Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.1, Feb. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.1.67

함수 호출의 안전성 향상을 돕는 스마트 계약 코드 재작성기*

이 수 연,† 정 형 근, 조 은 선‡
충남대학교

Smart Contract Code Rewritter for Improving Safety of Function Calls*

Sooyeon Lee,† Hyungkun Jung, Eun-Sun Cho‡
Chungnam National University

요 약

Solidity에서 다른 계약의 함수를 호출할 때 특정 문제로 인해 호출할 수 없으면 fallback 함수가 실행된다. 이 fallback 함수는 임의로 작성될 수 있는 함수로 어떤 동작을 하는지 정해지지 않았기 때문에 함수의 동작을 알 수 없어 공격에 이용되기 쉽다. 본 논문에서는 이러한 위험성을 개발자의 부담 없이 줄이기 위해 전처리기를 이용한 해결방법을 제안한다. 개발자는 새롭게 정의된 키워드를 이용해서 의사표시를 하고, 전처리기는 해당 키워드에 따라 상태변수와 조건문을 이용해서 전처리 과정을 진행하여 위험성을 줄인다.

ABSTRACT

When a Solidity smart contract has a problem in calling a function of another contract, the fallback function is supposed to be executed automatically. However, it may be are arbitrarily created, with their behaviors unknown to developers, and fallback function execution is vulnerable to exploits by attackers. in In this paper, we propose a preprocessing based method to reduce the risk with less overhead of developers'. Developers mark the intention using the newly defined keywords in this paper, and the preprocessor reduces the risk by preprocessing the conditional variables and conditional statements according to the keywords.

Keywords: Solidity, Preprocessor, Rewritter, Fallback functions

I. 서 론

Solidity[1]란 암호 화폐 중 하나인 이더리움(ethereum)의 스마트 계약[1](smart contract)을 만들기 위한 high-level 언어 중 가장 널리 사용되는 것이다. 이더리움[2] 사용자는 Solidity를 이용해서 계약을 작성해 거래에 이용할 수 있다.

하지만 Solidity는 여러 가지 취약점을 가지고 있다고 알려져 있어, 스마트 계약을 통한 금융 거래의 피해가 우려되고 있다. 특히 다른 계약의 함수를 호출하는 부분에서 다수의 취약점이 밝혀진바 있는

Received(10. 1. 2018), Modified(12. 10. 2018), Accepted(12. 10. 2018)
* 본 논문은 2018년도 하계학술대회에 발표한 우수논문을 개선 및 확장한 것임. 또한 이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(2018-0-00251, 스마트 컨트랙트 프라이버시 보호 및 취약점 분석 기술개발)
† 주저자, djm02309@o.cnu.ac.kr
‡ 교신저자, eschough@cnu.ac.kr(Corresponding author)

1) 스마트 계약(smart contract) 또는 스마트 컨트랙트란 블록체인 기반으로 금융거래, 부동산 계약, 공증 등 다양한 형태의 계약을 체결하고 이행하는 것을 말한다.[2]
2) 블록체인 기술을 기반으로 스마트 계약 기능을 구현하기 위한 분산 컴퓨팅 플랫폼이다.[3]

데, 분산된 스마트 계약 환경에서, 호출자가 호출 동작에 대해 파악하고 제어하는 것이 쉽지 않아 피해가 발생되는 측면이 있다. 예를 들어 다른 계약의 주소를 통해 함수를 호출할 때 잘못된 주소 참조로 인하여 의도치 않은 계약의 함수가 동작되는 경우가 있다. 이 경우 해당 함수가 해당 주소에 존재하지 않는다면 이름이 없는 특별한 함수인 fallback 함수가 암묵적으로 대신 호출되는데, 그 동작을 호출자가 알 수가 없으므로 악의적인 계약이 이를 이용하여 탈취 코드를 fallback 함수에 넣고 수행되게 만들 수도 있다[4].

Solidity의 이러한 위험성을 줄이는 방법들로 계약 코드를 미리 분석하는 도구가 제안되었고[5][6], 강력한 타입을 지원하는 완전히 새로운 언어가 제시되기도 하였으며[7], 암호적인 기법을 써서 개인정보보호를 향상시키는 방법[8] 등도 제안되고 있다. 그러나 이러한 방법들은 분산 수행되는 스마트 계약의 속성을 충분히 반영하기 어려운 부분이 있고, 새로운 언어를 도입해야하는 등의 부담이 있다. 본 논문에서는 기존의 Solidity 언어 개발자들이 새로운 언어나 분석도구를 배우지 않아도 전처리기를 통해 간단히 안전성을 높일 수 있는 방안을 제안하고자 한다. 이 중에서도 함수 호출 경로 상에서 자주 발생하는 취약성을 줄이고자, 개발자의 의도에 의거하여 fallback 함수 호출을 방지하는 키워드를 계약에 추가하고 프로그램 변환으로 이를 지원하는 전처리기를 구현하였다.

본 논문의 순서는 다음과 같다. 다음 장에서는 연구 배경에 대해 고찰하고, III장에서는 취약점 완화를 위해 호출자가 fallback 함수의 사용을 제어하는 방안을 제시한다. IV 장에서는 이러한 기법이 현재 구현된 전처리기를 소개하며, V장에서는 결론을 맺는다.

II. 연구 배경

2.1 함수 호출 방식

Solidity에서 함수를 호출하는 방법은 기본적으로는 객체지향언어에서와 동일하다. 즉 스마트 계약을 나타내는 변수와 "." 및 함수 명을 조합하여 호출하는데, 예를 들어 `c.ping(42)`와 같은 호출은 변수 `c`가 어떤 계약임을 나타내고, `ping`은 함수 이름, `42`는 인자를 의미한다. 이러한 편리한 호출 방식은 Solidity 로 된 스마트 계약을 컴파일할 때 바이트코드와 함께 생성되는 함수 정보들에 근거한다. 예를 들어 `c.ping(42)`이 컴파일되면 통상 function selector로 불리는 EVM (Ethereum Virtual Machine)[9] 바이트코드 구문으로 변환되는데, 이때 해당 계약 내의 함수 이름 및 시그니쳐를 인코딩한 값 리스트에서 현재 호출된 함수 정보를 인코딩한 값을 찾아 매치되면 해당 함수를 호출하는 형태를 가진다. 이를 위해 변환된 결과 바이트코드는 고급언어의 `switch` 문과 같은 구조를 가진다.

Solidity에서는 이러한 함수 시그니처 정보가 개발 단계에서 제공되지 않는 경우를 위해 `call`, `delegatecall` 등의 저수준 명령들도 제공된다. `call`은 수행 단계에서 함수이름으로 호출할 수 있도록 해주며, `delegatecall` 방식은 `call` 방식과 대부분 동일하나 호출된 함수가 실행될 때 함수가 소속된 계약의 문맥(context)에서 수행되지 않고 호출자의 문맥을 사용한다.

Fig 1은 계약 `E`가 `call` 함수를 이용해 다른 계약의 인스턴스인 `contract_B`의 `f1` 함수를 호출하는 예이다. `contract_B`는 직접 주소 값으로 접근하고 있는데, `_value`만큼의 비용이 `contract_B`에 전달 될 수 있다.

이러한 저수준 명령을 이용하면 주어진 주소 값을 통해 다른 계약의 저장소 또는 변수에 접근하여 값을 조작할 수 있다. 하지만 이러한 방식으로 계약에 대한 기능을 호출하면 계약에 대한 보안 위험성을 수반한다. 예를 들어 `contract_A`가 나타내는 주소가 무의미한 주소이거나 계약 `B` 타입의 개체가 아닐 수도 있고, 계약 `B` 에 256bit 정수 인자를 받는 `f1`이라는 함수가 정의되지 않았더라도 개발 단계에서 확인하기 어렵다. 따라서 이러한 저수준의 호출방식은 Solidity가 의도한 것과 달리 타입 안전성

```
contract E {
   address contract_B =
   0xcf804947e5c324c61c4f91c53fa8f14a20447aeb;
   B _b = B(contract_B);
   function  Dangerous_call(uint256  _value)  {
    contract_B.call(bytes4(
          sha3("f1(uint256)")), _value);
   }
}
```

Fig. 1. Example of function call using call in Solidity

(type-safety)를 위배할 수 있기 때문에 세심한 주의가 요구된다.

2.2 Fallback 함수와 취약점

Fallback 함수는 인자와 이름과 리턴 타입이 없는 함수로서 각 계약마다 최대 1개까지 존재한다. 이 함수가 이용되는 경우는 크게 두 가지로 볼 수 있다. 첫째, 일반적인 함수 호출 시 호출된 함수와 매치되는 함수가 계약 내에 존재하지 않는 예외상황에 호출된다. 자세히 말하면, 계약 실행 중 특정 함수를 호출했을 때 해당 함수와 일치하는 시그니처(계약 내 함수이름을 SHA3한 hash값의 4바이트 값)가 없거나 함수 호출 시 타입 또는 인자가 맞지 않는 상황 등을 말한다.

예를 들어 만일 타입 A의 계약의 주소 contract_A에 대해, contract_A.call(bytes4(sha3("f1(uint256)")),_value) 와 같은 함수가 호출되었을 때 해당 f1이 계약 A에 존재하지 않는다면 자동적으로 fallback 함수가 불리게 된다. 즉, 계약 A의 정의가 Fig 2와 같고 contract_A가 A타입의 올바른 계약 인스턴스 주소를 가지고 있다고 할 때, A에 없는 f1() 함수가 호출되는 경우, fallback 함수가 불리면서 x가 1로 바뀐다.

컴파일러가 해당 계약에 특정 함수가 있다는 것을 확인한 상황에서도 fallback 함수는 호출될 수 있다. Fig 3는 Alice에 함수 ping이 존재하는지 미리 컴파일러가 확인한 경우이다. 하지만 c가 실제로 Alice의 주소인지와 Bob에 정의된 Alice 인터페이스가 실제로 수행하는 시점의 Alice의 인터페이스와 일치하는지 등은 컴파일러가 확인할 수 없기 때문에, 'c.ping(42);' 가 실행될 때 이 함수와 시그니처가 일치하는 함수가 없는 상태가 되어 여전히 예외상황이 발생할 가능성이 있다. 그러면 fallback 함수가 자동 실행된다.

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

Fig. 2. Example of the codes that might be call Fallback function

```
contract Alice {
   function ping(uint) returns (uint)
}
contract Bob {
   function pong(Alice c) {c.ping(42);}
}
```

Fig. 3. Example of smart contract[1]

둘째, 해당 계약이 다른 곳으로부터 ether를 받고자 하는 경우에도 fallback 함수가 수행된다. 스마트 계약에서 ether를 이동시키는 방법은, 받을 계좌의 transfer 또는 send를 호출하거나, payable이라는 키워드가 명시된 함수 내에 ether 이동을 명시하고 호출하여 수행하는 방법이 있다. 이동시킬 금액은, 전자의 경우 호출시 전달되는 파라미터인 value를 통해 명시되며, 후자의 경우 페이로드인 함수 인자 (data) 를 통해 전달받게 된다. 이 때 전자의 경우 수신 계좌의 fallback 함수가 수행되며 이를 위해서 fallback 함수는 payable로 설정되어 있어야한다. 이 때 만일 fallback 함수가 payable로 설정되어 있지 않으면 예외 상황이 발생된다. 또한 fallback 함수를 구현하지 않은 계약에서 fallback 함수가 호출되어야하는 상황이 발생하는 경우도 예외상황이 발생되며 ether를 돌려보낸다.

문제는, 호출자가 fallback 함수가 호출되었는지 여부와 fallback 함수의 동작을 명시적으로 알기 힘들기 때문에, 스마트 계약의 수행이 개발자가 의도한 대로 되지 않을 수 있으므로 위험성이 내재되어 있다는 것이다. 예를 들어, 만약 악의적인 목적을 가진 사용자가 위와 같은 계약을 작성하고 fallback 함수 내에 고의로 failure를 유발하는 코드를 넣으면 계약의 정상적인 진행이 멈출 수 있으며, 악성코드를 넣어둔 후에 계약이 실행된다면 본래의 fallback 함수의 기능 대신 악성코드가 실행될 가능성이 있다.

2.3 기존 연구

Remix등[10]의 널리 사용되는 Solidity 개발환경에서는 계약 프로그램의 패턴에 대해 경고를 발생시킨다. Oyente[5], DappGuard[6], ZEUS[11] 등은 정적 동적 분석을 통해 스마트 계약의 코드 보안을 점검한다. 싱가포르 대학의 Oyente는 오픈소스 보안 검사 도구로, symbolic execution[12]를 기반으로 검사를 수행한다. 입력값에 대한 제약 조건을 수집하고, 수집한 제약들을 가지고 CFG(Control Flow Graph)를 통하여 프로그램이 에러 상태에 도달할 수 있는지 Z3

Bit-Vector Solver[13]를 통하여 증명한다. DappGuard 또한 스마트 계약 코드를 점검하는 도구이다. 내부적으로 Oyente를 이용하여 정적분석을 진행하고 있으며 Oyete보다 점검하는 취약점의 종류가 좀 더 많다. ZEUS는 스마트 계약 코드를 비트코드로 바꾸고 사용자가 지정한 정책을 바탕으로 계약의 취약점을 분석한다. 하지만 스마트 계약 코드를 LLVM[14] 비트코드로 변환하는 과정에서 스마트 계약만의 특징이 생략되면서 정확한 분석을 하지 못한다는 한계가 지적되기도 한다. 이러한 도구들은 스마트 계약에 대해 취약점이 존재하는지 분석 후 사용자에게 결과를 보여주게 된다. 역컴파일러들은 EVM 바이트코드로 존재하는 계약들을 Solidity 언어로 바꾸어 보여줌으로써 해당 계약을 이용하려는 이용자들이 내용을 확인하는 데에 도움을 주게 된다[15].

그러나 보다 적극적으로 코드를 안전하게 바꿔주거나 강제하는 방법은 많지 않다. Solidity의 `require` 구문[16]을 활용하는 것도 코드의 안전성을 높이는 데에 크게 일조하지만 해당 구문의 표현력의 제약으로 인해 fallback 함수 호출 허용 여부 등을 표시하는 것은 가능하지 않다.

III. 제안하는 전처리 기법

본 논문에서는 함수를 호출 할 때 해당 함수의 수행 중 fallback 함수의 호출을 방지하는 코드 전처리 방안을 제안한다. 개발자는 호출되는 함수 수행 중 fallback 함수가 호출되지 않도록 하는 의도를 명시하고, 만일 fallback 함수가 호출된 경우 비정상 종료를 하게 된다. 이를 위해 개발자는 민감한 함수 수행에 대해 `NONFALLBACK` 키워드를 명시한 Solidity 코드를 작성한다. 이것을 입력으로 하는 전처리기는 안전한 수행결과를 주는 Solidity 코드를 주게 된다.

Fig 4의 `solidity_revised.g4`는 수정된 Solidity 문법을 의미하며, 파서생성기인 ANTLR[15]의

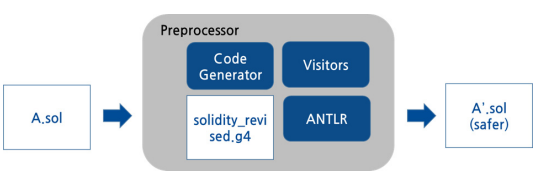

Fig. 4. Overall process

```
1: contract Alice {
2:   function ping(unit) {returns (uint)}
3: contract Bob {
4:     function pong(Alice c) {
5:       NONFALLBACK c.ping(42);
6:      }
7:    function () payable { }
...
```

Fig. 5. Example of the codes with the keyword NONFALLBACK (before preprocessing)

```
...
contract Bob {
     function pong(Alice c) {
        c.ping(42);
        if (fb == FALLBACK) {
          // error
        }
     }
    function () payable {
      fb = FALLBACK;
    }
...
```

Fig. 6. Example of the conceptual codes with the keyword NONFALLBACK (after preprocessing)

Visitor 인터페이스를 구현하여 함수 호출 코드를 식별하고 개발자에게 알리는 코드를 생성 삽입하는 형태로 재작성하고 있음을 의미한다. 전처리기에 의해 수정된 프로그램은 수행 도중 fallback이 호출된다면 계약의 실행이 멈추게 되며, 테스트 용도로 사용되는 경우에는 fallback 함수의 호출 여부가 개발자에게 알려지고 수정을 유도하는 효과가 있다.예를 들면 개발자가 Fig 5의 5번째 줄과 같이 키워드(`NONFALLBACK`)를 통해 fallback 함수가 불리지 않기를 원하는 의사 표시를 하면, 전처리기는 Fig 6에 나타난 바와 같이, fallback 함수가 불릴 경우 오류로 나타내주는 코드를 인위적으로 삽입한다. 즉, 이러한 조건문을 추가함으로써 fallback 함수가 호출되었는지 확인하게 한다. 추가된 조건문을 활용하기 위해 Solidity 코드에 fallback 함수의 내부에 특정 상태변수 `fb`를 지정하는 코드를 추가하여 이용하게 된다.

이러한 방식으로 코드를 재작성하고 나면, 스마트 계약 실행 중 fallback 함수가 실행될 경우 상태변수 `fb`가 설정되고, `if` 조건문에서 `fb`가 설정된 것을 확인한 후 개발자에게 알려주게 된다. 따라서 개발자

는 자신이 코드 수정이 없이 키워드(NONFALLBACK)만 이용하면 전처리기를 사용하게 됨으로써 보다 안전한 코드를 작성할 수 있게 된다.

IV. 구 현

본 장에서는 제안한 기법에 대한 실제로 구현한 내용을 소개한다. 우선 새로운 키워드를 인식을 위해 Fig 7과 같이 solidity_revised.g4를 제작하였다. 또한 NONFALLBACK 키워드를 함수 호출의 앞에 명시하기 위해 함수 호출 구문의 수정도 필요하다.

그 후 ANTLR 파서를 이용해 코드를 재작성하는 전처리기를 작성하였다. 개발자가 작성한 Solidity 파일이 전처리기에 입력으로 들어오면 ANTLR를 이용한 파서와 코드 생성기가 전처리 과정을 진행한다. 파서가 NONFALLBACK 키워드를 인식한 경우, 키워드를 포함하고 있는 함수 호출 부분 다음 위치에 fallback이 호출되는 경우 오류를 발생시키는 코드를 삽입한다. 또한 fallback 함수 내부에 fallback 함수가 불리었음을 표시를 할 수 있도록 전역 플래그를 삽입한다. 그런데, 여기서 fallback 함수의 호출 여부가 호출자에 전달되려면 정보를 저장하는 전역변수가 필요한데 Solidity에서는 전역변수를 임의로 지정할 수 없다는 한계가 있다.

따라서 제안하는 전처리기의 결과 코드에서 사용하기 위해서 Fig 8과 같이 ForFallback이라는 별도의 Solidity 라이브러리(library)를 추가로 작성하여 삽입하였다. 라이브러리 대신 별도의 계약으로 만들고 전역변수를 포함하게 하는 방법도 고려하였으나, 동일한 코드를 재사용하는 것을 목적으로 만들어진 라이브러리가 계약 보다 적합한 면들이 있었다. 예를 들어, 계약의 변수를 변경하기 위한 함수는 가스의 소모가 필연적인 반면 라이브러리는 추가 비용이 들지 않아 플래그를 지정하는 데에 용이하다.

단, 라이브러리는 안전한 재사용을 위해 하나의 인스턴스만 존재하는 것을 가정하므로 함수의 지역변수를 가지지 못한다. 따라서 라이브러리 ForFallBack

```
elemetaryTypeName
  : 'address' | 'bool' | 'string' | 'var' | Int
| Uint | 'byte' | Byte | Fixed | Ufixed |
NonfallbackKeyword;
NonfallbackKeyword: 'NONFALLBACK';
```

Fig. 7. solidity_revised.g4 file with the additional keyword **NONFALLBACK**

```
library ForFallBack {
    struct FallBackFlag{ bool fb; }

    function getFB(FallBackFlag self) returns
(bool) {
        return self.fb;
    }
    function setFB(FallBackFlag self) returns
(bool) {
        self.fb = true;
        return self.fb;
    }
}
```

Fig. 8. Definition of library ForFallBack

에서는 정보가 담겨있는 변수를 타입 struct ForFallBack의 self 변수로 넘겨받아 처리하게 되며, 해당 변수는 호출자에서 정의하게 된다. getFB는 해당 플래그 fb의 값을 읽어주고, setFB는 true로 정의한다.

전역 변수 대신 새로운 계약을 작성했으므로 기존에 존재하는 계약에, 라이브러리 ForFallback 의 각 함수에 self로 전달할 변수 fallBackFlag의 선언을 추가한다. 모든 계약마다 이러한 추가가 필요하므로 추상 상위타입으로 정의하고 각 계약을 계승 받는 것으로 간략화 시킬 수 있다.

그림 Fig 10의 Alice와 Bob의 정의에서 보이는 것처럼, 모든 계약은 Fig 9와 같이 정의된 NonFallBackEnabled을 계승하도록 만들고, 플래그의 getFB와 setFB를 전달할 함수를 각각 정의한다. NONFALLBACK 키워드가 직접 명시된 ping의 호출에는 피호출된 계약의 플래그도 함께 확인한 후 호출자의 플래그에 반영한다. 이 때 대상이 되는 함수 호출은 fallback 함수관련 플래그의 getter, setter와 무관한 함수로 한정한다.

구현한 결과는 다음과 같이 나타난다. Fig 11은

```
contract NonFallBackEnabled {
    using ForFallBack for
            ForFallBack.FallBackFlag;
   ForFallBack.FallBackFlag fallBackFlag;

   constructor() payable {
         fallBackFlag = false;
   }
...
}
```

Fig. 9. Definition of contract NonFallBackEnabled

```
contract Alice is NonFallBackEnabled {
...
    function ping(unit) {returns (unint)}
    function () payable {
      setFB(flag);
    }
    function getFB() returns (bool) {
        return fallBackFlag.getFB();
    }
    function setFB() returns (bool) {
        return fallBackFlag.setFB();
    }
...
}
contract Bob is NonFallBackEnabled {
    function pong(Alice c) {
        c.ping(42);
        if (getFB() == true) {
          revert(“error”);
        }
     }
    function () payable {
      setFB();
    }

    function getFB() returns (bool) {
        return fallBackFlag.getFB();
    }
    function setFB() returns (bool) {
        return fallBackFlag.setFB();
    }
...
}
```

Fig. 10. Revised code of contract Bob

전처리기에 입력으로 들어갈 간단한 스마트 계약 코드의 예시이다. 이와 같은 코드가 전처리기를 거치면 Fig 12와 같이 재작성 된다. Sink 계약과 Test 계약

```
pragma solidity ^0.4.5;
contract Test {
   function() public { x = 1;}
   uint x;
}
contract Sink {
   function() public payable {}
}
contract Caller {
   function callTest(Test test) public {
      NONFALLBACK test.call(0xabcdef01);
   }
}
```

Fig. 11. Example of simple smart contract code entered as input to preprocessor

```
pragma solidity ^0.4.5;
contract Test is NonFallBackEnabled {
   function() public { setFB(flag); x = 1;}
   uint x;

    function getFB() returns (bool) {
        return fallBackFlag.getFB();
    }
    function setFB() returns (bool) {
        return fallBackFlag.setFB();
    }
...
}

contract Sink is NonFallBackEnabled {
   function() public { setFB(flag); }

    function getFB() returns (bool) {
        return fallBackFlag.getFB();
    }
    function setFB() returns (bool) {
        return fallBackFlag.setFB();
    }
...
}
contract Caller is NonFallBackEnabled {
   function callTest(Test test) public {
      NONFALLBACK test.call(0xabcdef01);
      if (getFB() == true)  {
          revert(“error”);
       }
     }
    function () payable {
      setFB();
    }

    function getFB() returns (bool) {
        return fallBackFlag.getFB();
    }
    function setFB() returns (bool) {
        return fallBackFlag.setFB();
    }
   }
}
contract NonFallBackEnabled {
    using ForFallBack for
            ForFallBack.FallBackFlag;

    ForFallBack.FallBackFlag fallBackFlag;

   constructor() payable {
         fallBackFlag = false;
   }

...
}
library ForFallBack {
    struct FallBackFlag{ bool fb; }

    function getFB(FallBackFlag self) returns (bool) {
        return self.fb;
    }
    function setFB(FallBackFlag self) returns (bool) {
        self.fb = true;
        return self.fb;
    }
}
```

Fig. 12. Example of simple smart contract code results through preprocessor

은 fallback 함수 내부에 플래그 값을 변경하는 부분을 포함하고 있다. NONFALLBACK 키워드를 포함한 ‘test.call’ 다음에는 위에서 언급했던 조건문 등이 추가된 것 또한 확인할 수 있다. 또한 모든 계약은 상위 계약으로 NonFallBackEnabled를 두고 계승 받고

있는 것을 확인할 수 있다.

구현한 결과에 대한 성능은 아래 표와 같다.

Oyente와 ZEUS의 경우 개발자가 분석기를 이용해서 따로 테스트를 해야 하는 별도의 노력이 필요한 도구이다. 본 논문의 전처리 방법과 `require`문은 개발자가 개발과정에서 키워드를 이용해 원하는 부분에 삽입하는 것으로 충분하다. 처리 시간 면에서는 Oyente는 350초, ZEUS는 약 60초의 테스트 시간이 필요하다는 것이 알려져 있다. 반면 제안하는 전처리 방법은 논문에 포함된 예제의 경우 0.7초 정도의 전처리 오버헤드가 드는 것으로 측정되었다. Solidity 언어 고유의 기능인 require문을 사용하는 경우를 제외하면, 제안하는 전처리 방법의 오버헤드가 더 적다는 것을 알 수 있다. 본 논문의 전처리 방법은 현재 fallback함수에만 초점을 두고 구현이 되어 있는 반면, Oyente와 ZEUS는 각각 4가지와 8가지의 취약점에 대해서 동작하고 있으므로 현재 다루고 있는 적용범위로 보면 더 넓은 편이다. 향후에는 제안하는 전처리 기법이 fallback함수 외에도 처리할 수 있도록 적용범위를 넓힐 계획이다. `require`문은 Solidity의 일반적인 변수나 값으로 된 조건식으로 불린 식을 표현하는 것이므로, 취약점과 관련된 추상화된 상태의 표현에 적용되는 경우에는 표현이 거의 불가능하거나 복잡도가 높아지는 한계가 있다.

Table 1. Comparison with other tools

	Developer's overhead	Processing time	Target vulnerabilities
Oyente	Testing via the analyzer	350s (testing time)	Mishandled exceptions, Reentrancy Handling and more
ZEUS	Testing via the analyzer	60s (testing time)	Integer over or underflow, Reentrancy, transaction state dependence and more
Proposed Method	Specifying `NONFALLBACK` in development phase	0.715s (preprocessing time)	fallback function invocation (currently) and custom vulnerabilities described with a new keyword (future)
`require` in Solidity	Specifying `require` in development phase	N/A	custom vulnerabilities described with plain conditional statements

V. 결 론

Solidity에서 다른 스마트 계약의 함수를 호출할 때 예외상황이 발생하거나 계약 간 금전적 이동이 있을 때 fallback 함수가 실행되는 경우가 많다. 이 fallback 함수는 임의로 작성될 수 있는 함수로 어떤 동작을 하는지 알 수 없어 공격에 이용되기 쉽다. 본 논문에서는 전처리기를 이용해 fallback 함수의 내용에 특정 상태변수를 지정하는 코드를 추가하고, 함수를 호출할 때 조건문을 이용해 함수를 호출한 계약에게 알리는 방법으로 안전성을 향상시키는 방안을 제안하였다. 이로써, fallback 함수의 호출 여부를 개발자가 알기 쉽게 하여 호출 여부를 파악 수 있어 코드 수정을 유도하는 장점이 있다.

제안하는 방법은 소스코드 레벨에서 fallback 함수를 판별하기 때문에 기본적인 제약사항 및 오버헤드가 발생한다. 향후에는 이 부분을 보완하기 위해 소스 코드 외에도 바이트 코드의 재작성을 진행할 예정이며 이를 위해 Solidity 컴파일러와 EVM을 개선할 계획이다.

References

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

[2] smart contract. "smart contract", https://ko.wikipedia.org/wiki/스마트계약

[3] Ethereum. "ethereum", https://www.ethereum.org/

[4] 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, pp. 164-186, Apr. 2017

[5] 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, pp. 254-269, Oct. 2016

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

[7] 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

[8] 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

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

[10] Remix, "remix", https://remix.ethereum.org/

[11] 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

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

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

[14] LLVM, "LLVM", https://llvm.org/

[15] Solidity Decompiler, "solidity decompiler", https://ethervm.io/decompile

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

[17] ANTLR, "ANTLR" http://www.antlr.org/

〈저자소개〉

이 수 연 (Sooyeon Lee) 학생회원
2018년 2월: 충남대학교 컴퓨터공학과 졸업
2018년 3월~현재: 충남대학교 컴퓨터공학과 석사과정
〈관심분야〉 프로그래밍언어, 프로그램 분석

정 형 근 (Hyungkun Jung) 학생회원
2017년 2월: 충남대학교 컴퓨터공학과 졸업
2017년 3월~현재: 충남대학교 컴퓨터공학과 석사과정
〈관심분야〉 프로그래밍언어, 이벤트 처리

조 은 선 (Eun-Sun Cho) 정회원
1991년: 서울대학교 계산통계학과 학사
1993년: 서울대학교 전산과학과 석사
1998년: 서울대학교 전산과학과 박사
1999년~2010년: 한국과학기술원 연구원
2000년~2001년: 아주대학교 정보통신전문대학원 조교수 대우
2002년~2006년: 충북대학교 조교수
2006년~현재: 충남대학교 컴퓨터공학과 교수.
〈관심분야〉 프로그래밍언어, 프로그램 분석, 이벤트 처리