Security Assurance for Smart Contract

Ence Zhou
Information Technology Laboratory
Fujitsu Research & Development Center
Suzhou, China
zhouence@cn.fujitsu.com

Song Hua
Information Technology Laboratory
Fujitsu Research & Development Center
Suzhou, China
huasong@cn.fujitsu.com

Bingfeng Pi
Information Technology Laboratory
Fujitsu Research & Development Center
Suzhou, China
winter.pi@cn.fujitsu.com

Jun Sun
Information Technology Laboratory
Fujitsu Research & Development Center
Beijing, China
sunjun@cn.fujitsu.com

Yashihide Nomura
Information Systems Technologies Laboratory
Fujitsu Laboratories Ltd.
Kawasaki, Japan
y.nomura@jp.fujitsu.com

Kazuhiro Yamashita
Information Systems Technologies Laboratory
Fujitsu Laboratories Ltd.
Kawasaki, Japan
y-kazuhiro@jp.fujitsu.com

Hidetoshi Kurihara
Information Systems Technologies Laboratory
Fujitsu Laboratories Ltd.
Kawasaki, Japan
kurihara.hide@jp.fujitsu.com

***Abstract*—Currently, Bitcoin and Ethereum are the two most popular cryptocurrency systems, especially Ethereum. It permits complex financial transactions or rules through scripts, which is called smart contracts. Since Ethereum smart contracts hold millions of dollars, their execution correctness is crucial against attacks which aim at stealing the assets. In this paper, we proposed a security assurance method for smart contract source code to find potential security risks. It contains two main functions, the first is syntax topological analysis of smart contract invocation relationship, to help developers to understand their code structure clearly; the second is logic risk (which may lead to vulnerabilities) detection and location, and label results on topology diagram. For developers' convenience, we have built a static analysis tool called SASC to generate topology diagram of invocation relationship and to find potential logic risks. We have made an evaluation on 2,952 smart contracts, experiment results proved that our method is intuitive and effective.**

Keywords—ethereum blockchain, smart contract, topological analysis, logic risk location, security assurance

I. Introduction

Blockchains such as Bitcoin [1] and Ethereum [2] are reported frequently in the news recently. Both of them are distributed blockchain ledgers. Compared with Bitcoin, Ethereum permits complex financial transactions or affair rules through scripts which is called smart contract. Thousands of distributed applications in several industry areas have been created on Ethereum blockchain. Especially in 2017, through the statistics on the number of verified smart contracts on the website Etherscan, we found the number is increasing rapidly. Currently, Solidity [3] is the most popular programming language to write smart contracts. However, this language is not perfect enough, it contains many security vulnerabilities [4]. A famous malicious attack was on TheDAO contract [5] which exploited subtle details of the EVM semantics to transfer roughly $50M worth of Ether into the control of an attacker.

After the TheDAO attack, many scholars and organizations started to study the security problems of Ethereum smart contracts. Solgraph [6] generates a graph that visualize function control flow of solidity contract and highlight potential security vulnerabilities. However, it doesn't support cross-file smart contract analysis and vulnerability detection is inadequate. For logic risk analysis, Atzei et al. [4] list the vulnerabilities caused by poor knowledge of the Solidity programming language. Luu et al. [7] use symbolic execution to detect risks in EVM bytecode programs, but it couldn't locate detected risks to specific functions, and the detection method has several limitations for some specific risks (e.g., timestamp risk). Bhargavan et al. [8] outline a framework to analyze and verify both the runtime safety and the functional correctness of Ethereum contracts by translation to F*. However the examples they considered are a little simple and the translation haven't yet been fully implemented.

In order to avoid potential vulnerabilities while developing smart contracts, in this paper we propose a security assurance solution for smart contract. Our solution consists of two main functions. The topology diagram generation of invocation relationship for smart contracts, which supports cross-file smart contract, can help developers to understand their code structure clearly. Besides that, we expand several more kinds of logic risks, and can detect and locate them through symbolic execution and syntax analysis. Both functions have been integrated into SASC (The acronym of this article's title) and this tool will soon be published on Github.

II. Topological Analysis

Solidity [3] is a new programming language designed for writing smart contracts in Ethereum blockchain, it's a programming language which is a little similar to JavaScript. At present, there lacks efficient IDE to support Solidity programming, and tools to do the work of code structure analysis. Therefore, in order to have a clear understanding of the code invocation structure, and to prevent unknown risks, we design a method to generate topology diagram of invocation relationship for smart contract.

978-1-5386-3662-6/18/$31.00 ©2018 IEEE

A. Meta Data Repository for Cross-File Smart Contracts

Solidity-parser [9] is a syntax analysis tool for smart contract, but it can only analyze single file of smart contract each time. Besides, Solidity-parser does not support function invoke relationship analysis. In this paper, we defined four kinds of meta data from source code of a smart contract, they are function, event, modifier and variable. Firstly these four kinds of meta data will be extracted from a smart contract to build a meta data repository by using Solidity-parser, then each line of this smart contract will be analyzed to generate topology diagram combined with the extracted meta data repository.

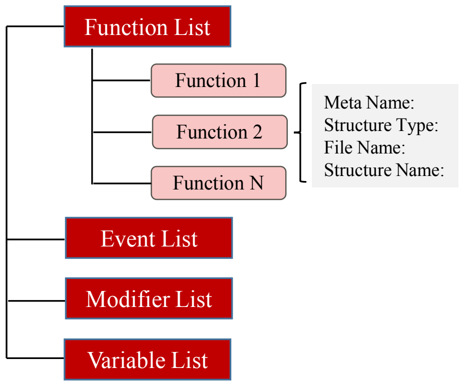


Fig. 1. Structure of a meta data repository.

The structure of meta data repository is shown in Figure 1. Each list is one kind of meta data. And each meta data contains four types of property as following:

1) Meta Name: The name of a meta data, it is a function name in function list or an event name in event list, and so on.

2) Structure Type: Two structure types have been defined: contract and library, representing whether it is extracted from a contract or a library.

3) File Name: For cross-file smart contract, this attribution describes a meta data extracted from which file.

4) Structure Name: It's the contract name or library name from which the meta data has been extracted.

By using these four properties, we can guarantee the uniqueness of a meta data.

B. Invocation Relationship Analysis

Once the meta data repository of a smart contract has been created in the above section, we can analyze each line of its source code to generate the invocation relation. The Solidity-parser is just only a syntax analysis tool, it hasn't point out where the meta data comes from. Therefore combining with some assistant information and with the aid of the built meta data repository, we can obtain the exact information of the meta data, and build the invocation relationship. Based on the features of Solidity language, several invocation rules and search scope determination have been used to generate invocation relationship.

1) Invocation rules: In Solidity, modifier is a specific kind of function, it can be used to change the behaviour of a function easily. Modifiers are inheritable properties of contracts and may be overridden by derived contracts. So, for topological analysis of invocation relationship, we set three rules as following:

a) A function entity may contain several modifiers, events and variables.

b) A modifier entity may contain functions, events and variables.

c) Events and variables can only be invoked by functions or modifiers.

Figure 2 shows a topology diagram of invocation relationship among functions and modifiers.

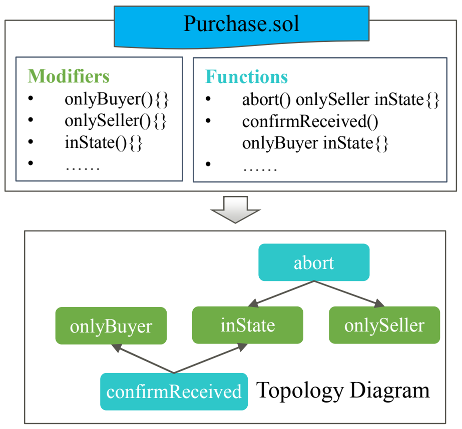


Fig. 2. An example of functions invoke modifiers.

2) Meta data search scope determination: For a cross-file smart contract, or a single contract composed of several sub-contracts, we need to comfirm where is the invocated meta data from. Solidity supports multiple inheritance through property of "is", so we can restrict the search scope to its inherited contracts. Figure 3 shows an example of search scope determination.

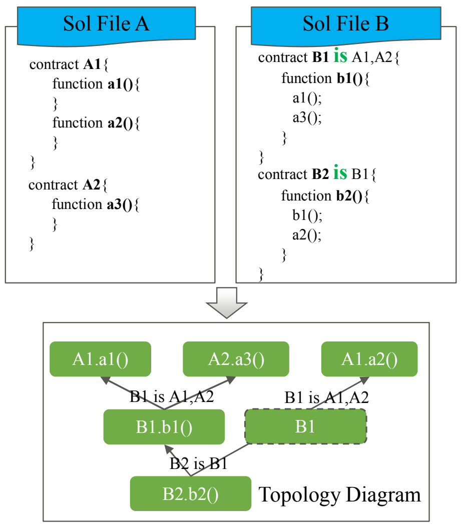


Fig. 3. Search scope determination through property "is".

In Figure 3, the contract B1 in file B inherits from contract A1 and A2, so the search scope is restricted to the composition of contract B1, A1 and A2. Therefore function B1.b1() invokes function a1(), a3(), we just search function a1(), a3() among contract B1, A1 and A2. As contract B2 inherits from B1, so when analyzing the source code in B2, the search scope is the composition of contract B2, B1, A1 and A2.

III. Logic Risk Expansion and Location

Luu et al. [7] defined only four kinds of logic risks. They are Call Stack Risk, Reentrancy Risk, Transaction Order Risk and Timestamp Risk. All of these risks focus on digital asset transfer. Luu et al. also developed a tool called Oyente [10] to detect whether contracts have these four kinds of logic risks, however it can't locate risk to specific function.

In actual situation, due to the imperfection of the EVM, some statements may cause risks or interrupt program running, although there is no digital assets transfer happening. In the following section, we first expand the detection range of timestamp risk and expand another two kinds of risks. Then we propose a method to locate risk to specific function, This is required if developers want to focus and test the risk functions thoroughly.

A. Logic Risk Expansion

Combined with the actual development experience and the attentions provided by the Solidity document [3], we expand the detection range of timestamp dependency, and add another two kinds of logic risks.

a) Expanded Timestamp Risk: For Ethereum blockchain, the miner who creates the new block can choose the timestamp with a certain degree of arbitrariness, i.e. a tolerance of 900 seconds [11]. This may expose the contract to attacks. Luu et al. [7] just focus on the detection of branch statements where digital asset transfer operations exist. However, according to our actual development experience [12] which will be published, block.timestamp as a factor could be used to do some complex risk affairs without asset transfer. As shown in Figure 4, code in line 4 is used to calculate the current year by timestamp. If it is the end of the year, miners can modify the timestamp value within the tolerance of 900 seconds. It will affect the value of the year, which leading to inaccuracy in battery price calculation in line 6.

```
1 function calculateBatteryPrice(){
2   ……
3   uint timestamp = block.timestamp;
4   year = uint16(1970 + timestamp / 31622400);
5   ……
6   price = getPrice(year);
7 }
```

Fig. 4. An example of the expanded timestamp dependency risk. The function implements a battery price calculation depending on the year which has been used.

b) Tx.origin Risk: Solidity official document suggests developers do not use tx.origin for authorization [3]. If a developer use tx.origin for authorization, he may not the only user of his contract. Other people may want to use his contract and want to interact with it via a contract they have been written. Figure 5 shows an example of this kind of risk. The line 8 in Figure 5 (b) creates an attack for UserWallet by invoking transferTo through UserWallet, and the UserWallet using tx.origin for authorization in line 5 of Figure 5 (a). In this situation, the tx.origin will be the original address that kicks off the transaction, which is still the owner address of contract UserWallet. Then the AttackUserWallet instantly drains all the funds of the UserWallet.

```
1 Contract UserWallet{
2   address owner;
3   ……
4   function transferTo(address dest, uint amount){
5     require(tx.origin == owner);
6     dest.transfer(amount);
7   }
8 }
```

(a)

```
1 interface UserWallet{
2   function transferTo(address dest, uint amount);
3 }
4 Contract AttackUserWallet{
5   address owner;
6   ……
7   function(){
8     UserWallet(msg.sender).transferTo(owner,
             msg.sender.balance);
9   }
10 }
```

(b)

Fig. 5. An example of an attack caused by tx.origin. (a) A contract of user wallet. (b) A contract of attack wallet.

c) Zero Division Risk: In development process we found that division by zero in Solidity before Solidity version 0.4 will result in zero, not an exception [13]. For the contract IOU showed in Figure 6, variable "price_in_wei" is an important factor to compute assets in line 9. If this variable was set to zero through the function "updatePrice" by an attacker, the variable "iou_to_purchase" in line 9 will be set to zero in the future invocations and don't throw any errors, this may cause program risks. Although at Solidity version 0.4.15, division by zero throws a runtime error [14], we still need pay more attention to contracts which written by previous Solidity version.

```
1  contract IOU {
2    ……
3    function updatePrice(uint256 _price) {
4      price_in_wei = _price;
5    }
6    ……
7    function purchase() payable {
8      ……
9      uint256 iou_to_purchase = (msg.value *
             10**8) / price_in_wei;
10     ……
11     eth_sent[msg.sender] += msg.value;
12     total_iou_purchased += iou_to_purchase;
13   }
14   ……
15 }
```

Fig. 6. An example of the zero division risk.

B. Logic Risk Location

On the basis of Luu's research result, we first add detection rules of the expanded logic risks, which mentioned in the above section, into the symbolic execution module. And then locate detected logic risk to specific function of a contract's source code. Figure 7 shows the whole structure of our method.

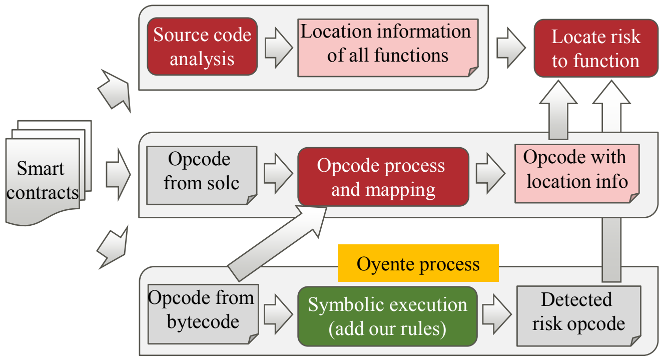
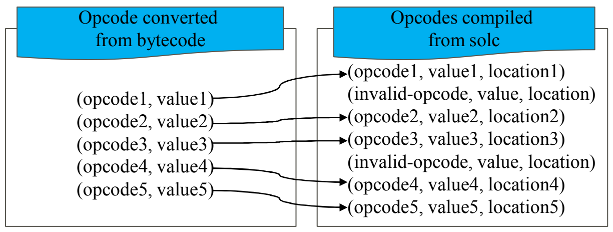


Fig. 7. Logic risk detection and location structure.

The process of symbolic execution using the opcodes which converted from bytecode of smart contracts that don't contain location info. So it can't directly map detected risk opcode to source code. The opcodes that compiled from solc [15] contain location info. However, since they contain some invalid opcodes, they are not exactly the same with the opcodes which converted from bytecode, and they can't be executed directly for symbolic execution. As a result we designed a mapping mechanism of setting a series of rules to recognize invalid opcodes which compiled from solc, and then add location info for opcodes which convert from bytecode. Figure 8 shows the detail.



Opcode converted from bytecode
(opcode1, value1)
(opcode2, value2)
(opcode3, value3)
(opcode4, value4)
(opcode5, value5)
Opcodes compiled from solc
(opcode1, value1, location1)
(invalid-opcode, value, location)
(opcode2, value2, location2)
(opcode3, value3, location3)
(invalid-opcode, value, location)
(opcode4, value4, location4)
(opcode5, value5, location5)

Fig. 8. Opcode mapping mechanism.

At last, in order to locate risk to specific function, we first obtain all the location info of all functions in the topological analysis stage, and then combine with the location info of risk opcodes, we can locate a detected risk to a specific function.

IV. EVALUATION

We have developed a tool called SASC for static analysis of Ethereum smart contracts, it integrated all the above mentioned features, and provides static report in HTML format. SASC will soon be published in Github.

For logic risk detection, We collected 4,744 smart contracts with source code from website etherscan.io [16] which publishes smart contracts with verified source code continuously as of September 30, 2017. These contracts currently hold a total number of 10,973,029 transactions at the time of writing, on an average, a contract generated 2,313 transactions.

Sorting contracts by the number of their transactions in descending order, we tested top 2,952 contracts on a single computer with 8GB memory which runs 64-bit Ubuntu 16.04. We set a timeout for the symbolic execution of 10 minutes per sub-contract, and the timeout for Z3 constraint solver [17] is set to 1 second. Our tool totally takes 502 hours to analyze all these contracts, on an average, it takes 612 seconds (about 10 minutes) to analyze a contract, approximately 82 paths, 500 blocks, 5,236 opcodes are executed for each contract.

A. Logic Risk Statistics

In section III we totally defined six kinds of logic risk, they are Call Stack Risk, Transaction Order Risk (TO), Reentrancy Risk, Timestamp Risk, Tx.origin Risk and Zero Division Risk. Figure 9 shows the experiment result.

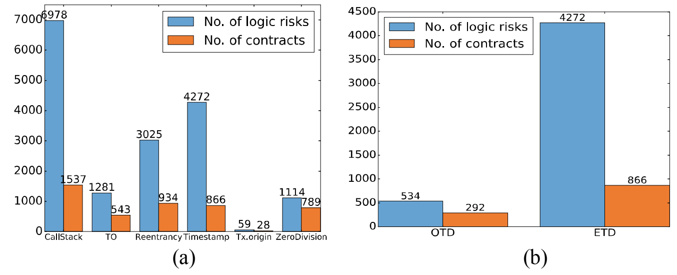


Fig. 9. (a) Number of logic risks and risk contracts detected under each logic risk. (b) Compare with Oyente (OTD: Original Timestamp Dependency, ETD: Expanded Timestamp Dependency).

Figure 9 (a) shows that the most frequent risk is call stack, it is detected out 6,978 times in 1,537 contracts, approximately 52.07% of the contracts in our benchmark. That's because the call operation often occurs at invocation among functions or digital currency transfer between different accounts. The least risk is Tx.origin, only occurs 59 times in 28 contracts, 0.95% of the contracts in our benchmark. One important factor is that the Solidity official document suggests developers do not use this statement.

In addition, we compared the detection result of timestamp with Oyente. As Figure 9 (b) shows obviously, after expanded the detection range of timestamp risk, we detected out more timestamp risks, about 4,272 times in 866 contracts. Meanwhile, Oyente detected out 534 times in 292 contracts. That's because we detect not only timestamp risk in branch statement where digital assets transfer exist, but also statement which contains timestamp factor.

B. Execution Time Evaluation

In this experiment, running time of different contract's symbolic execution may change violently. Some contracts with large amount of code have short running time. On the contrary, some contracts with small amount of code have long running time. For symbolic execution, some opcodes form a block, and some blocks form an execution path. So we consider the execution time is related to these three factors. This experiment totally executed 243,188 paths, 1,477,423 blocks and

15,458,157 opcodes, executed 82 paths, 500 blocks and 5,236 opcodes per contract on an average.

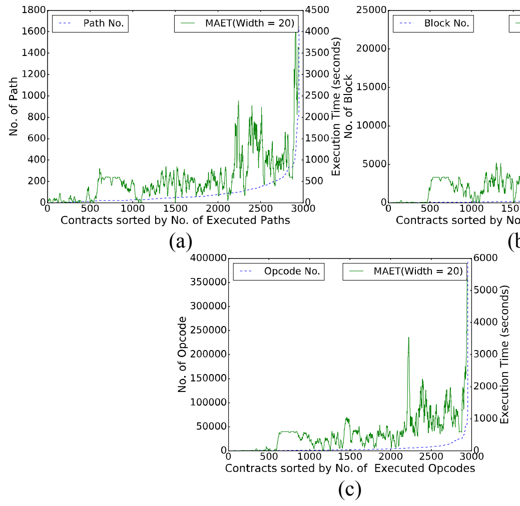


Fig. 10. Evaluation of contracts' execution time. (a) Comparison chart of execution time and the number of executed paths. (b) Comparison chart of execution time and the number of executed blocks. (c) Comparison chart of execution time and the number of executed opcodes.

In order to verify our thoughts, we draw the relationships of different factors and execution time on charts, like Figure 10 shows. For the execution time of adjacent contracts may change violently, to makes the line smoothly, we use Moving Average [18] to express each contract's execution time, which referred to as Moving Average Execution Time (MAET). We observed that the execution time is almost linearly depends on the number of executed paths, blocks or opcodes.

V. Conclusion and Future Work

Blockchain technologies have been developed rapidly in recent years, and the development of smart contracts further promote the diversity of blockchain applications. Since smart contracts can't be withdrawn once they have been deployed, we should make full quality assurance before they are deployed.

This paper proposed a static analysis method for Ethereum smart contracts. It contains two main functions: topological analysis of function invocation relationship, logic risk detection and location. Through topological analysis, developers may have a clear understanding with the operations of their source code, and further avoid potential risks. For logic risk detection and location, we first optimized existed detection range of logic risks, and then expanded two kinds of logic risks to guarantee the security of smart contracts. Combined with syntax analysis, we can locate logic risks to specific functions. Based on these features, we developed the tool which called SASC. It can provide high quality assurance for Ethereum smart contracts.

Although SASC has made topological analysis and logic risk detection and location of smart contracts, but risks are not real errors. As a result, further techniques are needed to confirm the detected risks are errors or not, such as dynamic execution. Dickerson et al. [19] have done some related work about concurrency execution. Next, we will consider how to generate test cases automatically based on the analyzed topology diagram and the detected risk functions, and then consider how to execute generated test cases dynamically under limit situations, like limited gas, limited execution environment and so on. Furthermore, we also will do some work about log analysis to confirm some kinds of risk. Our goal is to ensure the security of smart contracts at both levels of static analysis and dynamic execution.

Acknowledgment

We thank our colleagues Shenbin Zhang, Haoli Sun for giving modification suggestions in an early draft, and also thank Yao Chen and Yi Qian for designing the UI of SASC and doing the statistics of the evaluation in section four.

References

[1] Satoshi N. "Bitcoin: A peer-to-peer electronic cash system," May 2009.

[2] Ethereum. https://github.com/ethereum/.

[3] Solidity. A new programming language for writing smart contracts on Ethereum blockchain. https://solidity.readthedocs.io/en/develop/.

[4] Atzei, N., Bartoletti, M., Cimoli, T. "A survey of attacks on Ethereum smart contracts," Principles of Security and Trust (POST). LNCS, 2017, vol. 10204, pp. 164-186.

[5] TheDAO smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[6] Solgraph. Generates a DOT graph that visualizes function control flow of a Solidity contract. https://github.com/raineorshine/solgraph.

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

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

[9] Solidity-parser. A source code analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/solidity-parser.

[10] Oyente. An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente.

[11] Block validation algorithm - Ethereum wiki, https://github.com/ethereum/wiki/blob/master/Block-Protocol-2.0.md#block-validation-algorithm.

[12] Song H, Ence Z, Bingfeng P, et al. "Apply blockchain technology to electric vehicle battery refueling," Proceedings of the HICSS-51 Track on Organizational systems and technology, Jan 2018.

[13] Solidity integer division problem. https://ethereum.stackexchange.com/questions/3010/how-does-ethereum-cope-with-division-of-prime-numbers.

[14] Solidity document about integers. http://solidity.readthedocs.io/en/v0.4.15/types.html#integers.

[15] Solc, the Solidity compiler. https://github.com/ethereum/solidity.

[16] Etherscan. A website provides contracts with verified source codes. https://etherscan.io/contractsVerified.

[17] Microsoft Corporation. The Z3 theorem prover. https://github.com/Z3Prover/z3.

[18] Moving Average. https://en.wikipedia.org/wiki/Moving_average.

[19] Dickerson T, Gazzillo P, Herlihy M, et al. "Adding Concurrency to Smart Contracts," arXiv preprint arXiv:1702.04467, 2017.