and gas consumption. The recorded monitoring data is stored in form of JSON and fed back to the user through

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

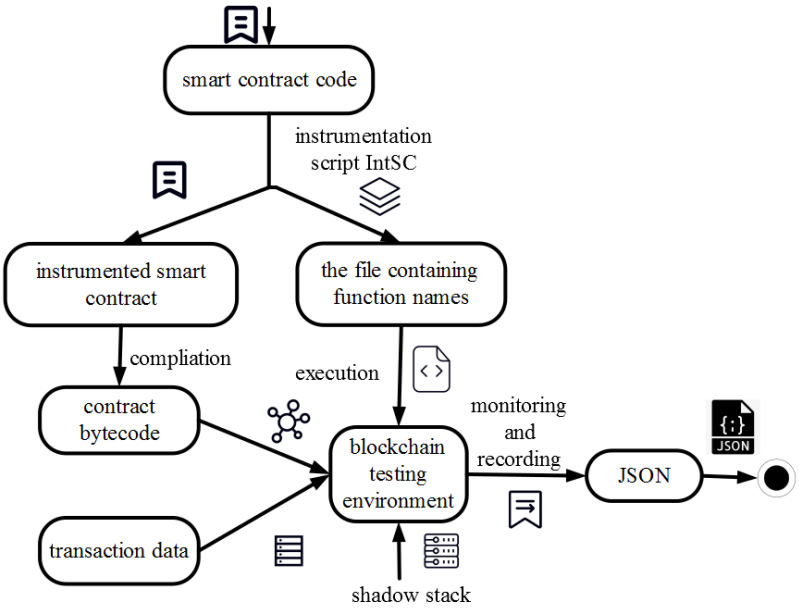

FIGURE 2: Flow chart of monitoring mechanism.

visualization and intelligent analysis.

Algorithm 1 INSTRUMENTATION

```
1:  variables file and funcNames are initialized
2:  for scLine in smart contract code do
3:      regular expression is used in this line to get the func-
        tion name after "function"
4:      funcName = function name
5:      if funcName is not NULL then
6:          funcNames <- funcName
7:          file <- scLine
8:          file <- "bytes32 funcName="+funcName+";"
9:      else
10:         file <- scLine
11:     end if
12:     file_inst <- file
13:     file is set NULL
14: end for
15: file_names <- funcNames
```

1) SMART CONTRACT CODE INSTRUMENTATION METHOD

Based on instrumentation algorithm, an instrumented smart contract named *file_inst* (function names are inserted) and an independent file (*file_names*) containing all the function names are created. It is displayed as **Algorithm 1**. The contract source code is loaded and examined line by line automatically. First, a variable *file* of string type is initialized to store the instrumented contract code, and an array variable *funcNames* is generated to retain all function names of the contract. Then, for each line in the source code, the function name after the keyword "function" is obtained by regular expression and is stored into *funcNames*. Variable *scLine* represents the content of this line, while variable *funcName* is utilized to indicate local function name. Symbol "=" represents assignment, and "<-" means addition. If *funcName* is not empty, it will be pushed into the array *funcNames*. Then *scLine* and *funcName* in form of bytes32 type (the contract function is created in form of bytes32 type, so it is designed the same in the monitoring system) will be added into *file* respectively. If *funcName* is empty, *scLine* will be inserted into *file* directly. Then the content of *file* will be resolved and pushed into the file *file_inst* and set empty. The organization of *file_inst* including the instrumented part is similar to the source code. The function names in *funcNames* will be flushed into *file_names* after the recycling structure. The core procedure is shown in lines 2-11 of **Algorithm 1**. When the traversal process is finished, the code of *file_inst* has included the function names. It could be applied to judge the start and end position of the function at the time of pushing and popping the inserted function name in the stack structure. Meanwhile, these positions are helpful for calculating the instruction number and gas consumption of specific function. One demonstration of code instrumentation is shown as Figure 3. The codes colored with red are the illustration content of instrumentation. After that, the instrumented contract code could be deployed on the testing blockchain environment for monitoring work.

2) SMART CONTRACT CODE DEPLOYMENT

Besides the characteristic of codes, smart contract represents a special transaction after being deployed on blockchain. The procedure is shown as figure 4. If one transaction address *to*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

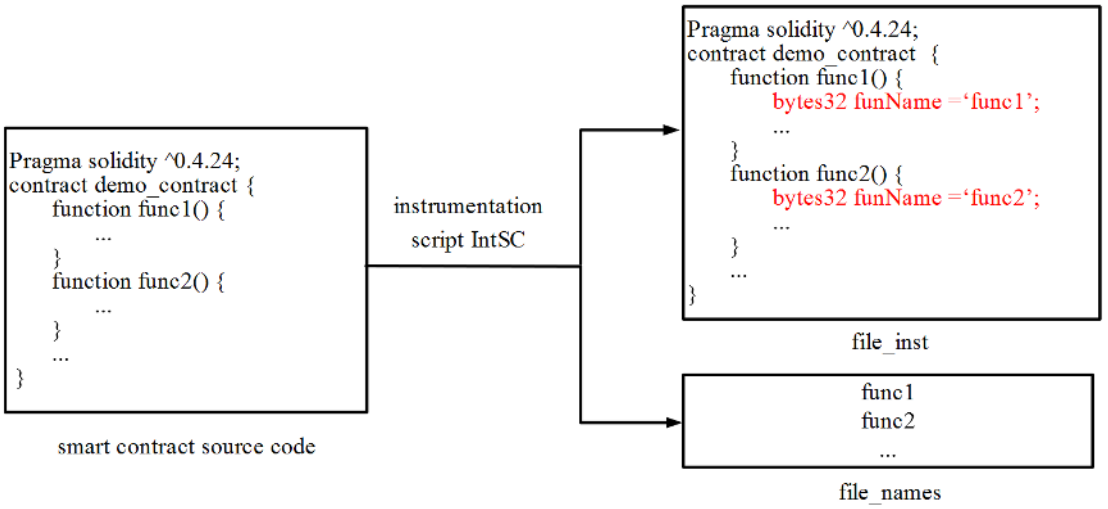

FIGURE 3: Demonstration of code instrumentation.

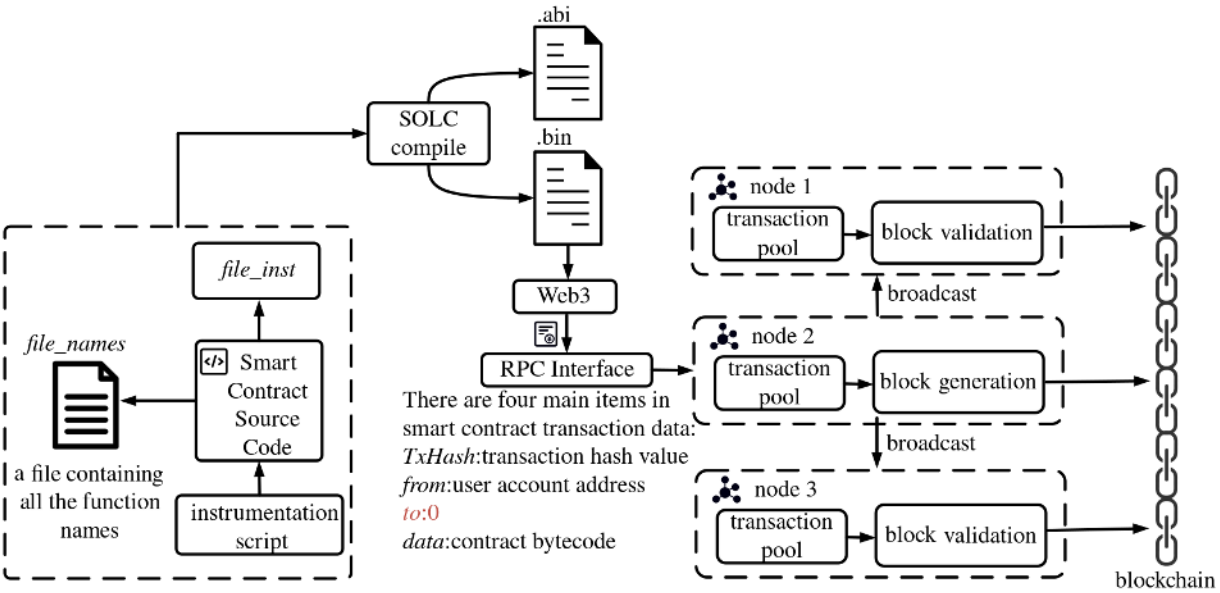

FIGURE 4: Schematic diagram of smart contract deployment.

is empty (i.e. nil), it indicates that a smart contract is created. Then, the instrumented code is compiled by SOLC to generate two files: one file takes bin as its suffix while the other one uses the suffix of abi. The special transaction is converted into a Message object, which includes the transaction's hash, the *from* address (user account address), the *to* address (i.e. nil or 0), and the file suffixed with bin. Furthermore, the EVM bytecode is sent to the Ethereum network through RPC interface. The deployment of instrumented smart contract is described as follows. The smart contract address is generated when the block is packaged, then the contract is verified by all the blockchain nodes through Ethereum network. The contract code related to its address will be stored on the blockchain system and the deployment of instrumented code is finished.

3) BLOCKCHAIN VIRTUAL MACHINE DYNAMIC MONITORING MODULE

EVM is a stack-based virtual machine in nature. Smart contract code could be acquired according to its address, and then is loaded on local EVM for execution. The illustration of smart contract calling and EVM execution is displayed as figure 5. There are four main items in smart contract transaction data: the first one *TxHash* is transaction hash value, the second one *from* address indicates user account address, the third one *to* address means contract address, the last one *data* is the function name and related parameter. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

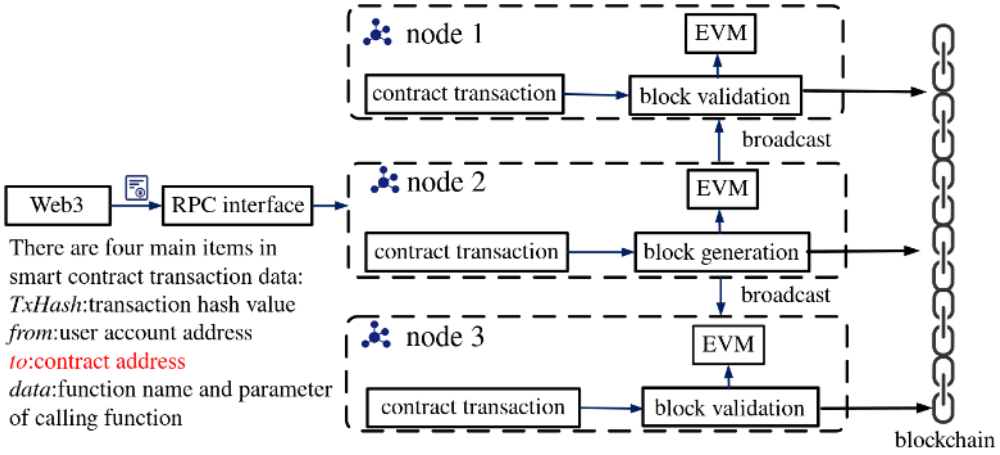

FIGURE 5: Illustration of smart contract calling and EVM execution.

caller takes advantage of the contract address and file suffixed with abi to acquire stored contract code from the blockchain database, then obtains the contract function bytecode through the called function name and related parameter. Essentially, the smart contract code is also a transaction. When this special transaction is packaged and a block is generated, then the called function will be executed on local EVM environment. Meanwhile, the block result will broadcast to other nodes on Ethereum for verification. If successful, the EVM on these nodes will be invoked for code execution and the block will be stored on blockchain.

The module proposed in this paper collects the information of current execution function in the contract by monitoring local EVM. One obstacle is that the entrance and exit of contract bytecode function are difficult to be acquired. Since the function execution on EVM is based on stack structure, we construct a "shadow stack" and related data structures to simulate the function management policy for monitoring. At the same time, the function information could be recorded.

Figure 6 is the graph of monitoring architecture, which is composed of execution area, shadow area, *Decision Engine*, *Recording Engine*,*record_file*,etc. According to this, the monitoring principle is clear. The function bytecode is called through *data* and *to* address. An array *funInfo* is created to store all the function's instructions and their gas values, and a gas mapping table can be designed to query instruction's gas consumption. More important, a "shadow stack" is established for identifying the function execution cycle, and an array *fun_names* is constructed for function name retrieving (*fun_names* contains all the function names of the contract after instrumentation and the data is from *file_names*). The procedure is presented as follows:

(1) Initialization phrase. Two variables (*I* and *pc*) are initialized. *I* is utilized to count the executed instruction and is set to zero, while *pc* points to current instruction and is configured to the address of initial instruction in this smart contract;

(2)During the process of code execution, we can acquire the instruction *op* from variable *pc* with the function *GetOp* (*op=GetOp(pc)*);

(3)Then the instruction enters *Decision Engine*;

(4)If the metric of gas consumption is required, the gas value will be obtained from gas mapping table by *op*;

(5) *Decision Engine* stores the variables of *I*,*op*,*gas* into *funInfo*;

(6) *Decision Engine* judges whether *op* is PUSH32. If it is, the following 32-bit data (the data is generated from intrumentation and the type is bytes32) next to PUSH32 instruction will be sent to *fun_names* for query. When one element in the array is matched, a signal "yes" will be returned. Otherwise, the signal is "no" and the operation will jump to (8). If *op* is not PUSH32, the operation will jump to (8) as well;

(7)The function name, current time and current instruction number *I* (i.e. *funName*,*startTime* and *startInst*) will be pushed into "shadow stack";

(8)*op* is pushed into the stack in the execution area. If *op* is a POP instruction, the operation will jump to (9). Otherwise, it will jump to (2);

(9)*Decision Engine* sends a signal to *Recording Engine*. Meanwhile, the stack top element (POP operation) of execution stack is popped and sent to *Recording Engine*. *Recording Engine* receives it and compares it with the stack top element of "shadow stack"(obtained from Peek method, and the top stack element is still in the stack). If the two elements are not equal, the operation jumps to (2). If they are equal, it means this function is completed. *I* at this point is equal to *endInst*, while the current time is *endTime*. Then the stack top element of "shadow stack" (POP operation) is popped, and *funcName*, *startTime* and *startInst* are acquired. Further, a function name, i.e. parent function name *fatherName*, can be obtained from the stack top element of "shadow stack" with Peek method. Subsequently, the total gas consumption value *sumGas* is acquired from summary of instruction's

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

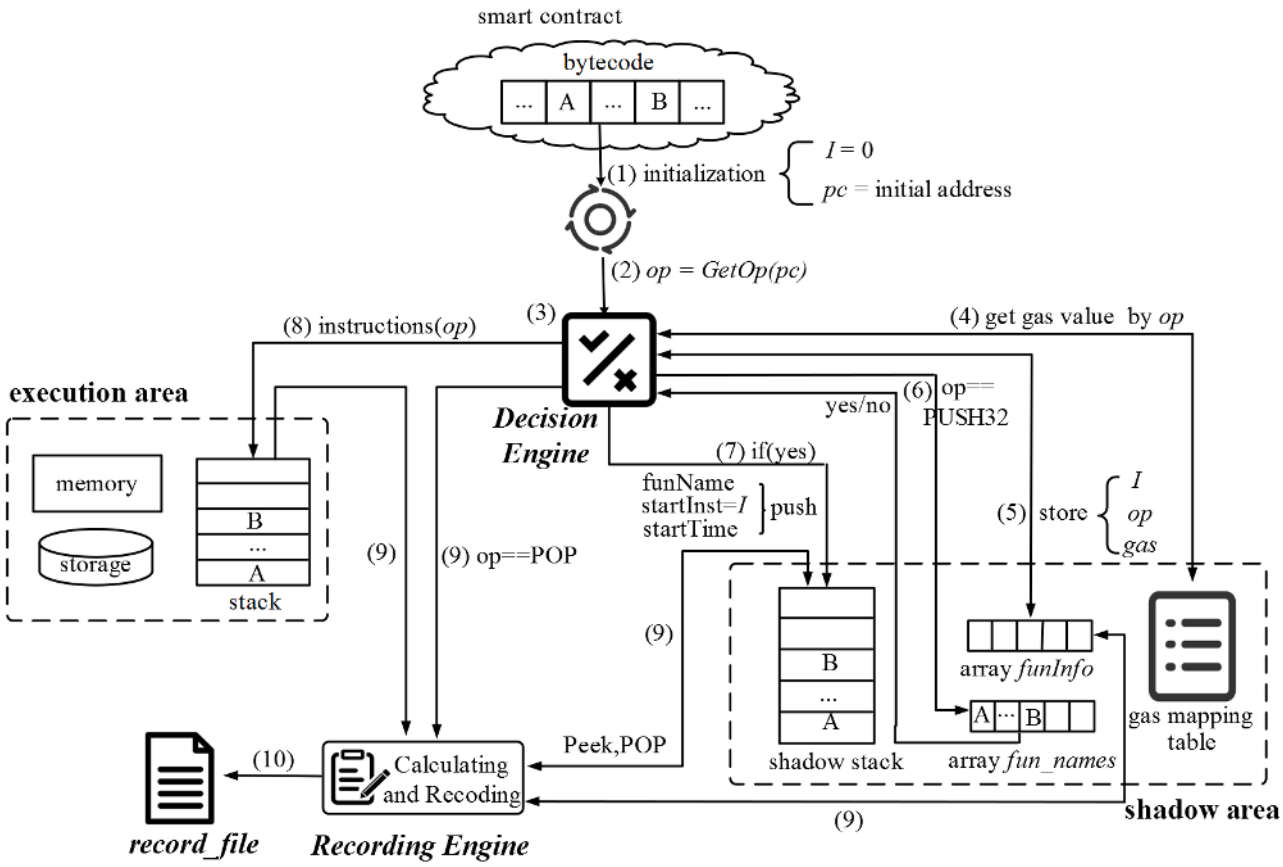

FIGURE 6: Schematic diagram of EVM monitoring architecture.

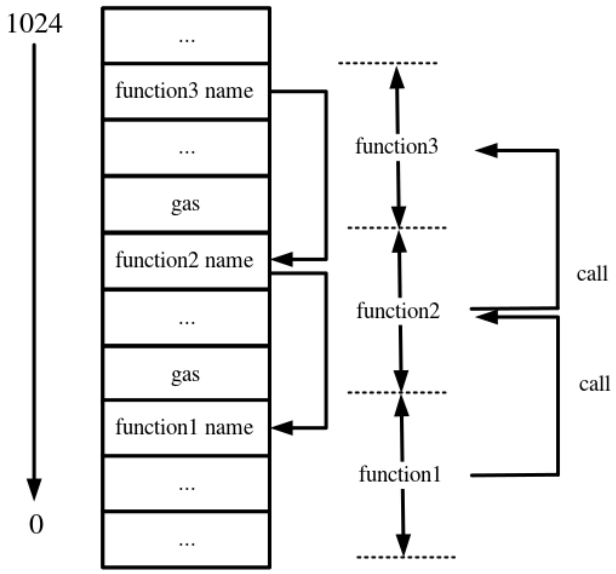

FIGURE 7: Schematic diagram of "shadow stack" operation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

TABLE 1: Description Of Output Content

Parameter Name	Explanation
funcName	function name
fatherName	the function father's name
sumTime	function timing
sumInst	the function execution time
sumGas	the gas cost of the function

gas consumption in the area of *funInfo* array determined by the sign of *startInst* and *endInst*. Similarly, the function execution instruction number *sumInst* is obtained by subtraction of instruction numbers at the two positions (i.e.*sumInst* =*endInst-startInst*) referred above. The method to acquired execution time is the same;

(10) Finally, *funcName*, *fatherName*, *sumGas*, *execution time* and *sumInst* are outputted in form of Json, and it is stored into *record_file*. Then the operation jumps to (2) till the program finishes.

Figure 7 is a schematic diagram of "shadow stack" operation. The calling relationships between function 1, function 2 and function 3 are depicted as follows: when function 1 is executed, function 1 calls function 2, and then function 2 calls function 3. It obeys the regulation of stack.

The system not only monitors the function calling relationship, but also records execution time, instruction number, and the gas cost. Generally, there are five primary elements stored in form of JSON, as shown in Table 1: *funcName* is current function name; *sumTime* is the function execution time; *fatherName* is the upper-level function name, that is name of the function which calls the current one. If the current function is the first one, the value of *fatherName* is nil; *sumInst* is used to calculate all the instruction number of current function; *sumGas* is the gas cost of the function. However, the system can be configured with different metrics. If the gas is not required, the step (4) will removed and relevant variable will be discarded. The flexible mechanism will help the system to be more effective.

B. MONITORING SYSTEM ANALYSIS MODULE

After instrumentation, monitoring and recording, the smart contract's behaviors are required to be analyzed. This part primarily introduces two analysis methods: visual display and intelligent analysis.

1) VISUAL DISPLAY

(1) Call tree graph analysis

This paper monitors the dynamic execution of smart contract and records related data for analysis. The calling relationships between functions can be visually displayed through a call tree graph.

It is supposed that a smart contract is deployed on the blockchain test environment composed of four nodes. Figure 8 is a diagram of call tree instance. Each square represents a function, and the arrows represent the calling relationships between functions. The function monitoring information includes function name, instruction number, gas cost as well as the execution time for all the blockchain nodes. For instance, if function 1 calls function 5 multiple times, it will be marked in the figure. The graph also supports to display multi-level function invoking.

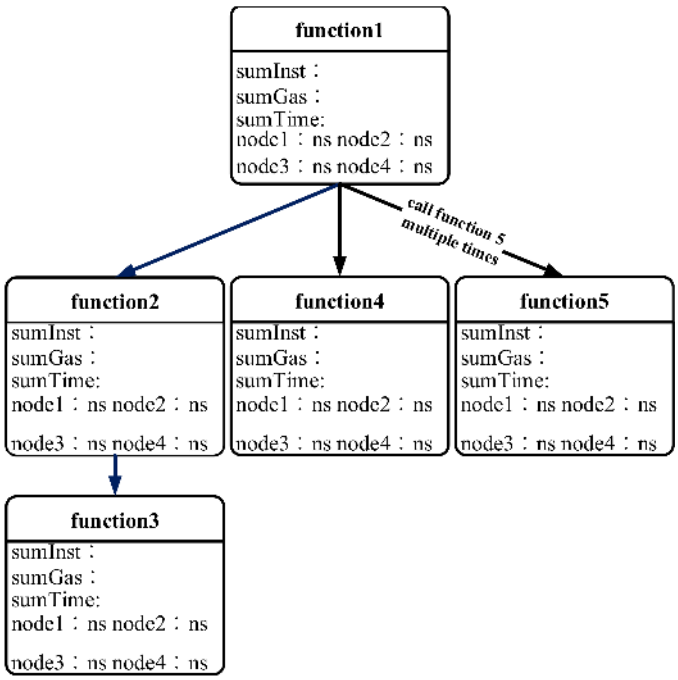

FIGURE 8: The diagram of call tree instance.

(2) Function classification statistics

Smart contract is compiled into bytecode and executed on EVM. At present, the number of instructions are approximate 142 in the instruction set, which includes computation instruction, operation instruction (stack, memory and storage), information acquisition instruction (blocks and smart contract relevant information) and system instruction, etc.

According to instruction classification of Ethereum project yellow paper [10], the instruction could be classified in summary as follows: computation, environmental information description, storage and system operation. And this categories are actually applied for the function in smart contract. The regulation can be configured as the percent of instruction number or execution time in the function for one specific type of instruction. As shown in Table 2, it can be seen in the table that the number of storage instruction is the most, and the instructions for system operation are focus on *call*, *callcode*, and *delegatecall*.

2) INTELLIGENT ANALYSIS

In addition to visualization, another method could be chosen that is intelligent analysis. We use an instance of gas consumption analysis for demonstration.

The gas consumption is an indispensable indicator to ensure the transaction's normal execution on Ethereum. Most instructions in smart contract require a certain gas fee. It is supposed that the number of gas consumption exceeds the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

TABLE 2: Instruction Classification Table

Instruction category	Function description	instruction illustration
computation	instruction that performs arithmetic and logic operation	ADD, MUL, SUB, DIV, LT, EQ, SZERO, AND, OR, NOT etc.
environment information description	instruction which is used to print smart contract's environment, block information and logs	ADDRESS, BALANCE, CALLER, LOCKHASH, TIMESTAMP, LOG, etc.
storage	stack, memory, storage, replication, and swap operation	PUSH1 PUSH32, POP, SSTORE, SLOAD, MLOAD, DUP, SWAP, etc.
system operation	system operation	CALL, CALLCODE, DELEGATECALL, etc.

TABLE 3: Instruction List of Problem Prone

Problem prone	Instruction	Cause of the problem	Contract optimization proposal
reentrancy vulnerability	CALL	the CALL method is used for transfer	check if gas limit is set as the parameter of CALL
gas consumption	SSTORE, SUICIDE, etc.	a lot of gas costs	reduce or avoid the use of this type of instruction
integer overflow	ADD, MUL, ADDMUL, etc.	beyond the maximum integer range	build a rule to determine if there is an overflow
logical error	CALL, CALLCODE, DELEGATECALL	improper use of call instruction	intelligent discovery and reminder

account's gas maximum(i.e. gas limit), then all the operations of current transaction will be stopped and the smart contract will roll back to original state. Moreover, the account may need to pay a certain extra fee to the miner (consensus protocol), which could be avoided.

Before deploying the contract on productive blockchain, each bytecode is expected to be understood and used correctly. There are three methods for transfer: address.transfer(amount), address.send(amount) and address.gas(gas_value).call.value(amount). The gas limit of the first two methods is determined, which can effectively prevent the occurrence of reentrancy vulnerability (the attacker uses fallback function of smart contract to repeat transfer and cause economic loss). However, the third method does not have gas limit. The "call" instruction is prone to generate reentrancy vulnerability. When the monitoring system identifies the "call" instruction, more attention should be paid for reentrancy vulnerability. The following rule can be defined to automatically detect the security of "call" instruction: check whether there is a gas limit value in the first parameter of "call" command. When the gas consumption exceeds the limit, transfer operation will be prevented, then repeated transfer is avoided. If the first parameter is not set, the risk of economic loss (extra gas consumption) is also very high. Therefore, a reasonable gas limit is required for the transfer operation.

The objective of high quality contract code emphasizes on security and effectiveness. The developer should consider it and do the code optimization cautiously. In addition to the "call" instruction related above, we summarize four types of problem prone instructions displayed in Table 3. The first one is the "call" instruction problem about reentrancy vulnerability. The second category is gas consumption prone as well, frequent use of which will cause gas exhaustion warnings. For instance, *SSTORE* instruction and *SUICIDE* instruction, both of which consume lots of gas. It is not recommended to employ them frequently. The third category refers to computation operation instruction, such as *ADD*, *MUL*, *SUB*, etc., which may cause the problem of integer overflow. The final type is also from the calling instruction including *CALL*, *CALLCODE* and *DELEGATECALL*, causing logic error. The warning rules based on the problem prone instructions above are prepared and developed in the analysis module, it will play a significant role for problem identifying and optimization suggestions.

IV. EXPERIMENTAL ANALYSIS

In this section, two experiments were designed to verify the performance overhead and the analysis method for the monitoring and analysis system SCMon. The experimental environment is a test blockchain system.

A. PERFORMANCE ANALYSIS OF MONITORING MODULE

Performance overhead is an important criterion for monitoring system, which is directly related to the disturbance of program behavior. If the overhead is too large, it will affect the monitoring result, normal execution of the program, and even the program's correctness.

This work evaluates the system's overhead by comparing the function execution time between the original and instrumented codes. Smart contract is different from traditional program. Once being deployed, it is not easy to be modified or interrupted. Then the blockchain test environment was utilized in our experiment. In addition, the focus was the function-level disturbance, not the entire system. Therefore, five typical functions with different execution time were chosen and the bytecode execution time before and after instrumentation was measured. The experiment was tested 10 times continuously to calculate the average result for accuracy. It can be seen in figure 9 that the smart contract function runs at millisecond level (the time is converted into the form of nanosecond for clear display in the figure). By contrast, the code execution time after code instrumentation increases approximate 5% compared to the execution time without

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

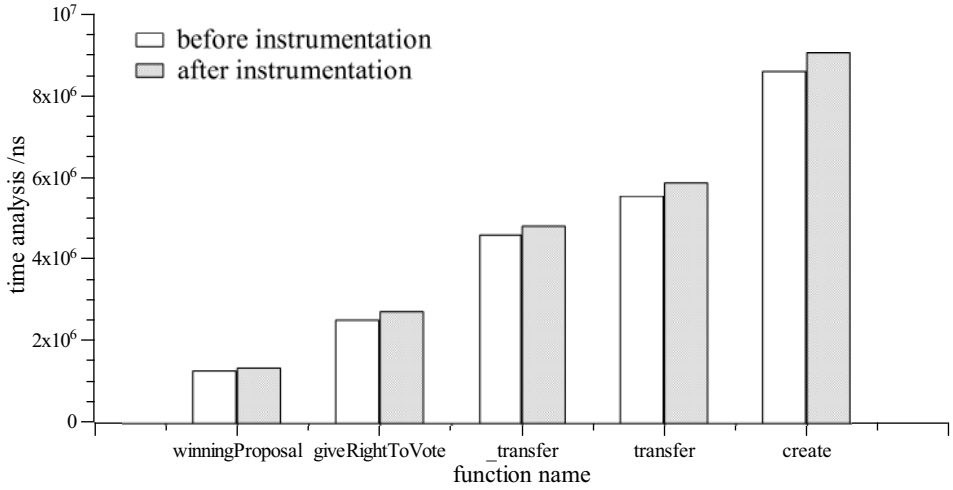

FIGURE 9: Comparison graph of function execution time with code instrumentation.

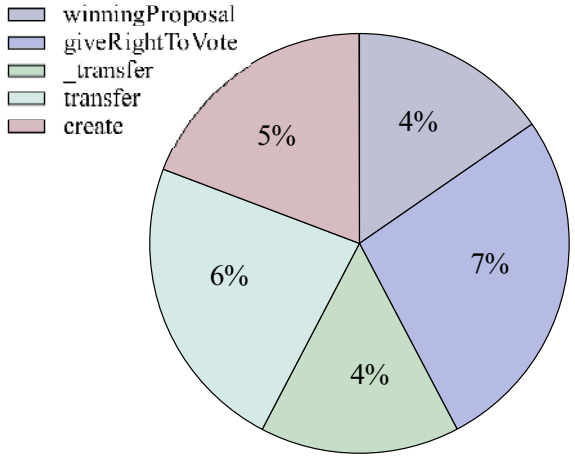

FIGURE 10: Growth rate of function execution time with code instrumentation.

code instrumentation. Furthermore, what can be clearly seen in figure 10 is the steady growth of execution time with code instrumentation. In summary, the instrumentation overhead is small and acceptable, then the monitoring work can be effectively finished.

B. SMART CONTRACT ANALYSIS METHODS

To verify the analysis method of SCMon, a voting case was instrumented by the script IntSC and executed on a blockchain test environment composed of 4 nodes.

This contract case is described as follows: *delegate* function is used to authorize the caller's voting rights to an agent 's address *to*, and some relevant information of *to* is checked, such as whether the address has voting rights, whether it has voted and the voting weight. Then some internal functions of the contract are invoked, including *nulladdress*, *delegation*, *setarry*, *data2*, *data4*, *getdata1*, *getdata4* and *assign* functions, to ensure the normal execution and assign relevant data. First, the agent's address *to* is acquired from the *getdata4* function. We can use *nulladdress* function to determine whether this address is 0, and *delegation* function can offer the result whether the address *to* is the caller's address. The conditions of " *to*! =address (0)" and "*to*! =msg.sender" are required for normal contract execution. Either *nulladdress* function or *delegation* function returns true, the *delegate* function will stop. Second, *data2* function can provide the result whether the caller has the voting right. If the caller has the right to vote, *data4* function is invoked for agent's address assignment. Finally, *assign* function can be utilized to decide whether current delegate has voting right and whether it has already voted. If the voting operation is finished, *getdata1* function can supply the caller's voting weight. Meanwhile, *setarry* function is responsible for increasing the voting number with invoking *add* function.

1) Function relationship call tree

After monitoring, the smart contract execution information

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

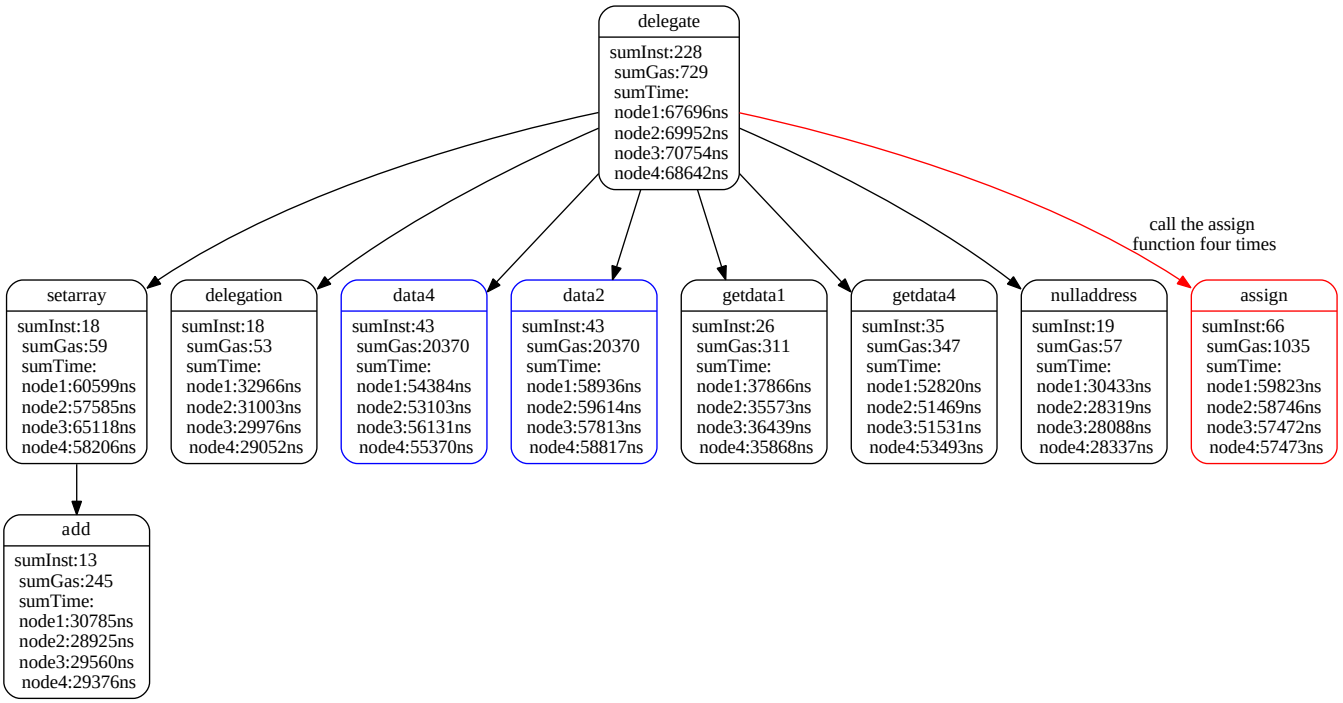

FIGURE 11: The instance graph of monitoring function in vote contract call tree.

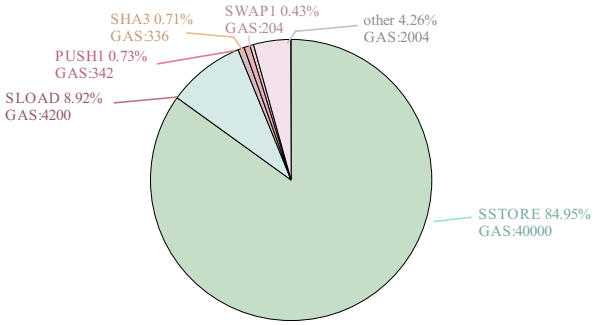

FIGURE 12: Gas consumption distribution diagram.

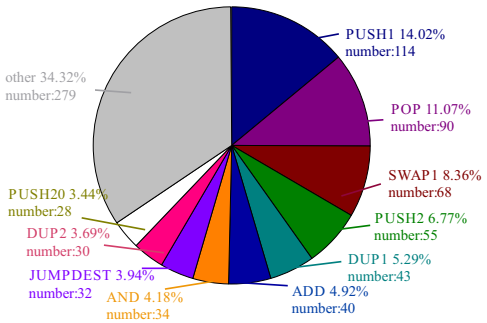

FIGURE 13: Instructions distribution diagram.

will be collected. Then function relationship call tree as displayed in figure 11 is easy to be considered. When the *delegate* function is executed, the calling relationships between these functions, each function's execution time, instruction number, as well as the amount of gas consumed, can be clearly seen from the graph. The function execution time on the 4 blockchain nodes is similar. The red arrow indicates that calling frequency between *delegate* and *assign* function is the highest, and the function with most gas consumptions is marked to help users to observe conveniently.

2) Gas consumption distribution and classification

The gas fee on Ethereum is the basis requirement for smart contract execution. Due to the limited amount, the reasonable use of gas has become an indispensable factor for contract optimization.

The SCMon system dynamically monitors the transaction bytecode (*delegate* function). When a function costs too much gas, a deep cause exploration will start. The instruction which costs most will be identified by intelligent analysis and it is helpful to optimize the targeted code. In our testing case, two functions invoked by *delegate* function cost more than 20,000 gas, which may cause a gas exhaustion warning and stop the transaction. Figure 12 shows instruction-gas consumption distribution for this transaction. From this figure, we can see that the gas consumed by *SSTORE* instruction is 40,000, while this transaction total gas cost is 47086. It is easy to infer that the proportion of *SSTORE* is 84.95%, *SLOAD* is only 8.92%, while others are fewer. From intelligent analysis, the system will give the developer some suggestions for code optimization. For instance, more "memory space" in Solidity is recommended to be utilized instead of "storage space" (*SSTORE* and *SLOAD*), which generally consumes more gas.

Meanwhile, *delegate* function incudes 813 instructions in total and the distribution of instruction number is illustrated in figure 13. If the instruction numbers are sorted in descending order, the first five ones will be listed as follows: *PUSH1*, *POP*, *SWAP1*, *PUSH2*, and *DUP1*, which are all stack operations. Among them, there are 114 *PUSH1* instructions, which account for 14.02% of the total amount. According to the classification criterion, *delegate* function is attributed to storage class function and is mainly used to store data.

To conclude this section, the experiments were carried out to demonstrate that the monitoring and analysis technologies are useful for contract optimization and the prototype system is effective with high efficiency.

V. SUMMARY AND OUTLOOK

This paper proposes a function-level dynamic monitoring and analysis method for smart contract, and implements SCMon prototype system to form a closed loop on smart contract optimization. The characteristics are listed as follows:

1) The monitoring mechanism designs a function name instrumentation algorithm for code instrumentation, and the monitoring module is integrated into the blockchain test environment. A "shadow stack" and related data structures are established in the smart contract virtual machine to simulate the original stack's function management policy, which is used to record function call relationships with low overhead. It could satisfy the monitoring requirements of high efficiency and accuracy.

2) The monitoring method supports multi-dimensional metrics which include not only traditional function execution time, calling relationship, instruction number, but also smart contract related metric:gas consumption. These metrics are valuable for understanding function execution behavior.

3) The combination of visualization and intelligent analysis could help developer understand smart contract profoundly, and identify performance bottleneck, security vulnerability and abnormal behavior quickly as well.

4) Traditional instrumentation based monitoring and analysis methods are introduced into the field of smart contract engineering and improved to adapt the characteristics of blockchain and smart contract. This work enriches and develops the architecture of smart contract engineering, and extends smart contract application area.

This research is still in preliminary and experimental phase. More cases will be tested and more analysis methods will be explored to improve the system. It is also expected to be applied into productive environment in the near future.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," White Paper: https://bitcoin.org/bitcoin.pdf, 2008.

[2] V. Buterin, "A next-generation smart contract and decentralized application platform," White Paper, 2014. [Online]. Available: https://www.weusecoins.com/assets/pdf/library/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platformvitalik-buterin.pdf

[3] N Szabo, "Smart contracts: building blocks for digital markets," EXTROPY: The Journal of Transhumanist Thought, vol. 18, no. 2, 1996.

[4] D. Chris, Introducing Ethereum and solidity, Berkeley: Apress, 2017.

[5] M. di Angelo and G. Salzer, "A Survey of Tools for Analyzing Ethereum Smart Contracts," 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), Newark, CA, USA, pp. 69-78, 2019, doi: 10.1109/DAPPCON.2019.00018.

[6] M. di Angelo and G. Salzer, "A Survey of Tools for Analyzing Ethereum Smart Contracts," 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), Newark, CA, USA, pp. 69-78, 2019, doi: 10.1109/DAPPCON.2019.00018.

[7] I. NikoliĆ, A. Kolluri, I. Sergey, P.Saxena, A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18), 2018, pp. 653-663.

[8] D. Siegel, "Understanding the dao attack," in Retrieved June, 2016, vol. 13, 2018.

[9] P. McCorry, M. Möser and S. T. Ali, "Why preventing a cryptocurrency exchange heist isnąt good enough," in Cambridge International Workshop on Security Protocols, Springer, Cham, pp. 225-233, 2018.

[10] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, 2015.

[11] T. Cook, A. Latham, J. H. Lee, "Dappguard: Active monitoring and defense for solidity smart contracts," in Retrieved July, 2017, vol. 18, 2018.

[12] A. DIKA, "Ethereum smart contracts: Security vulnerabilities and security tools," Master's Thesis. NTNU, 2017.

[13] J. Feist, G. Grieco and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, pp. 8-15, 2019, doi: 10.1109/WETSEB.2019.00008.

[14] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, and Y. Alexandrov. "Smartcheck: Static analysis of ethereum smart contracts," In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 9-16, 2018.

[15] P. Tsankov, A. Dan and D. Drachsler, "Securify: Practical security analysis of smart contracts," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67-82, 2018.

[16] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. "ZEUS: Analyzing Safety of Smart Contracts," Network and Distributed System Security Symposium, 2018.

[17] C. Peng, S. Akca and A. Rajan, "SIF: A Framework for Solidity Contract Instrumentation and Analysis," 2019 26th Asia-Pacific Software Engineering Conference (APSEC), Putrajaya, Malaysia, pp. 466-473, 2019, doi: 10.1109/APSEC48747.2019.00069.

[18] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," arXiv preprint arXiv:1812.05934, 2018.

[19] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, and M. Sagiv, "Online detection of effectively callback free objects with applications to smart contracts," in Proceedings of the ACM on Programming Languages, vol. 2, no POPL, pp. 23-26, 2017.

[20] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan and Z. Chen, "EASYFLOW: Keep Ethereum Away from Overflow," 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 2019, pp. 23-26, doi: 10.1109/ICSE-Companion.2019.00029.

[21] R. Norvill, B. B. F. Pontiveros, R. State and A. Cullen, "Visual emulation for Ethereum's virtual machine," NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, Taipei, pp. 1-4, 2018, doi: 10.1109/NOMS.2018.8406332.

[22] F. Ma, Y. Fu, M. Ren, M. Wang, and X. Shi, "EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine," 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, pp. 554-558, Feb. 2019, doi: 10.1109/SANER.2019.8668038.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3046005, IEEE Access

YI DING received the Ph.D. degree from the School of Computer Science and Engineering, Beihang Univerity, Beijing, China. From 2017, he began to work in the School of Information, Beijing Wuzi University. His research interests include blockchain and smart contract technology, cloud computing, privacy protection, etc.

JIE LI was born in Beijing, China in 1983. She received the M.S. degree from the School of Economics, Anhui University, Anhui, China. She is a Ph.D. Candidate in the School of Computer Science and Engineering, Beihang Univerity, Beijing, China. From 2012, she began to work in the School of Information, Beijing Wuzi University. Her research interests include blockchain, security, etc.

CHENSHUO WANG was born in Tongzhou District, Beijing, China in 1998. He is currently pursuing the B.S. degree in the School of Information, Beijing Wuzi University, Beijing, China. His current research interests include blockchain and smart contract technology.

QIONGHUI ZHONG was born in Hunan, China in 1998. She is a M.S. Candidate in the School of Information, Beijing Wuzi University, Beijing, China. Her research interests include blockchain technology, privacy protection, etc.

HAISHENG LI received his Ph.D. degree from Beihang Univerity, Beijing, China. He is a professor in School of Computer Science and Engineering, Beijing Technology and Business University, China. He is member of China Graphics Society council and senior member of China Computer Federation etc. His current research interests include computer graphics, scientific visualization, blockchain technology, intelligent information processing, etc.

JINJING TAN was born in Anhui, China, in 2000. She is currently pursuing the B.S. degree in the School of Information, Beijing Wuzi University, Beijing, China. Her research interests include blockchain and smart contract technology.