ReJection: A AST-Based Reentrancy Vulnerability Detection Method

Rui Ma[1], Zefeng Jian[1], Guangyuan Chen[1], Ke Ma[2](✉), and Yujia Chen[1]

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

[2] Internet Center, Institute of Technology and Standards Research, China Academy of Information and Communication Technology, Beijing 100191, China

make@caict.ac.cn

Abstract. Blockchain is deeply integrated into the vertical industry, and gradually forms an application ecosphere of blockchain in various industries. However, the security incidents of blockchain occur frequently, and especially smart contracts have become the badly-disastered area. So avoiding security incidents caused by smart contracts has become an essential topic for blockchain developing. Up to now, there is not generic method for the security auditing of smart contracts and most researchers have to use existing vulnerability detection technology. To reduce the high false rate of smart contract vulnerability detection, we use ReJection, a detection method based on abstract syntax tree (AST), to focus on the reentrancy vulnerability with obvious harm and features in smart contracts. ReJection consists of four steps. Firstly, ReJection obtains the AST corresponding to the contract by the smart contract compiler *solc*. Then, AST is preprocessed to eliminate redundant information. Thirdly, ReJection traverses the nodes of the AST and records the notations related to reentrancy vulnerabilities during the traversal, such as Danger-Transfer function, Checks-Effects-Interactions pattern and mutex mechanism. Finally, ReJection uses record information and predefined rules to determine whether the reentrancy vulnerability is occurred. ReJection is implemented based on Slither, which is an open-source smart contract vulnerability detection tool. Furthermore, we also use the open-source smart contract code as the test program to compare experimental results to verify the effects with the ReJection and Slither. The result highlights that the ReJection has higher detection accuracy for reentrancy vulnerability.

Keywords: Vulnerability detection · Smart contract · Abstract syntax tree · Reentrancy vulnerability

1 Introduction

Blockchain as an independent technology originates from Bitcoin designed by Satoshi Nakamoto [1]. In his paper ⟨Bitcoin: a peer-to-peer electronic cash system⟩ [2], Nakamoto describes an electronic digital currency system that does not rely on trusted third parties. The underlying technology that supports the system is called blockchain.

Blockchain 1.0 is a virtual digital currency represented by Bitcoin and Litecoin. Blockchain 2.0 refers to smart contracts represented by Ethereum [3]. Smart contracts have brought a qualitative leap to the blockchain, but security issues caused by smart contracts have also drawn increasing attention.

Smart contract represents a trusted blockchain platform where developers can't stop any smart contract or modify the content of smart contract after deploying a smart contract. While bringing a credible advantage to smart contracts, it also brings a considerable degree greatly of security risks. Essentially, each smart contract is a program that has the possibility for error. If there are vulnerabilities in the smart contract, the user's digital currency funds may be taken away by the attacker. And that even leads to serious consequences. Due to the inability to stop and modify the smart contracts, however, it will be difficult to repair it by upgrading when there is a potential vulnerability in the smart contracts. Therefore, the security audit of smart contracts is particularly necessary.

To address the issue, we propose the ReJection. It is a vulnerability detection method based on abstract syntax tree for smart contract reentrancy vulnerability. That vulnerability has obvious features and larger hazard. ReJection analyzes smart contracts source code by static analysis techniques. Generally, ReJection is able to detect the reentrancy vulnerability generated by the dangerous transfer function by analyzing the compile results of the smart contracts source code. At the same time, ReJection has better detection and analysis effects for the Checks-Effects-Interactions pattern and mutex mechanism, which is the key to reentrancy vulnerability prevention.

We make the following contributions:

- We study the causes of the smart contract reentrancy vulnerability, and analyze the existing security audit method of smart contract.
- A detection method, ReJection, is proposed for the smart contract reentrancy vulnerability. By compiling and parsing the source code of smart contracts, ReJection could determine whether there is reentrancy vulnerability. At the same time, ReJection also provides better detecting for Checks-Effects-Interactions pattern and mutex mechanism, which are the key of reentrancy vulnerability.
- We implemented ReJection based on the existing open source vulnerability detection tool Slither, and experimented with ReJection through open source smart contract source code as experimental dataset. By comparing ReJection with Slither, the results show that ReJection improves the accuracy of reentrancy vulnerability detection.

2 Related Works

2.1 Blockchain and Smart Contracts

In a narrow sense, the blockchain is a chained data structure that combines data blocks in a chronological order in a sequential manner, and it cryptographically guarantees non-tamperable and unforgeable distributed ledgers. At the same time, the blockchain is also an innovation application of traditional computer technologies such as distributed data storage, consensus algorithms, P2P transmission, and various encryption algorithms in the new Internet era. At present, blockchain technology has been budded off from Bitcoin and has developed in many fields including financial trade, logistics, smart contracts, and sharing economy.

Blockchain 1.0 is a virtual digital currency represented by Bitcoin and Litecoin including its functions of payment, circulation and other similar currencies. It represents a digital currency-based application for decentralized digital currency transactions.

Blockchain 2.0 refers to the smart contracts represented by Ethereum [4]. The combination of smart contracts and digital currency provides extensive application scenario for the financial field and forms “Programmable Finance”. Smart contract is the core of Blockchain 2.0 and has Turing Completeness. It is an event-driven computer program that runs on a replicable shared blockchain distributed ledgers. It enables autonomous invoking data processing, accepting, storing and forwarding the value that corresponding to digital currency, as well as controlling and managing various types of intelligent digital assets on the blockchain. Smart contract is identified by a 160-bit identifier address whose code is compiled and deployed on the blockchain. All users can send a transaction to the address of the contract account through an external private account to sign a smart contract in the cryptocurrency. Therefore, smart contract makes the blockchain programmable and customizable, which gives the blockchain intelligence and indicates a development direction of the blockchain technology in the future.

2.2 Smart Contract Security Audit

Research on the smart contract security audits has just beginning.

Atzei analyzed the security vulnerabilities of Ethereum smart contracts [5], demonstrated a series of attacks that exploited these vulnerabilities, and provided a summary of the common programming pitfalls that could lead to vulnerabilities in Ethereum smart contracts.

Delmolino summarized the common mistakes made in coding smart contracts and the common pitfalls that are exposed when designing secure and reliable smart contracts, and proposed some suggestions for coding safety smart contracts [6].

Based on techniques such as symbolic execution, SMT solving and taint analysis, Bernhard Mueller proposed Mythril [7], a security analysis tool for

Ethereum smart contracts, to detect a variety of security vulnerabilities. He also analyzed the application of symbolic execution and constraint solving in smart contract security analysis.

Loi Luu analyzed several security vulnerabilities in smart contracts and found that about 44% of smart contracts have security risks. At the same time, a symbolic execution-based vulnerability detection tool Oyente [8] was proposed. It uses the decompiled code of the smart contract to construct a control flow graph based on basic blocks and further obtain the constraint path of the vulnerability by Z3 solver.

Bhargavan proposed a formal verification method for verifying smart contracts written by Solidity [9], and outlined a framework. The framework analyzes and verifies the safety of operational and accuracy of functional of Ethereum smart contracts by converting smart contracts to F*, which is a functional programming language for program verification.

To address the security problem of smart contracts, Petar Tsankov introduced Securify [10], a security analysis tool for Ethereum smart contracts. Securify extracts precise semantic information from the code by analyzing the function dependency graph of the contract, and then checks compliance and violation modes to capture sufficient conditions for verifying the vulnerability.

Grigory Repka developed the online smart contract static analysis tool SmartCheck to detect various security vulnerabilities in smart contracts [11], but he did not specify which detection techniques were used.

Hukai proposed a formal verification method for smart contracts [12], which can be used in the process of modeling, model checking and model validation.

Xin Wei analyzed the threat of smart contracts and the principle of the exiting vulnerabilities, summarized some common problems faced by smart contract vulnerability detection, and proposed an automatic vulnerability detection theory of smart contract [13].

Chengdu LianAn Technology [14] proposes an automatic formal verification platform VaaS for smart contract security issues.

2.3 Reentrancy Vulnerability

The occurrence of a smart contract reentrancy vulnerability means that the contract executes a callback operation. For smart contracts, function invocations can be made between the contract account and the external account to achieve more functionality of the smart contracts. Specifically, invoking to the fallback function is a kind of callback operation.

For each smart contract, there is at most one function without a function name. It does not need to be declared, and has no parameters and no return value. In addition, it needs to be visible to the outside. Such functions are called fallback functions. Once someone makes a transfer transaction to a contract account, the fallback function corresponding to the contract account is invoked.

Literally, reentrancy vulnerabilities are caused by repeated entry. In the Ethereum smart contracts, the attacker constructs malicious code in the fallback function of the contract address. Once the transfer function is executed to the vulnerable contract account, the contract account is forced to execute the fallback function of the attack contract due to its own vulnerability defect. That will trigger execution of the malicious code built by the attacker within the fallback function. The malicious code includes recalling the contract transfer function, which can result in the operation of reentering the contract to execute some operations like transfer Ether. Ultimately, it will lead to the theft of assets.

3 ReJection

3.1 Overview

After carefully analyzing the existing smart contract vulnerability detection methods and the reentrancy vulnerability characteristics, we propose the ReJection. ReJection is a detection method for smart contract reentrancy vulnerability based on abstract syntax tree (AST), which could improve the detection efficiency of the smart contract reentrancy vulnerability and the detection accuracy of the reentrancy vulnerability prevention condition.

ReJection detects vulnerability by traversing and parsing the source code of smart contract. It uses the open source smart contract compiler *solc* as the tool of syntax analysis to obtain the AST of the source code. By excluding the redundant information of the AST, ReJection could extract the key information about the vulnerability detection of the source code. That key information will be saved in the reserved nodes of AST. Then, the step-by-step nested traversal analysis is performed on the reserved nodes to detect the occurrence conditions of the reentrancy vulnerability. Specifically, that condition refers to whether the original contract account balance changes after executing the Danger-Transfer function. At the same time, the prevention conditions for reentrancy vulnerability are detected and analyzed. Generally, ReJection detects whether there is a change in the contract account balance and whether there exists the mutex mechanism after the execution of the Danger-Transfer function, respectively. Finally, the results of the analysis of the occurrence conditions and the prevention conditions are analyzed comprehensively to determine whether the contract is in danger of reentrancy vulnerability.

Figure 1 shows the detection scheme of ReJection, which can be divided into four parts.

(1) Obtaining AST. ReJection compiles the source code of smart contract by the *solc* compiler to generate an intuitive AST in *json* format and further outputs that to a local text file.
(2) Preprocessing the Redundant Node of AST. ReJection analyzes and preprocesses the AST to exclude redundant nodes and obtain the reserved nodes that reentrancy vulnerability may exist.
(3) Traversing the Reserved Nodes of AST. ReJection performs a step-by-step traversal analysis of the reserved nodes based on the attributes of the various nodes in the reserved nodes. For each contract, it detects whether there is

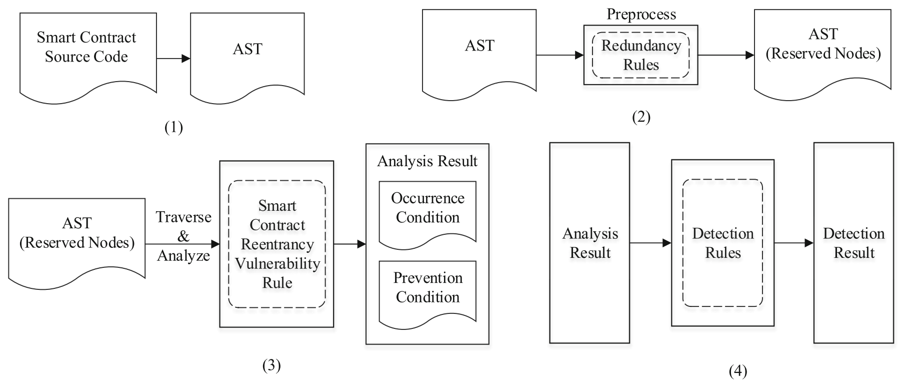

Fig. 1. Detection scheme of ReJection

a change in the contract account balance after the execution of the Danger-Transfer function, and whether there is a mutex mechanism in the contract. At the same time of traversing, ReJection also determines and assigns the parameters related to the reentrancy vulnerability so as to obtain the detection result of the Danger-Transfer function and the record list of the prevention conditions.

(4) Determining the Reentrancy Vulnerability. According to the analysis result of Step (3) and the reentrancy vulnerability determination rule summarized by the author, ReJection detects whether there is a reentrancy vulnerability generation condition. Finally, it could obtain the detection result of the reentrancy vulnerability.

3.2 Obtaining AST

AST is an abstract representation of the grammar structure of the source code. It intuitively expresses the structure in the programming language by the form of the tree. A node of AST represents one or more structures in the source code, while the AST contains a complete representation of the entire compilation unit. The AST in the Ethereum smart contract is obtained by the *solc* which is an open source compiler of the smart contract. In order to obtain intuitive representation of the AST, ReJection uses the *json* format to record the details of the AST.

Figure 2 shows a part of information of the AST of identifier node, where the *name* attribute represents the name of variable and the *typeDescriptions* attribute represents the details of return type. AST is represented with the form of the nested structure and each type of node has some special properties.

3.3 Preprocessing the Redundant Node of AST

In the AST obtained in the previous step, there is a considerable part of information that is not related to the reentrancy vulnerability detection, such as

```
{
        "argumentTypes" : null,
        "id" : 18,
        "name" : "balances",
        "nodeType" : "Identifier",
        "overloadedDeclarations" : [],
        "referencedDeclaration" : 15,
        "src" : "292:8:0",
        "typeDescriptions" :
        {
              "typeIdentifier" : "t_mapping$_t_address_$_t_unit256_$",
              "typeString" : "mapping (address => unit256)"
        }
}
```

Fig. 2. An Example of the AST of the identifier node

version information, compilation information. In order to improve the efficiency of the detection, it is necessary to preprocess the AST to exclude redundant information. The preprocessing divides all nodes into the reserved nodes and the redundant nodes. The information contained in the redundant nodes is independent of ReJection, while the information contained in the reserved nodes is closely related to the ReJection.

The preprocessing is described as follows:

(1) The root node, which is taken as a reserved node, represents all the information of the smart contract. ReJection further traverses the child nodes of the root node for analysis.
(2) ReJection traverses the child nodes of the root node to detect the node type *nodetype* of each child node. The values of the *nodetype* include *PragmaDirective*, *ContractDefinition*, and *ImportDirective*. The *PragmaDirective* represents the compiled version of the contract, the *ContractDefinition* indicates that the node is a library or contract, and the *ImportDirective* represents the other source files imported by the contract. The node of AST whose *nodetype* is *ContractDefinition* is identified as the reserved node and the other nodes are identified as the redundant nodes.
(3) ReJection further traverses the child nodes of the reserved node to detect its attribute *contractKind*. The node whose value of *contractKind* is *contract* is identified as the reserved node, and other child nodes are identified as the redundant nodes.

After the traversal, the remaining nodes in the AST are reserved nodes. The information of the nodes corresponds to the contract information.

3.4 Traversing the Reserved Nodes of AST

Traversing the reserved node is the most critical step in ReJection. The reserved nodes, which are obtained by the preprocessing of the redundant node, include all state variables and functions of the contract. The different types of the reserved nodes can be distinguished by the node type and return type. Aiming at different type of the reserved nodes, ReJection specifically analyzes it and assigns the value for the key parameters related to detecting reentrancy vulnerabilities. After the traversal of the reserved nodes has been fully completed, ReJection uses the above parameters to determine whether there is the reentrancy vulnerability in the contract according to the reentrancy vulnerability detection rule.

The key parameters associated with detecting reentrancy vulnerability are described in Table 1.

The method of traversing the reserved nodes is shown in Fig. 3. To conveniently explain the process of parsing AST, we give the following definition and abbreviations:

tCN: Current Node

tULN: Upper Level Node of Current Node

tNLN: Next Level Node of Current Node

Table 1. The definition of parameters

Variable name	Type	Initial value	The meaning of initial value
reentrancyCode	string	NULL	Save the Danger-Transfer function
isReentrancy	int	0	No Danger-Transfer function
variableMutex	array	NULL	Record a set of Mutex Mechanism variable
isMutex	int	0	No Mutex Mechanism
isChange	int	0	Exist the Checks-Effects-Interactions pattern
ifList	array	NULL	Used for Determination of Mutex Mechanism

Step 1. Traverse the reserved nodes and detect the node type based on the attribute *nodeType* of the *tCN*. If the *nodeType* is *VariableDeclaration*, the *tCN* represents a state variable and the return type *typeString* of *tCN* should be detected; while the information of the node whose *typeString* is bool or integer is recorded as state variable for further detection. If the *nodeType* is *FunctionDefinition*, the *tCN* represents a function and the *tNLN*, which is the next level node of the *tCN*, should be traversed by Step 2.

Step 2. Detect the *nodeType* of the node. If the *nodetype* is *ExpressionStatement*, the *tCN* represents an expression statement and the *expression* node of *tNLN* should be analyzed by Step 3. If the *nodetype* is *IfStatement*, *WhileStatement*, *DoWhileStatement*, or *ForStatement*, the *tCN* represents a statement of loop or judgment. Then, the *condition* node of *tNLN* should be processed as *expression* node and *body* node of *tNLN* should be continued analyzing by Step 2. If the *nodetype* is *Break*, *Continue*, or *Throw*, the *tCN* represents jumping out of the loop or judgement statement. So, it is necessary to return the node corresponding to the previous loop or judgement statement and continue performing Step 2.

Step 3. Detect *expression* node. This step is mainly for analyzing different *nodeType*.

FunctionCall: If the *nodeType* of the *tCN* is *FunctionCall*, the *name* attribute of the *expression* node of the *tNLN* is detected. If there is no such *name* attribute or the value of the *name* attribute is NULL, that *expression* node should be continued processing according to Step 3. If the value of the *name* attribute is *require* or *assert*, the *arguments* node of the *tNLN* should be continued processing by Step 3. It is noted that if that *arguments* node has return value, that value should be recorded into the *ifList* to determine the mutex.

MemberAccess: If the *nodeType* of the *tCN* is *MemberAccess*, the return type *typeString* of the *tCN* is detected. If the *typeString* contains the "*payable*" field, the value of *memberName* attribute of the *tCN* should be recorded, and that value should be appended to the original value of parameter *reentrancyCode* to concatenate with the *reentrancyCode*. After that, the *expression* node of the *tNLN* is continued detecting by Step 3.

Identifier: If the *nodeType* of the *tCN* is *Identifier*, and if the value of the *name* attribute of the *tCN* belongs to the state variable recorded in Step 1, and the value of *isReentrancy* is 0 at this time, the value of *name* attribute of the *tCN* is returned. Otherwise, if the value of the *name* attribute of the *tCN* does not belong to the recorded state variable, or the value of *typeString* attribute of the *tCN* is *address*, the value of *name* attribute of the *tCN* is appended to the value of parameter *reentrancyCode*. While if the *reentrancyCode* is begins with "*valuecall*", the value of parameter *isReentrancy* is assigned 1 and the value of *reentrancyCode* is assigned NULL. Otherwise, the *isReentrancy* and the *reentrancyCode* are unchanged.

Assignment: If the *nodeType* of the *tCN* is *Assignment*, it means that the node contains the assignment operator. If the value of *isReentrancy* is 1, the *leftHandSide* node of the *tNLN* should be continued detecting by Step 3; it is noted that if the *leftHandSide* node returns *true*, the value of parameter *isChange* should be assigned 0 at this time. If the value of *isReentrancy* is 0, the *leftHandSide* node of the *tNLN* should be continued detecting by Step 3. Meanwhile, if the return value of the *leftHandSide* node belongs to the state variable recorded in Step 1, the state variable should be assigned with the value of *value* attribute of the *rightHandSide* node of the *tNLN*.

Literal: If the *nodeType* of the *tCN* is *Literal*, it returns the value corresponding to the *value* attribute of the *tCN*.

BinaryOperation: If the *nodeType* of the *tCN* is *BinaryOperation*, the node represents a binary operation. If the value of *isReentrancy* is 0, both the *leftExpression* node and the *rightExpression* node of the *tNLN* should be continued detecting by Step 3. Meanwhile, if the return values of those two nodes belong to the state variable recorded in Step 1, the return values and the operator of the *tCN* are recorded in the *ifList* to further determine the mutex. And the result of current binary operation should also be returned.

Step 4. For each state variable recorded in Step 1, all the conditions recorded by the parameter *ifList* in Step 3 are used to judge one by one. If there is a state variable makes one of the conditions of *ifList* is unsatisfiable, the value of parameter *isMutex* is assigned 1.

3.5 Determining the Reentrancy Vulnerability

By analyzing, the execution of the Danger-Transfer function ⟨address⟩call. value()() may lead to the reentrancy vulnerability. Therefore, we summarize prevention ways including the following three types:

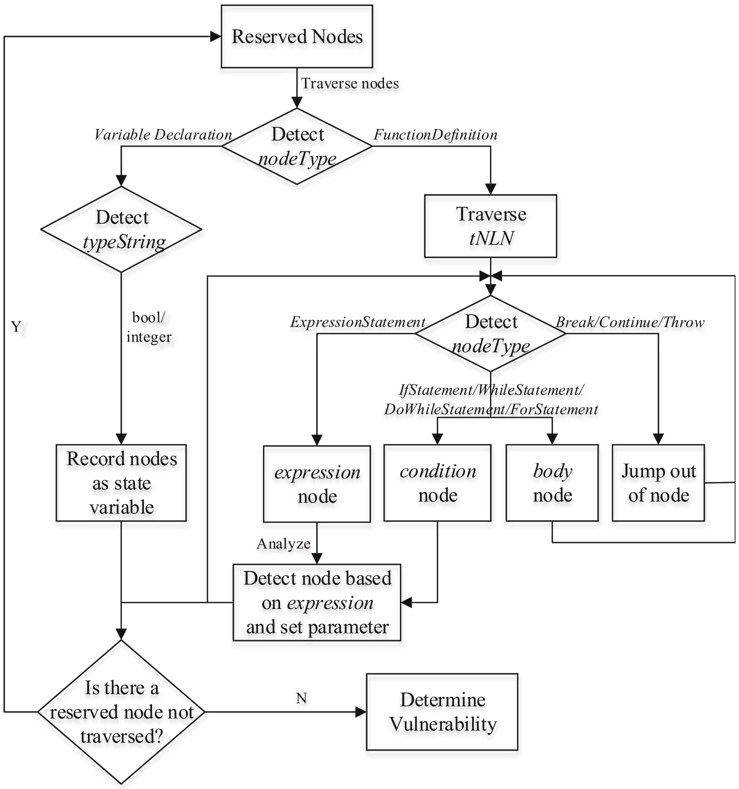

Fig. 3. Traversing the reserved nodes

(1) Use the secure transfer function, such as transfer(), instead of existing Danger-Transfer function;
(2) Use the Checks-Effects-Interactions pattern. Specifically, it completes all operations of the contract account before the transfer function is executed;
(3) Introduce mutex mechanism. Specifically, it avoids the occurrence of reentrancy vulnerability by using a flag of global variable.

According to the points (2) and (3) of that prevention ways, and combining with the analysis results obtained in Step 3 of Sect. 3.4, ReJection regulates 4 rules to determine reentrancy vulnerability.

Rule 1. If *isReentrancy* $= 0$, which indicates that the Danger-Transfer function does not appear, there is no reentrancy vulnerability.

Rule 2. If *isReentrancy* $= 1$ and *isMutex* $= 1$, which indicates that there appears the Danger-Transfer function, but mutex mechanism exists, there is no reentrancy vulnerability.

Rule 3. If *isReentrancy* $= 1$, *isMutex* $= 0$, and *isChange* $= 0$, there is no reentrancy vulnerability. This situation indicates that there appears the Danger-Transfer function and has not mutex mechanism, but it has the Checks-Effects-Interactions pattern which means it is no change in the contract account balance after the transfer function.

Rule 4. If and only if *isReentrancy* $= 1$, *isMutex* $= 0$, and *isChange* $= 1$, there must be a reentrancy vulnerability. This situation indicates that there appears the Danger-Transfer function and has not mutex mechanism, but it also has not the Checks-Effects-Interactions pattern.

4 Evaluation

We implement ReJection based on the open source smart contract vulnerability detection tool Slither. In order to verify detection accuracy of ReJection for reentrancy vulnerability, a comparative experiment between Slither and ReJection has been further completed. Specifically, it selects open source smart contract data set as the experimental source code; uses the smart contract compiler *solc* to generate AST as the input of ReJection and uses the smart contract source code as the input of the Slither; and records the vulnerability detection results of the Slither and ReJection to compare and analyze.

Table 2 shows the experimental results. The experiment compares 8 different contracts, in which the contract details are represented by the Danger-Transfer function (D-T), the mutex mechanism (Mutex) and the Checks-Effects-Interactions pattern (C-E-I). Here the Danger-Transfer function indicates that transfer function ⟨address⟩.call.value()(), and the mutex mechanism and the Checks-Effects-Interactions pattern is introduced in Sect. 3.5. For the value of corresponding cell, "Y" indicates that there exists Danger-Transfer function, the mutex mechanism or Danger-Transfer function in the contract, whereas "N" indicates that none of them exists. For the expected result, the detection result of the Slither and ReJection, "Y" indicates that the reentrancy vulnerability exists, and "N" indicates that there is no reentrancy vulnerability.

In Table 2, contracts *SimpleDAO*, *Reentrancy* and *BrokenToken* have the same contract details. They have Danger-Transfer functions and have no mutex mechanism and the Checks-Effects-Interactions pattern. According to the Rule 4 in Sect. 3.5, there must be existed a reentrancy vulnerability in the above contracts. The results of Slither and ReJection are consistent with the expected results, which shows the reentrancy vulnerability can be correctly detected by them.

Table 2. Comparison of experimental results

Contract	Contract details			Expected result	Detection result	
	D-T	Mutex	C-E-I		Slither	ReJection
SimpleDAO	Y	N	N	Y	Y	Y
Reentrancy	Y	N	N	Y	Y	Y
BrokenToken	Y	N	N	Y	Y	Y
ModifierEntrancy	N	N	N	N	N	N
SimpleDAOFixed	Y	N	Y	N	N	N
noCheck	Y	N	Y	N	Y	N
EtherStore	Y	Y	N	N	Y	N
EtherBankStore	Y	Y	Y	N	Y	N

Different from above contracts, contract *ModifierEntrancy* has no Danger-Transfer function. According to the Rule 1 in Sect. 3.5, there definitely has no reentrancy vulnerability in the contract. The results of Slither and ReJection are consistent with the expected results.

For contract *SimpleDAOFixed*, it has the Danger-Transfer function and the Checks-Effects-Interactions pattern, but there is no mutex mechanism. According to the Rule 3 in Sect. 3.5, there definitely has no reentrancy vulnerability in the contract. The results of Slither and ReJection are consistent with the expected results.

The contract details of contract *noCheck* is exactly the same as the contract *SimpleDAOFixed*, it should be no reentrancy vulnerability. However, in the real detection, ReJection got the correct result, whereas Slither had a false positive. Slither mistakenly believed that there was a reentrancy vulnerability in the contract.

Moreover, for contract *noCheck* and contract *EtherStore*, although they have different contract details, both of them have the Danger-Transfer function and have either a mutex mechanism or a Checks-Effects-Interactions pattern. According to the Rule 2 and Rule 3 in Sect. 3.5, there definitely has no reentrancy vulnerability in these two contracts. Unfortunately, in the real detection, Slither made a false result because it can not distinguish the difference between the two prevention ways. Instead, ReJection can detect these two kinds of prevention ways and report the correct detection result.

Similarity, for contract *EtherStore* and contract *EtherBankStore*, even although the contract details of them are not the same, both of them contain the Danger-Transfer function and a mutux mechanism. According to the Rule 2 of Sect. 3.5, there is no reentrancy vulnerability in them. While in the real detection, ReJection still got the correct result, and Slither had a false positive yet.

Although the examples of smart contracts involved in Table 2 are few, it is noted that they still cover all four types of determination rules proposed in

Sect. 3.5. For the above contracts, detection accuracy of Slither is 62.5%, while ReJection ones achieves 100%. ReJection is better than Slither in the detection effect. The main reason may be that ReJection fortifies with prevention conditions when detecting reentrancy vulnerability. Later, it is necessary to expand the number of testing contracts to further verify the detection capability of ReJection.

5 Conclusion

In this paper, we propose the ReJection, a method of smart contract reentrancy vulnerability detection based on abstract syntax tree (AST). By resolving the AST, ReJection uses the proposed redundancy rules to eliminate the nodes that are useless for vulnerability detection. That indirectly improves the efficiency of vulnerability detection. By traversing the reserved nodes in the AST, ReJection analyzes and records key information related to occurrence conditions and prevention conditions of the reentrancy vulnerability. Then, ReJection further detects whether there exists a reentrancy vulnerability generation condition depending on the reentrancy vulnerability determination rules summarized by the author. That improves the accuracy of the reentrancy vulnerability detection. Moreover, ReJection is implemented based on the Slither, which is an existing open source vulnerability detection tool of smart contract. The effectiveness of ReJection has been verified by comparing experimental results between ReJection and Slither.

References

1. Bitcoin Sourcecode. https://github.com/bitcoin/bitcoin/. Accessed 18 Jan 2016
2. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. Accessed 2018
3. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. White paper, pp. 1–36 (2014)
4. Parizi, R.M., Dehghantanha, A.: Smart contract programming languages on blockchains: an empirical evaluation of usability and security. In: Chen, S.P., Wang, H., Zhang, L.J. (eds.) ICBC 2018. LNCS, vol. 10974, pp. 75–91. Springer, Cham (2018). https://doi.org/10.10007/978-3-319-94478-4_6
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
6. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 79–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_6
7. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. In: The 9th Annual HITB Security Conference (2018)

8. Luu, L., Chu, D.H., Olickel, H., et al.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security 2016, pp. 254–269. ACM, New York (2016). https://doi.org/10.1145/2976749.2978309
9. Bhargavan, K., Swamy, N., Zanella-Bguelin, S., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security 2016, pp. 91–96. ACM, New York (2016). https://doi.org/10.1145/2993600.2993611
10. Tsankov, P., Dan, A., Drachsler-Cohen, D., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security 2018, pp. 67–82. ACM (2018). https://doi.org/10.1145/3243734.3243780
11. SmartCheck. https://tool.smartdec.net. Accessed 22 Oct 2017
12. Hu, K., Ai, X.M., Gao, L.C., et al.: Formal verification method of smart contract. J. Inf. Secur. Res. **2**(12), 1080–1089 (2016)
13. Xin, W., Zhang, T., Zou, Q.C.: Research on vulnerability of blockchain based smart contract. In: The 10th Conference on Vulnerability Analysis and Risk Assessment 2017, pp. 421–437 (2017)
14. Chengdu LianAn Technology. http://www.lianantech.com. Accessed 10 June 2019