2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

A Modified Smart Contract Execution Enviroment for Safe Function Calls

Sooyeon Lee
Dept. of Computer Science and Engineering
Chungnam National University
Daejeon, Republic of Korea
djm02309@o.cnu.ac.kr

Eun-Sun Cho
Dept. of Computer Science and Engineering
Chungnam National University
Daejeon, Republic of Korea
eschough @cnu.ac.kr

***Abstract*— When a Solidity smart contract has a problem in calling a function of another contract, the "fallback function" of the contract is supposed to be executed automatically. However, in many cases, a fallback function is arbitrarily created and called, with their behaviors unknown to developers, so that its execution is vulnerable to exploits by attackers. To reduce these risks, this paper proposes a method that provides developers with new keywords by modifying existing Solidity compiler and Ethereum Virtual Machine (EVM). Developers mark their intention using the newly introduced keywords, and the modified existing Solidity compiler and EVM uses flags and conditional statements to prevent calls of fallback functions to reduce the risk of calls to fallback functions.**

Keywords—Solidity, compiler, EVM (Ethereum VM), fallback functions, smart contract

I. Introduction

Solidity [1] is the most widely used high-level language for creating smart contracts for Ethereum [2], one of the cryptographic currencies. Ethereum users can use Solidity to create smart contracts and use them for transaction.

However, Solidity is known to have many vulnerabilities, followed by financial damages from transactions through smart contracts. In particular, many vulnerabilities have been identified, especially when invoking functions of other smart contracts. In a distributed smart contract environment, it is difficult for the caller to understand and control the call action, resulting in damage. For example, when a function of other smart contract is called through the address, an unintended function may be executed due to an incorrect address reference. In this case, if the function does not exist at that address, the fallback function, a special function without a name, is implicitly called instead. Since the caller is not aware of the behavior of the fallback function, attackers may take advantage of this to allow the malicious behaviors to be inserted into the fallback function and performed [3].

Several suggestions have been made to reduce these Solidity risks, such as tools to analyze smart contract codes in advance [4][5], completely new languages to support strong types [6], and a method to improve privacy by using cryptographic techniques [7]. However, these methods have difficulties in fully reflecting the nature of the distributed smart contracts, and there are burdens such as introducing a new language. In this paper, we introduce solc+ [8] and EVM+ [9], which modified the existing Solidity compiler (solc) [10] and Ethereum Virtual Machine (EVM) [11]. Solidity language developers can control invocations of the fallback function using additional keywords, analysis and monitoring provided by solc+/EVM+.

The sequence of this paper is as follows. Chapter 2 discusses the background of the research. Chapter 3 describes how the new keywords appear on the solc+/ EVM+. Chapter 4 introduces how these techniques were implemented in the solc+/EVM+, and Chapter 5 gives a conclusion.

II. Background

A. Solidity Environment

Geth (go-ethereum)[12] is an Ethereum byte code execution environment, that provides the command line interface for running a full Ethereum node implemented in the Go language. Ethereum traders can mine ethers (the cryptographic money in Ethereum) using Geth directly or conduct transactions through smart contracts running on EVM. The Ethereum protocol is also implemented in other languages such as C++ and Java, but currently Geth is the dominant one.

Source files containing smart contract written in Solidity will run through the EVM of Geth after they are compiled through solc, the Solidity Compiler. Solc itself is written in C++.

B. Fallback function vulnerability

A fallback function is the unique function of a contract with no parameter, no name, and no return type, and up to one for each contract. There are two main uses of this function.

First, the fallback function is called in an exception situation in a function invocation, when it matches no function in the contract. More specifically, an exception occurs when there is no signature (the 4-byte value of the SHA3 hash of the function name) matching the invoked function, or when type or parameters are not correct.

```
contract A {
      function() { x=1;}
      uint x;
}
```

Fig. 1. Example of the codes that might be call Fallback function

For instance, for contract address `contract_A` of contract A type defined in Fig 1., an exception will occur in the following function invocation even when `contract_A` is the correct address of type `A`. In this case, the fallback function is automatically called because the `f1()` does not exist in contract `A`, so the value of `x` changes to 1.

```
contract_A.call(bytes4(sha3("f1(uint256)")),
                _value);                                   (1)
```

A fallback function will also be called even if the compiler confirms that the contract has the invocated function. Fig. 2 shows the case where the compiler has identified the presence of a function `ping` in `Amy` in advance. However, the compiler cannot confirm whether `c` is actually the address of `Amy` and if the interface of `Amy` defined in `Bill` matches the

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00135

IEEE computer society

interface of Amy at the time it is actually performed. As a result, when "c.ping (42);" is executed, there is still the possibility of an exception situation due to the absence of a function whose signatures do not match. The fallback function is then executed automatically.

```
contract Amy {
     function ping(uint) returns (uint)
}
contract Bill {
     function pong(Amy c) {c.ping(42);}
}
```

Fig. 2. Example of smart contract [3]

Second, the fallback function is also executed if the contract wishes to receive ethers from another contract. In smart contracts, ethers move simply through transfer or by sending to the account to be received, or by creating and invoking special functions specified with the keyword payable. The amount to be moved is specified by "value", the parameter passed on calls in the former case, or explicitly received through a function parameter in the latter case.

Note that in the former case, the fallback function of the incoming account is executed. In this case, an exception will occur and the ethers will be returned to the callers, when the contract does not implement a fallback function or the fallback function is not set to payable. The problem arises in that it is difficult for the caller to clearly understand whether the fallback function was called and the behavior of the fallback function. For example, if a malicious user creates contract such a Fig.2 and intentionally puts a failure-causing code inside the fallback function, the normal progress of the contract can be stopped. And if the contract is executed after the malicious code is inserted, the malicious code may be executed instead of the function of the original fallback function.

C. Exisiting Research

In a widely-used, interactive development environment for Solidity developers, such as Remix [13], warnings are generated about the vulnerable patterns of contract programs. Oyente [4], DappGuard [5] and ZEUS [14] check the safety of smart contracts with static or dynamic analyses. These tools will analyze whether vulnerabilities exist for smart contracts and then show the results to users. Oyente from the University of Singapore is an open source security inspection tool that performs inspections based on symbolic evaluation [15]. It collects constraints on input values and demonstrates with the Z3 Bit-Vector Solver [16] that the program can reach erroneous states with control flow graphs (CFG) with the collected constraints. DappGuard, another tool for safety checks on smart contract codes, conducts static analysis internally using Oyente, and checks more types of vulnerabilities in addition. ZEUS changes its smart contract code to LLVM bit code [17] and analyzes the vulnerabilities of the contract based on user-specified policies. However, smart contract codes cannot be analyzed accurately as the characteristics of smart contracts are omitted in the process of lifting smart contract codes to bit code. On the other hands, decompilers will also help users who want to use smart contract in EVM byte code, by presenting contracts in the high level Solidity language to confirm their content [18].

However, all these approaches are performed after a contract is made, which means that they do not provides strong facilities to make the smart contract developers, one of the most important stakeholders for the safety of smart contracts, involved in the process. Although using a new safer language might be an inherent solution, such a significant change does not seem to be acceptable to the existing Ethereum ecosystem. Using the require[19] statement in Solidity also greatly contributes to enhancing the safety of the code, but it is not possible to indicate whether the fallback function is allowed to be called due to the constraints of the expressive power of the phrase.

III. Overview of the ProposeD method

In our previous study [20], a keyword was provided to developers to control the fallback function, along with a method to rewrite the code using a pre-processing machine. When rewriting the code in this study, a library was created to reduce gas usage to control invoke of fallback. However, we cannot get rid of gas consumption completely with the library approach.

In this paper, we propose modified solc (solc+) and EVM (EVM+) to support the ability to control the fallback function by providing keywords to developers. solc+ provides developers with two keywords: NONFALLBACKON and NONFALLBACKOFF. Using these two keywords, developers can specify a range to prevent the invocation of the fallback function. At the beginning of the point where you do not want the fallback function to be called, write down the keyword NONFALLBACKON, and write down the keyword NONFALLBACKOFF from the part where you want to allow the fallback function to be called. Fig.3 is an example of the conceptual code with keywords. When code is written as in Fig. 3., solc+ recognizes NONFALLBACKON and NONFALLBACKOFF.

```
contract Charlie {
  function ping(uint) returns (uint)
  function () payable { }
}
contract Dave {
  function pong(Charlie c) {
    NONFALLBACKON;
    c.ping(42);
    NONFALLBACKOFF;
  }
}
```

Fig. 3. Example of conceptual codes with the keyword NONFALLBACKON and NONFALLBACKOFF

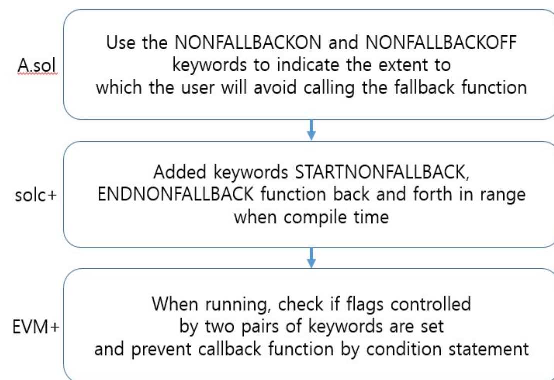

Fig. 4. Flow for suggested method

Solc+ also analyzes the fallback function body of the callee and insert the new pair of byte codes for STARTNONFALLBACK and ENDNONFALLBACK in appropriate positions of the generated byte code. And then when a smart contract with keywords is executed in EVM+, the fallback function located in the keyword's area is not executed by the conditional statement. This process shows as in Fig. 4.

IV. Implementation

This chapter introduces the actual implementation of the proposed method. We modified the solc and EVM of Geth.

A. *solc+*

We modified the solidity compiler (solc) so that developers could use keywords NONFALLBACKON and NONFALLBACKOFF. First, we modified solc's parser to recognize the two new keywords. Next, with a simple program analysis, solc+ determines if it is currently a fallback function body. If this is true, that is, it is at the beginning and the ending of the fallback function body, solc+ internally add STARTNONFALLBACK and ENDNONFALLBACK encoded in new EVM byte codes. The four new keywords were configured with unique types of expression nodes in the parser and inserted to the appropriate locations for the abstract syntax tree (AST) [21]. And then, added visit methods that related to new keyword nodes in every visitor existing in solc+. In particular, solc+ manipulates new keyword nodes properly encoded with the suitable code when generating bytecode with Visitors [22].

B. *EVM+*

The sol file compiled with solc+ has to be executed, so we modified the EVM of Geth (EVM+).

```
NONFALLBACKON = 0x25
NONFALLBACKOFF=0x26
STARTFALLBACK=0x27
ENDFALLBACK=0x28
```

Fig. 5. Assign an opcode to added keywords

First, opcode was added for new keywords (NONFALLBACKON, NONFALLBACKOFF, STARTNONFALLBACK, ENDNONFALLBACK) as shown in Fig.5. 0x25, 0x26, 0x27, and 0x28 values were assigned to each keyword because this range of EVM opcode set is not pre-allocated.

```
type ForFallback struct{
    IsNonFallBackEnforced    bool;
    StartingNonFallback bool;
}
```

Fig. 6. Declares flags to be controlled by keywords

```
func opNonFallbackON(pc *uint64, interpreter *EVMInterpreter, contract
 *Contract, memory *Memory, stack *Stack)([]byte, error){
    fallbackFlag.IsNonFallBackEnforced = true;
    fmt.Print("Set Non fallback mode")
    return nil, nil;
}
```

Fig. 7. Instruction for added opcode (ex.0x25)

Then, as shown in Fig. 6, we created a structure with flags to be controlled by keywords and added an instruction for them, as shown in Fig. 7. Among the instructions, the keywords NONFALLBACKON and NONFALLBACKOFF control the IsNonFallBackEnforced flag, and the keywords STARTNONFALLBACK and ENDNONFALLBACK control the StartingNonFallback flag. New opcodes and their instructions were added to the jumptable in Geth+ so that computations could be executed in the interpreter.

```
if fallbackFlag.IsNonFallBackEnforced
    && fallbackFlag.StartingNonFallback{
    return nil, fmt.Errorf("NonFallback option is on", nil)
}
```

Fig. 8. Added conditional statement in interpreter.go

Finally, as shown in Fig. 8, a conditional statement was added to the filter to prevent the next operation if the flag that related to the non-fallback is set.

```
NONFALLBACKON   pc=00000000 gas=10000000000 cost=0
 UsersetFallback=false StartingNonFallback = false

STOP            pc=00000001 gas=10000000000 cost=0
 UsersetFallback=true StartingNonFallback = false
```

Fig. 9. Result of executing opcode 0x25 on EVM+

After building EVM+, execute the NONFALLBACKON keyword assigned as opcode 0x25, as it appears as Fig .9. As a result of execution the keyword NONFALLBACKON shows that the value of the UsersetFallback flag has changed, which indicates that the user (caller) requests a safer execution without fallback function invocation. It also shows the pc and gas quantity for each opcode, and we can ensure that no gas is consumed to set and manipulate the new keywords and flags. It also works in the same way for Opcode 0x26, 0x27, and 0x28.

V. Conclusions

The fallback function is often executed when an exception occurs when invoking a function of another smart contract from Solidity, or when there is a financial movement between contracts. The content of this fallback function is a function usually written with insufficient care, so it is easy to leverage for attacks. In this paper, we have proposed a method of controlling the invocation of the fallback function with new keywords by modifying the existing solc and EVM of Geth. This allows the developer to decide whether to execute the fallback function using two keywords: NONFALLBACKON and NONFALLBACKOFF. Using the keywords and help of the new compiler and EVM, developers are able to write code that reduces the risk of fallback functions.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIT) (2018-0-00251, Privacy-Preserving and Vulnerability Analysis for Smart Contract) .

References

[1] Solidity, https://solidity.readthedocs.io/en/ develop/

[2] Ethereum. https://www.ethereum.org/

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, "A survey of attacks on Ethereum smart contracts", Proceedings of the 6th International Conference on Principles of Security and Trust, Page164-186, April 22-29, 2017

[4] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena and Aquinas Hobor, "Making Smart Contracts Smarter", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Pages 254-269, October 24 - 28, 2016

[5] Thomas Cook, Alex Latham and Jae Hyung Lee, "DappGuard : Active Monitoring and Defense for Solidity Smart Contracts", mit, 2017

[6] Jack Pettersson and Robert Edström, "Safer smart contracts through type-driven development Using dependent and polymorphic types for safer development of smart contracts", Master's thesis in Computer Science Department of Computer Science and Engineering Computing

Science Division Chalmers University of Technology and University of Gothenburg Gothenburg, Sweden, 2016

[7] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In Proceedings of the 2016 IEEE Symposium on Security and Privacy, SP '16. IEEE Computer Society, 2016

[8] solc+ , https://github.com/PLASLaboratory/solc_plus

[9] EVM+, https://github.com/PLASLaboratory/EVM_plus

[10] solc, https://github.com/ethereum/solidity

[11] EVM, https://en.wikipedia.org/ wiki/Ethereum#Virtual_Machine

[12] Geth, https://github.com/ethereum/go-ethereum/wiki/Geth

[13] Remix, https://remix.ethereum.org/

[14] Sukrit Kalra, Seep Goel, Mohan Dhawan and Subodh Sharma, “ZEUS: Analyzing Safety of Smart Contracts”, Network and Distributed Systems Security (NDSS) Symposium 2018, IBM Research and IIT Delhi, 2018

[15] Symbolic execution, https://en.wikipedia.org/wiki/Symbolic_execution

[16] The Z3 theorem prover, https://github.com/Z3Prover/z3.

[17] LLVM, https://llvm.org/

[18] Solidity Decompiler, https://ethervm.io/decompile

[19] Solidity require, https://solidity.readthedocs.io/en/v0.4.24/control-structures.html

[20] Sooyeon Lee, Hyungkun Jung and Eun-Sun Cho, “Smart Contract Code Rewritter for Improving Safety of Function Calls”, Journal of the Korea Institute of Information Security and Cryptology, vol. 29(1), pp.67-75. 2019

[21] Alfred V. Aho, Monica S. Lam, Jeffrey D. Ullman and Ravi Sethi, Compilers:Principles,Techniques,and Tools, Pearsin Education. 2011

[22] Visitor, https://en.wikipedia.org/wiki/Visitor_pattern