Smart Contract Vulnerability Detection Based on Symbolic Execution Technology

Yiping Liu(✉), Jie Xu, and Baojiang Cui

Department of Cyberspace Security,
Beijing University of Posts and Telecommunications, Beijing, China
linkleep@bupt.edu.cn

Abstract. With the rapid development of the blockchain, smart contract technology has been widely applied. The number of smart contracts has grown at a high rate and nearly at an average of thousands per day. However, the correctness and security of the smart contract itself are facing huge problems. The well-known DAO vulnerability, and Parity multi-signature wallet' vulnerabilities have leaded to a hundreds of millions dollars loss, and they are both caused by the security problems of smart contracts. Once the smart contract vulnerability is exploited, it is very likely to bring the loss of cryptocurrencies, the disorder of the financial order and other catastrophic consequences. Therefore the security of smart contracts is imminent. This project has designed and implemented a vulnerability detection system of Ethereum smart contract. The system uses the assembly instruction sequences of the smart contract to generate the control flow graph, then performs symbolic execution and vulnerability constraint solving over the control flow. The system can detect some common types of vulnerabilities, such as the integer overflow and underflow vulnerability, reentry vulnerability and unchecked call return value vulnerability. It has a high accuracy of detection result, and gives support for export vulnerability report.

Keywords: Ethereum · Smart contract · Control flow · Symbolic execution · Vulnerability detection

1 Introduction

With the rise of Bitcoin, blockchain technology has gradually appeared in people's vision. In April 2014, Gavin published the Yellow Paper of Ethereum [1] and the concept of smart contracts began to spread widely. Ethereum is an open source decentralized blockchain platform, mainly used for the execution of smart contracts. Smart contracts are programs deployed on the Ethereum network and executed by the Ethereum virtual machine. The Ethereum consensus protocol guarantees the fairness of contract execution.

Smart contract technology is widely used in various fields such as infrastructure, commercial retail, games, social media and communications because of its

W. Lu et al. (Eds.): CNCERT 2021, CCIS 1506, pp. 193–207, 2022.
https://doi.org/10.1007/978-981-16-9229-1_12

safety, reliability, fairness, and efficiency characteristics. At the same time, the security of smart contracts is also facing huge challenges.

In June 2016, the DAO security breach broke out, which caused a loss of 60 million dollars. The first vulnerability of parity multi-signature wallet resulted in a $30 million loss, and the second vulnerability led to a freezing of $100 million. So far, the losses caused by the security issues of smart contracts have ranged from 30 million to 152 million dollars, and the upper limit number is still growing.

The security issues of smart contracts have emerged rapidly in the past two years. How to judge the correctness and security of the smart contract codes effectively has become an important direction of today's blockchain security research.

This paper analyzes the characteristics of Ethereum smart contract vulnerabilities and proposes a smart contract vulnerability detection technology based on symbolic execution and constraint solving. Experimental results show that the technology can detect common vulnerabilities in 1552 different contracts with high accuracy.

This article is mainly divided into five parts. Section 1 mainly introduces the background and summary of this article; Sect. 2 introduces related work; Sect. 3 introduces the most current types of vulnerabilities in smart contracts; Sect. 4 introduces framework design and vulnerability detection details of our system; Sect. 5 introduces the experimental results of our vulnerability detection, the last section summarizes our main contributions

2 Related Work

At present, there have been a lot of related work on smart contract vulnerability detection, and the main methods adopted are fuzzing testing, symbolic execution, formal verification and other technologies. Based on the special operating environment, life cycle and program characteristics of smart contracts, these studies have improved existing program analysis techniques to achieve better automated vulnerability mining effects.

Oyente [2] is one of the earliest researches on automated smart contract vulnerability mining. It takes smart contract bytecode as input and uses four components including CFG builder, explorer, core analysis and validator to perform CFG construction, symbolic execution, constraint solving, and false alarm filtering. Oyente can detect common types of vulnerabilities such as integer overflow vulnerabilities and stack overflow vulnerabilities of smart contracts.

Osiris [3] conducts further research and development on the basis of Oyente, using symbolic execution technology to detect integer vulnerabilities in smart contracts, mainly detailed to the types of vulnerabilities such as integer overflow, symbol conversion and so on. Compared with Osiris, Oyente pays more attention to the arithmetic operation instructions in the smart contract. At the same time, it introduces the taint analysis technology to mark the source and transfer direction of the operands, filter out invalid vulnerabilities that cannot be exploited, and improve the accuracy of the vulnerability detection results.

Echidna [4] is one of the earliest open source smart contract fuzzing solutions. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions. Testers need to add specific detection code to the smart contract source code in order to judge whether there is a vulnerability based on the return status of Echidna.

Another fuzzing program, ContractFuzzer [5], is mainly aimed at smart contract vulnerability detection. It can generate fuzz test inputs according to the ABI specification of smart contracts, define test oracles for detecting security vulnerabilities. And record the run-time state of the smart contract, analyze the log and report security vulnerabilities through the Ethereum Virtual Machine (EVM) instrumentation. ContractFuzzer tools include an offline EVM instrumentation tool and an online fuzzing tool. The offline EVM instrumentation tool enables the fuzzing tool to monitor the execution of smart contracts and extract execution logs for vulnerability analysis by instrumenting the EVM.

ZEUS [6] uses formal verification methods to detect smart contract vulnerabilities. It applys abstract interpretation and symbolic execution to automate the formal verification of smart contracts. And it also uses a smart contract written in a high-level language as input and user assistance to generate a standard for the correctness and fairness of the XACML template style.

It translates these contracts and specific guidelines into a low-level intermediate representation, such as LLVM bytecode, and encodes the execution semantics to correctly infer contract behavior. Then static analysis is performed on the intermediate code to determine the predicates that must be declared for verification. Finally, ZEUS puts the modified IR into the verification engine, which uses CHCs to quickly verify the security of the smart contract.

Securify [7] also uses formal verification methods to detect the vulnerabilities of smart contracts, which can analyze whether there are vulnerabilities in smart contracts with given characteristics. Securify derives the dependency graph by analyzing the bytecode of the smart contract. Then, according to the given characteristics, it analyzes whether the semantic information of the contract satisfies or violates these characteristics and judges whether there are loopholes in the contract. The input of Securify is the bytecode of the smart contract and a series of patterns, which are described in domain-specific language. The output is the location of the specific vulnerability. Users can write patterns by themselves, so Securify is extensible.

Teether [8] uses automatic injection to detect contract vulnerabilities. It looks for critical paths in the control flow diagram of the contract, and then determines the key instructions whose parameters can be controlled by the attacker. Once a path is determined, symbolic execution is used to convert this path into a series of constraints. Using the constraint solver, you can infer what transactions the attacker must perform to trigger the vulnerability.

3 Background

3.1 Reentrancy Vulnerability

Ethereum smart contracts can call and utilize the codes of other external contracts. When the contract executes the transfer operation, if the counterparty account is a contract account, the callback function in the contract account will be called. If the called contract is a contract constructed by an attacker, there is likely to be malicious code in it. The DAO attack exploited the reentrance loopholes in the contract code, causing economic losses of up to 60 million dollars.

Figure 1 is an example of an error when you forget to check the return value.

```
1  contract EtherStore
2  {
3      uint256 public withdrawLimit = 1 ether;
4      mapping(address => uint256) public balances;
5      function depositFunds() public payable
6      {
7          balances[msg.sender] += msg.value;
8      }
9      function withdrawFunds(uint256 _weiToWithdraw) public
10     {
11         require(balances[msg.sender] >= _weiToWithdraw);
12         require(_weiToWithdraw <= withdrawLimit);
13         require(msg.sender.call.value(_weiToWithdraw)());
14         balances[msg.sender] -= _weiToWithdraw;
15     }
16 }
```

Fig. 1. An error instance of forgetting to detect the return value.

The code in line 13 can be used to transfer money to the msg.sender account, but if msg.sender is a contract account, it will call the callback function in the destination contract to perform the transfer operation. The attacker can construct a special callback function to cut off the control flow. So that the contract will continue to execute lines 11–13 of code without executing 14, then the condition of line 11 will be met forever, until the attacker takes out all the balance in the contract. Figure 2 is the attack contract constructed for this EtherStore contract. Lines 15–21 are the special callback function constructed by the attacker. When the 13th line of the EtherStore code is executed, the attack's callback function will be called, and the attack contract will call the withdrawFunds function of EtherStore when the 19th line is executed. The 14th line in EtherStore is not executed, so the 11th line still meets the conditions to achieve reentry, and continuous reentry can take out all the balance in the EtherStore contract.

```
import "EtherStore.sol";
contract Attack
{
    EtherStore public etherStore;
    constructor(address _etherStoreAddress)
    {
        etherStore = EtherStore(_etherStoreAddress);
    }
    function pwnEtherStore() public payable
    {
        require(msg.value >= 1 ether);
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }
    function () payable
    {
        if(etherStore.balance > 1 ether)
        {
            etherStore.withdrawFunds(1 ether);
        }
    }
}
```

Fig. 2. An attack contract target for EtherStore contract.

3.2 Integer Overflow Vulnerability

The Ethereum Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable can only be represented by a certain range of numbers, respectively (u)int8/16/24/.../256. For example, a uint8 can only store numbers in the range [0,255]. Attempting to store 256 into a uint8 will become 0. Therefore, performing calculations without checking user input can easily occur the calculation result exceeds the maximum range that the variable type can represent. This situation is called integer overflow or underflow. Integer overflow vulnerabilities can be easily exploited by attackers to perform logic processes that developers did not anticipate. Figure 3 is a contract with an integer overflow vulnerability.

If an attacker maliciously calls the increaseLockTime function to cause the lockTime variable overflowed in line 12. Then the attacker can break the time limit and call the withdraw function to successfully withdraw the account balance. Figure 4 is a contract with integer underflow vulnerability.

The balance variable on lines 11 and 12 may underflow. When the attacker does not deposit money in the contract, the value of balance is 0. Then the attacker calls the transfer function and sets *_value* to any positive integer, the balance-value will underflow and becomes a very large positive integer.

```
contract TimeLock
{
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;
    function deposit() public payable
    {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] += now + 1 weeks;
    }
    function increaseLockTime(uint _seconds) public
    {
        lockTime[msg.sender] += _seconds;
    }
    function withdraw() public
    {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balances[msg.sender]);
    }
}
```

Fig. 3. A contract with an integer overflow vulnerability

```
contract Token
{
    mapping(address => uint) balances;
    uint public totalSupply;
    function Token(uint _initial)
    {
        balances[msg.sender] = totalSupply - _initial;
    }
    function transfer(address _to, uint _value) public
returns (bool)
    {
        require(balances[msg.sender] - _value >= 0);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        return true;
    }
    function balanceOf(address _owner) public constant
returns (uint balance)
    {
        return balances[_owner];
    }
}
```

Fig. 4. A contract with integer underflow vulnerability

3.3 Unchecked Call Return Value Vulnerability

Some methods in solidity can send Ether to external accounts. For example, transfer() method, send() method, call method, etc. The call() and send() functions will return a boolean value after completion. The return value is true to indicate the operation is successful, and false to indicate failure. So when the send or call function fails, the contract will not terminate execution and complete the state rollback, but will simply return false. Therefore, without checking the return value of the send or call operation, it is likely that the result of inconsistent intentions of the developer may occur. Figure 5 is an example of an error when you forget to check the return value.

```
contract Call_Vul
{
    // ... local variables definitions
    function withdraw(uint256 _amount) public
    {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
    }
}
```

Fig. 5. An error when you forget to check the return value

If the call is used to send ether to a smart contract that does not accept them (for example, because it does not have a fallback function) or a callback occurs during the call to an external smart contract, the send operation may run out of gas and cause the send to fail and return false. Since the return value is not checked in our example, even though msg.sender has not received the money, the balance in the account has decreased.

4 Vulnerability Detection Methods

In this section, we will introduce our vulnerability detection system in detail. Our system is designed to detect common types of vulnerabilities such as reentrancy vulnerability, integer overflow vulnerability, and unchecked call return value vulnerability in smart contracts. Our system uses bytecode, the address or the source code of a smart contract as input, and outputs the vulnerability detection result, including specific information such as the type and the location of the vulnerability. Figure 6 depicts the workflow of our vulnerability detection system. It mainly includes the following five modules.

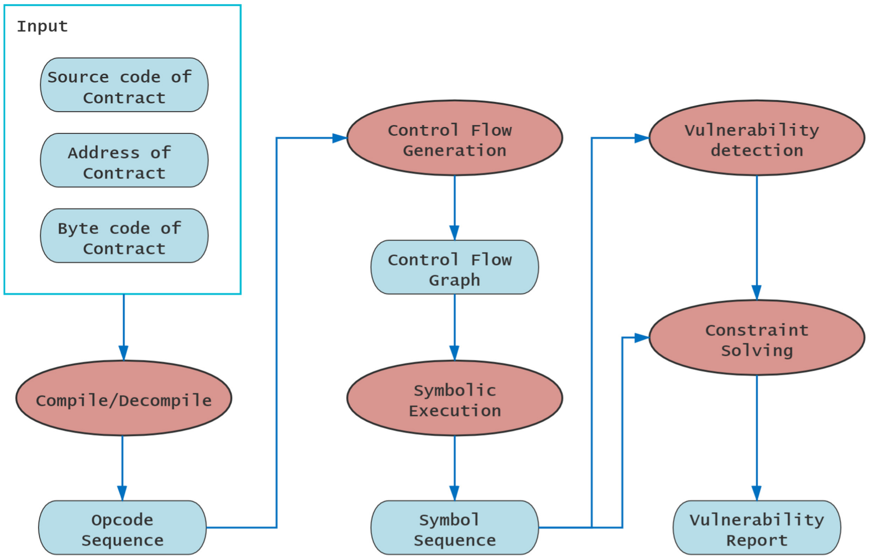

Fig. 6. Workflow of our vulnerability detection system.

Compile and Decompile. The input of the system may be solidity code or EVM bytecode, so it needs to be compiled or decompiled to generate the opcode sequence. The generated instruction sequence includes instruction offset, instruction name and instruction parameters.

Control Flow Generation. The control flow graph is composed of the basic block and the edge between them. The basic block is analyzed and constructed from the first instruction, and it is ended when *JUMP* instruction is encountered and the next basic block is generated. The basic block is ended when the *JUMPI* instruction is encountered and it will generate two basic blocks with different conditions then record the rules that the jump conditions need to meet.

Symbolic Execution. Our system constructs a virtual machine executed by the EVM bytecode and use the control flow graph in the control flow generation module to traverse all possible paths and record the possible results of each step according to Depth-First-Search principle. At the same time, the constraint conditions of each step and the operand will be transformed into the variable type of constraint solving.

Vulnerability Detection. Vulnerability detection is synchronized with the execution of the contract in the virtual machine. If a sensitive operation that may trigger the vulnerability condition is encountered during the execution, it will jump into the vulnerability detection module. The vulnerability detection module will solve the jump conditions and vulnerability status. The existence of possible explanations indicates the vulnerability may exist.

Constraint Solving. The constraint solving module provides assistance for the virtual executor module and the vulnerability detection module. It is mainly used to convert variables into z3 type variables and to solve various conditional constraints.

4.1 Control Flow Generation

The control flow generation module is used to determine all possible path conditions during program execution. The edges of the control flow graph represent each basic block in the process of program execution, and the edges between nodes represent the jump conditions between nodes. The generation of control flow graphs is not necessary in the process of symbolic execution, but generating control flow graphs can help better understand the calling relationship between programs or perform the next step of analysis. The detailed of control flow generation process is as follows:

1. After generating the opcode sequence, fetch an instruction from it.
2. Determine whether this instruction is *JUMP* or *JUMPI*.
3. If it is *JUMP*, take out the parameter which is the target address of this instruction.
4. If it is *JUMPI*, take out the parameters which include jump conditions and jump addresses.
5. If it is *CALL/CALLCODE/DELEGATECALL/STATICCALL*, you first need to set this address as the end of current node.
6. If it is *RETURN* instruction, record the offset of this instruction as the end of current node. Check whether the next instruction is the last instruction or *JUMPDST*, if not, throw an error.
7. If the instruction is *JUMPDST*, traverse all nodes to find the starting address of this *JUMPDST* instruction, and set the currently executing NodeID as the ID of this node.

4.2 Symbolic Execution

Based on the control flow graph, the symbolic execution module uses Depth-First-Search principle to execute each node in the graph, and updates the global state at the same time. When encountering a conditional jump instruction or an operand operation instruction, converted it into a variable type for constraint solving, so that the vulnerability detection module can solve the constraint for the vulnerability. The symbolic execution module is divided into three parts: the state design of the virtual machine, the instruction realization of the virtual machine and the path call of the symbolic execution.

1. State design of the virtual machine. Virtual machine of the system is modeled on the Ethereum virtual machine(EVM). After each transaction is executed, the state of the EVM, including the program counter, the stack, and account balance may change. Figure 7 is an example of the internal state transition of the virtual machine after a transaction is completed. The transition

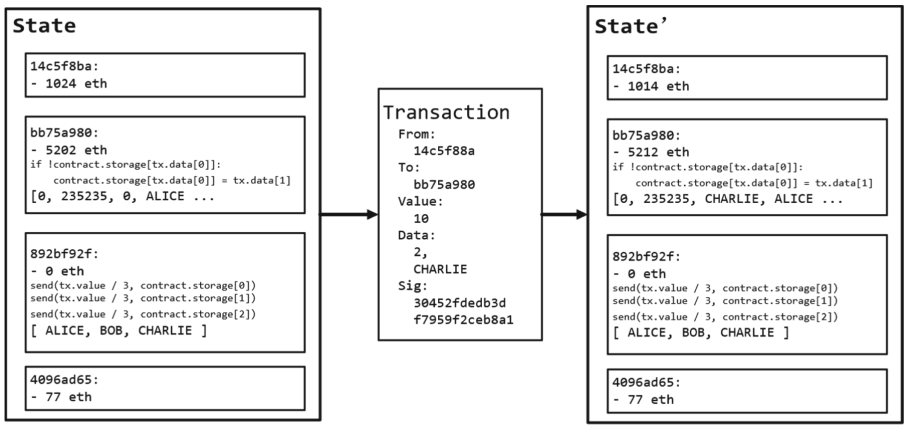
State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ...
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ...
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Fig. 7. An example of the EVM internal state transition after a transaction is completed.

models of these three states are implemented separately in the virtual machine we built.

2. Implementation of virtual machine instructions. The current EVM instructions have implemented a total of 142 instructions. According to the instructions of the EVM, our system has implemented 117 commonly used instructions.
3. The principle of path calling. Before symbolic execution of control flow graph, the calling principle of the path must be determined first. Here we use the Depth-First-Search principle. Differ from the general graph traversal progress, we need to save the state of this path when it is ended.

4.3 Vulnerability Detection

We will introduce the details of reentrancy vulnerability, integer overflow vulnerability and unchecked call return value vulnerability detection method separately.

Reentrancy Vulnerability Detection. The main reasons of why reentrancy vulnerabilities exist are as follows:

1. When using the *call* instruction to transfer money to a contract account, the contract's callback function will be called.
2. The *call* instruction does not restrict the use of Gas by default
3. Complete the transfer operation before reducing the user account balance.

The *call* instruction has 7 parameters, the meaning of each parameter is shown in Table 1. Considering the gas limit of the other two transfer operations *transfer* and *send* are both 2300, if the gas can be greater than 2300, it is equal

Table 1. Parameters of call instruction.

Parameter	Meaning
Gas	Gas limit
Address	Target address of the transfer operation
Value	Transfer amount
In	Input data address of the call instruction in EVM memory
Insize	Length of input data
Out	Output data address of the call instruction in EVM memory
Outsize	Length of output data

to no restriction. So we use $gas > 2300$ as a vulnerability constraint. And if the address can be equal to the malicious contract address, there may be a reentrancy vulnerability.

Integer Overflow Vulnerability Detection. Integer vulnerabilities has been widespreaded and brought a lot of loss. In EVM, there may be integer overflow operations such as *add*, *sub*, *mul*, and *div*. The detection for these four operations are shown in Table 2.

Table 2. Vulnerability detection methods for integer operation.

Op	Detection
ADD	Under the constraints of entering this node, if the sum of two operands is less than one of the operands is solved, there may be an overflow error
MUL	Under the constraints of entering this node, if the multiplication of two operands is greater than $2\hat{2}56$, there may be an overflow error
SUB	Under the constraints of entering this node, if the minuend number is greater than the subtracted one, there may be an underflow error
DIV	Under the constraints of entering this node, if the divisor can be equal to 0, there may be an underflow error

When encountering with these four instructions, *add*/*sub*/*mul*/*div*, enter the integer overflow detection module to perform overflow detection. Perform different functions for different operations, and if the solution result exists, add the vulnerability information to the issue.

Unchecked Call Return Value Vulnerability Detection. The correct way to write the *call* or *send* function should be:

```
require(msg.sender.send(value))
```

Due to the existence of require statement, if the return value of *send* is false, it will directly revert and exit. When calling *CALL/CALLCODE/DELEGATECALL/STATICCALL* or other commands, the return value of the command is recorded and stored in a variable of z3 format. If this block is normal ended by *RETURN* or *STOP*, take out the return value of the previous *CALL/CALLCODE/DELEGATECALL/STATICCALL* instruction. If the value can be equal to 0, it means that the *send/call* operation can be failed. In addition to other operations, there may be a vulnerability that does not check the return value of *call* at this time.

4.4 Constraint Solving

The z3 solver supports all theoretical solution classifications, so we use the z3 solver here to solve the constraints of the path state. According to the parameter types in EVM, use z3 solver to realize two types of variables, they are: *Bool* and *BitVec*. *Bool* represents the Boolean type, *BitVec* is a bit array type. Generally the length is 256 bits, because EVM does not have a series of data types such as int8/int16/int24/.../int256 like *Solidity*. When this data type is converted to bytecode, it will be stored as a 256bit variable. Aiming at these two data types and using the z3 solving library function, the realization of the data operation rules is redesigned.

5 Evalution

We did a vulnerability detection experiment on 1552 contracts from awesome-buggy-erc20-tokens [9] data set, which shows 1320 contracts are detected as vulnerabilities, and the accuracy rate of the vulnerability detection is 85.1%. Figure 8 shows the vulnerability detection results.

We manually checked twenty contracts which are reported as vulnerable by our system and we think all of them have vulnerabilities. For example, the contract in 0x0b76544F6C413a555F309Bf76260d1E02377c02A[1] has no-Approval, owner-control-sell-price-for-overflow, owner-decrease-balance-by-mint-by- overflow, totalsupply-overflow and transfer-no-return issues defined by awesome-buggy-erc20-tokens. Our tool has detected an integer overflow vulnerability in this contract. Check the contract code snippets in Fig. 9, and we found line 5 does have an integer overflow problem.

[1] https://etherscan.io/address/0x0b76544F6C413a555F309Bf76260d1E02377c02A.

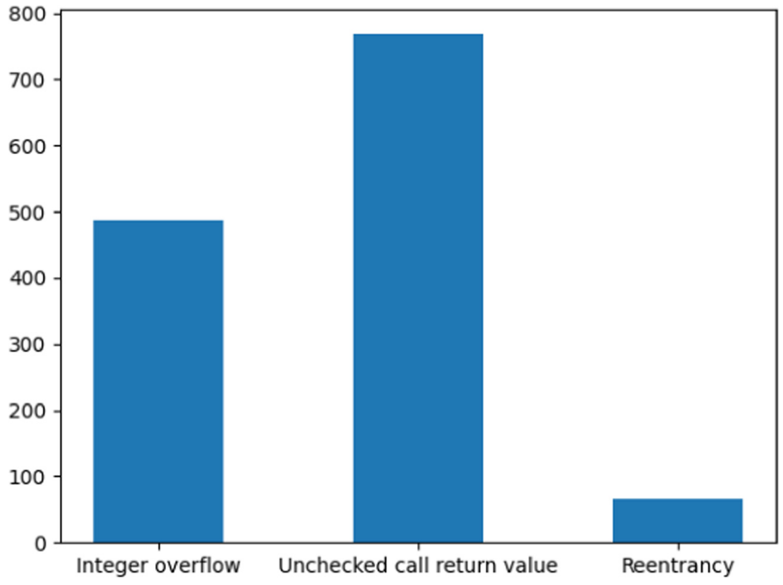

Fig. 8. Vulnerability detection results of awesome-buggy-erc20-tokens.

```
contract INTToken is owned, token
{
    //....
    function sell(uint256 amount) {
      require(this.balance >= amount * sellPrice);       // checks if the contract has enough ether to buy
      _transfer(msg.sender, this, amount);               // makes the transfers
      msg.sender.transfer(amount * sellPrice);           // sends ether to the seller. It's important to do this last to avoid recursion attacks
    }
    //...
}
```

Fig. 9. Code snippets of 0x0b76544F6C413a555F309Bf76260d1E02377c02A

6 Conclusion

We investigate the most common contract security issues and the most widely used smart contract vulnerability detection methods currently. Then we design and implement a smart contract detection system. The main functions covered by the system are as follows:

(1) **Disassembly of EVM bytecode.** Disassemble the EVM bytecode to generate an opcode sequence for specific instruction analysis in the next step.

(2) **Generation control flow graph.** Generate control flow graph of the contract to provide a basis for symbolic execution.
(3) **symbolic execution implementation.** Implementation a simple Ethereum virtual machine according to the Ethereum EVM instructions and the EVM design principle. Perform path depth-first traversal on the control flow graph, store each data as a variable in the form of z3, and use symbolic variables to represent path constraints.
(4) **Vulnerability detection.** Detect integer overflow vulnerabilities, reentrance vulnerabilities, and unchecked call return value vulnerabilities of smart contracts.

We tested the vulnerability detection system on awesome-buggy-erc20-tokens data set and analyzed the experimental results which shows our system has a good performance on the vulnerability detection accuracy.

Acknowledgments. This work is supported by CNKLSTISS and the National Natural Science Foundation of China (Grant No. 61802025).

References

1. https://ethereum.github.io/yellowpaper/paper.pdf
2. Luu, L., Chu, D.H., Olickel, H., et al.: Making smart contracts smarter. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269 (2016). https://doi.org/10.1145/2976749.2978309
3. Torres, C.F., Schütte, J., State, R.: Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676 (2018). https://doi.org/10.1145/3274694.3274737
4. Grieco, G., Song, W., Cygan, A., et al.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 557–560 (2018). https://doi.org/10.1145/3395363.3404366
5. Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 259–269. IEEE (2018). https://doi.org/10.1145/3238147.3238177
6. Kalra, S., Goel, S., Dhawan, M., et al.: Zeus: analyzing safety of smart contracts. In: NDSS, pp. 1–12 (2018). https://doi.org/10.14722/ndss.2018.23082
7. Tsankov, P., Dan, A., Drachsler-Cohen, D., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018). https://doi.org/10.1145/3243734.3243780
8. Krupp, J., Rossow, C.: teether: Gnawing at Ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 1317–1333 (2018)
9. https://github.com/search?q=awesome-buggy-erc20-tokens