M-A-R: a Dynamic Symbol Execution Detection Method for Smart Contract Reentry Vulnerability

Zexu Wang[1], Bin Wen[1,2,3(✉)], Ziqiang Luo[2,3], and Shaojie Liu[1]

[1] School of Information Science and Technology, Hainan Normal University, Haikou 571158, China
binwen@hainnu.edu.cn
[2] Cloud Computing and Big Data Research Center, Hainan Normal University, Haikou 571158, China
[3] Key Laboratory of Data Science and Intelligence Education of Ministry of Education, Hainan Normal University, Haikou 571158, China

Abstract. The original intention of smart contract design is to execute every transaction in the blockchain spontaneously, efficiently and fairly, meanwhile, smart contract plays an important role in the blockchain activities. With the development of blockchain, the vulnerability of smart contract becomes more and more obvious. The security vulnerability detection of smart contract is very important. This paper proposes M-A-R, a dynamic symbol execution method focusing on efficient detection for reentry vulnerability, realizes the security detection of the source code of smart contract, optimizes the design of its implementation method, then compares it with the existing related tools. The results show that M-A-R approach can detect the reentry vulnerability efficiently and has good universality and scalability.

Keywords: Smart contract · Dynamic symbolic execution · Reentry attack · Vulnerability detection · Blockchain

1 Introduction

Whether in the public blockchain or consortium blockchain, smart contract plays a very important role in the correct execution of transactions in the blockchain. As the blockchain, once a transaction is chained, it cannot be tampered with, that is: code is law. The design concept of smart contract is another form of human will, so the security problem of smart contract is inevitable, the incorrect use of syntax rules by programmers, the characteristics of blockchain, and the security of code change with the development of blockchain. In short, smart contracts with defects will make assets and transactions vulnerable and vulnerable, seriously affecting the normal operation of blockchain transactions. At present, the number of tools for smart contract vulnerability detection is small, and few tools can achieve high accuracy and efficient detection. The existing smart contract vulnerability detection technologies mainly include symbol execution,

H.-N. Dai et al. (Eds.): BlockSys 2021, CCIS 1490, pp. 418–429, 2021.
https://doi.org/10.1007/978-981-16-7993-3_32

fuzzy testing, formal verification, program analysis and taint analysis. The main idea of symbolic execution is to convert the uncertain input values into symbolic values to promote the analysis of program execution. The method of dynamic symbol execution can maximize the code coverage. In this paper, we will use the improved method of dynamic symbol execution to detect reentry attacks, simplify the operation process of dynamic symbol execution, and maximize the code coverage in a short time.

The main contributions of this paper are as follows:

Novel Idea: we introduce M-A-R, which tracks transactions by simulating transactions to detect whether reentry attacks occur. At the same time, we simplify the process of solving constraints to improve the speed of operation and ensure the reliability and applicability of detection;

Comprehensive Evaluation: the effectiveness of M-A-R in detecting the vulnerability of reentry attack is evaluated by experiments with existing detection tools with good performance;

High Reliability: the new idea of dynamic symbol execution detection in this method can greatly simplify the process of solving dynamic symbol execution constraints and make the detection process more efficient and reliable;

Open Source: in order to facilitate the follow-up research, upload the experimental source code and experimental cases to GitHub.[1] .

The organization of the remaining sections is as follows. In Sect. 2, we will introduce some background knowledge, attack mechanism and the advantages and disadvantages of existing detection tools.In Sect. 3, we review reentrancy attack of Ethereum smart contracts.In Sect. 4,we will talk about Main research methods and deficiencies for reentry attack detection. Then in Sect. 5, we will introduce the detection principle and implementation method of dynamic symbol execution. In Sect. 6, we will introduce the experiment and evaluation. Finally, In Sect. 7, we will summarize the experimental results and propose future works.

2 Reentry Attack

Smart contract is a piece of code running on the blockchain platform. It has three cache units for storage: stack, memory and storage. Stack operations will disappear with the end of the transaction, and will not be stored. Memory is the address area of bytes allocated at run time. Both of them are easy to lose. Therefore, the balance of the account information that needs to be saved after the transaction is saved by the storage unit. At the same time, the gas price higher than stack and memory calculation is needed to calculate the storage.

When a smart contract performs a program call, the stack is needed to record the return address at the end of the call. It can be used directly when calling the function inside the contract. When calling the function of other contracts outside the contract, it is called externally. Call instruction can send information to the outside, but there is no gas consumption limit when calling the call

[1] https://github.com/woods1060/M-A-R.

instruction of the external function. By default, all the remaining gas will be used to execute this command. Using the call instruction to call and transfer the external contract, the risk of reentry attack will be greatly increased.

Reentry attack is the most typical vulnerability in high level language of smart contract vulnerability, and it is also one of the most serious vulnerabilities under attack, resulting in huge losses. Although most of the existing vulnerability detection tools support the detection of reentry attacks, there is still a long way to go for the great results and efficiency of vulnerability detection and the influence of comprehensive factors to achieve large-scale popularization and use.

3 Attack Mechanism

From the perspective of transaction, reentry vulnerability ultimately achieves the result of multiple transfers, but only sends a transfer request once. When contract A calls the function in contract B through external call, contract A calls contract B again in the same transaction, thus it continues to loop until the remaining gas is exhausted. This type of attack is reentry attack. The most notorious attack is Dao attack. Attackers steal more than 50 million dollars worth of ether from Dao contracts, causing huge economic losses.

```
contract Bank{

  ...

  function withdrawBalance(uint amount){
      if(!(msg.sender.call.value(amount)())){
      revert();
    }
      userBalance[msg.sender]-= amount;
  }

    ...
}
```

The function *withdrawBalance*() of Bank contract is to let the registered account withdraw its assets. When the user calls this function, the contract first checks whether the user's balance is greater than the amount of funds withdrawn. If the check is passed, the assets requested to be withdrawn will be transferred to the user in the form of eth, and the corresponding balance will be deducted from the user's account.

Due to the operation mechanism of Ethereum smart contract, if the received transfer address is a contract address, the fallback function of the address will be triggered. This mechanism may be used by malicious attackers to launch reentry attacks. Attack is an attack contract, and as code shows:

```
contract Attack{

    ...

    function Attack() public{
      Bank.withdrawBalance(100000000000000000);
  }
    function() public payble{
      Bank.withdrawBalance(100000000000000000);
    }

    ...

}
```

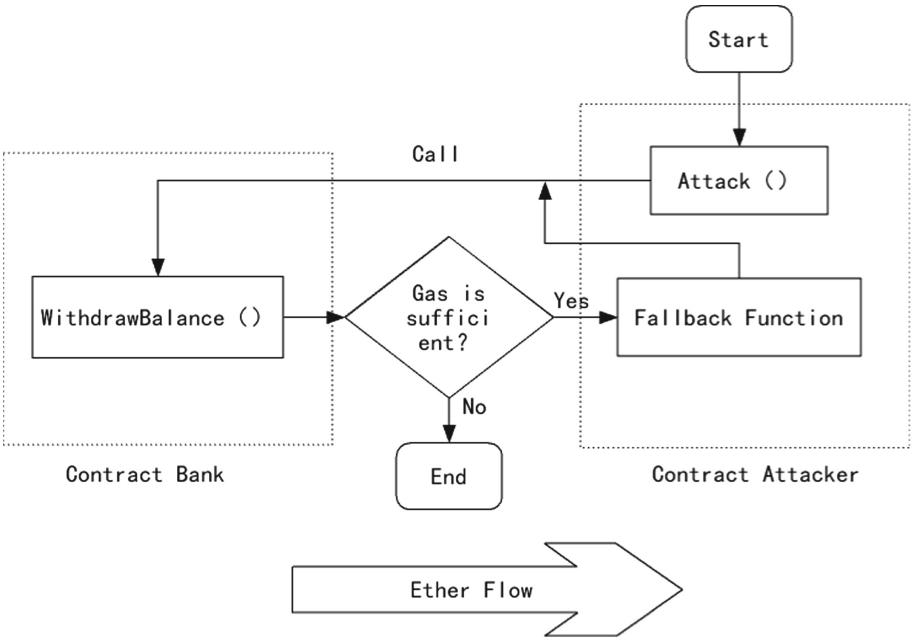


Fig. 1. Flow chart of reentry attack

The attacker only needs to call the victim contract's *withdrawbalance*() function through the function *attack*(), and the victim contract's function *withdraw* − *balance*() of *mag.sender.call.value*(*amount*)() statement is executed, the fallback function in the attack contract will be triggered. If the attacker launches a call to the victim contract withdraw again in this function, before the user's balance

is reduced, Recursion calls are made repeatedly until gas is exhausted, and ether assets in the victim contract are stolen continuously. The whole attack process is shown in Flow chart of reentry attack (see Fig. 1).

The attacker constructs a loop to continuously steal ether in the victim contract. Reentry vulnerabilities need to meet a number of conditions, that is, transfer and storage modification, logical atomic binding, transfer ahead, and transfer using call instruction. Although the reentry vulnerability can be avoided by developers modifying the operation order at the high-level language level, its root cause is the gas mechanism of Ethereum virtual machine call instruction call, fallback function mechanism, recursive access to allow low-level calls and other features.

4 Main Research Methods and Deficiencies

4.1 Program Analysis

Program analysis is a common technology. For smart contract vulnerability detection, it is usually divided into static program analysis and dynamic program analysis [1]. Static program analysis mainly uses the static control flow and data flow information of the program for analysis, while dynamic program analysis can further collect the runtime information of the program [3]. In static analysis, slither [2] performs data flow analysis and taint analysis to detect vulnerabilities in entity programs, but slither execution is limited by other static tools;

4.2 Formal Verification

Formal verification technology is an effective technology to verify whether the program meets the expected design properties and security specifications. Hirai et al.[9,10] use $Isabelle/hol$ to formalize the instruction semantics in Ethereum virtual machine, so as to use it to manually prove the security of a program. It only supports part of the instructions, and it can not fully express the semantics of the supported instructions.

Kevm [12] uses K framework to formalize the semantics of smart contract. K framework is a verification framework based on Reachability logic and independent of high-level language [11]. Kevm tries to use it to implement some semantic related analysis tools on smart contracts [12]. It is difficult for kevm to complete relatively complete program analysis, and it consumes a lot of human work investment, which makes its scalability poor. At the same time, the $f*$ language tried by bhargavan et al. [3] also faces the same problem.

Zeus [13] translates the Solidity source code into the intermediate language of LLVM, uses XACML on it to complete the formulation of verification rules, finally completes the formal verification work through SeaHorn. Although converting Solidity to LLVM as an intermediate language, traditional methods and tools can be used to complete the automatic detection of the program. However, in the process of completing the conversion, whether the semantics of the smart

contract can be correctly converted is an object that needs to be studied, and the Solidity language has some unique properties of its own, which may not be supported by the traditional LLVM intermediate language.

Securify [14] is a typical automated security attribute verification tool, which mainly analyzes the bytecode of the smart contract and makes semantic factual inferences, expresses it through the DataLog language. Securify uses the security attribute verification code defined by DataLog for rule verification. By matching semantic facts with verification rules, the security of smart contracts is determined.

4.3 Symbol Execution

Oyente [16] implements a symbolic execution engine for smart contracts, which is one of the earliest work of automatic vulnerability mining for smart contracts. Oyente includes four components, CFG Builder, Explorer, Core Analysis, and Validator, mainly analyzes bytecode files. CFG Builder analyzes the contract in advance and creates a basic control flow chart to form a topological structure with basic blocks as nodes and jump relationships as edges. And add symbolic execution to solve the problem that the jump relationship between some blocks cannot be determined. Therefore, the main work of symbolic execution is to complete the missing transfer relationship. Explorer is mainly to complete the work of symbolic execution, mainly to traverse the code of each basic block in the control flowchart, and determine the jump relationship between each block. In the process of symbolic execution, the Z3 constraint solver is mainly used to solve the conditions, and Explorer determines the jump relationship according to the results of the constraint solver. Core Analysis is also an important component of Oyente. It completes the design of the model based on the information collected by Explorer and realizes the identification of vulnerabilities in smart contracts. Finally, the Validator filters the analysis results of Core Analysis.

However, some of Oyente's detection schemes are not perfect. In the actual smart contract detection work, a large number of false positives be reported, and the vulnerabilities involved are not comprehensive enough. Only several contract vulnerabilities detection schemes such as conditional competition, timestamp dependency, unchecked return value, and reentrance vulnerabilities are implemented. At the same time, the tool also lacks maintenance and updates by related developers, which can no longer meet the current needs of smart contract development for security vulnerability detection.

After that, more and more researches on automatic vulnerability mining of smart contracts, such as Osiris [4], Mythril [5], Manticore [6], SmartCheck [7], Securify [8], etc., which use symbolic execution technology, begin to appear. They mainly increase the transaction depth of symbolic execution analysis to simulate more real contract execution and explore deeper state space, To achieve more accurate vulnerability detection, but symbol execution faces the problems of path explosion and not easy to expand.

The main idea of symbolic execution is to convert the uncertain input in the process of program execution into symbolic value to promote program execution

and analysis [15]. Due to the influence of branches, loops and other structures, symbolic execution often faces problems such as path explosion. However, compared with traditional applications, smart contract has less code, fewer paths and shorter length, which is more suitable for symbolic execution analysis. However, because the constraints of smart contract are transferred between different paths, the traditional symbolic execution approach can only collect the constraints on a single contract path, not the global constraints. This also brings challenges to the symbolic execution of smart contracts.

5 Detection Method of Dynamic Symbol Execution

This paper proposes a dynamic symbolic execution detection method named M-A-R for smart contract vulnerabilities. It not only combines the advantages of symbolic execution method to solve the security problem of smart contract, but also uses the idea of dynamic symbol to solve the problem of path explosion caused by too large constraint or too deep execution in the process of symbol execution, which greatly saves the time cost and computing resources of detection work. The improvement of M-A-R algorithm makes M-A-R perform well in the efficiency of vulnerability detection.

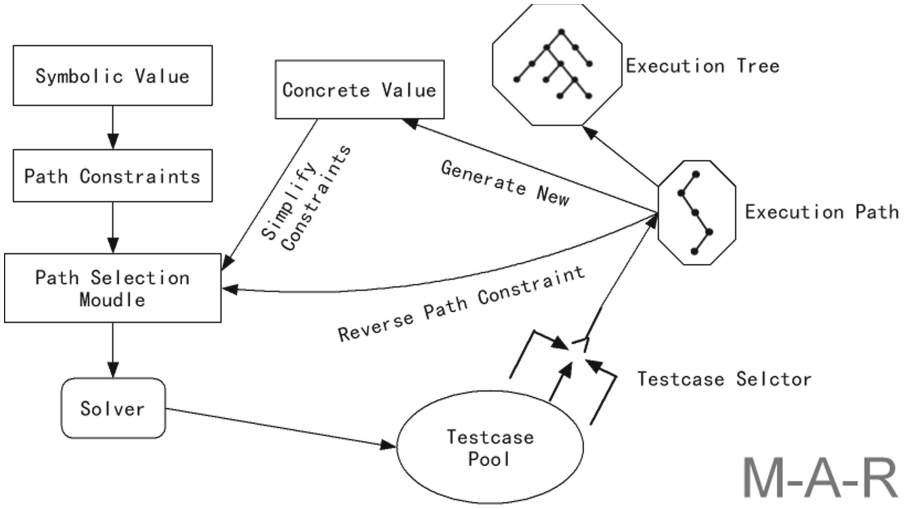


Fig. 2. M-A-R traversal process

The main idea is to use the given or randomly generated initial value to execute the program, but the input is still a symbolic value and form a path constraint. The program needs to select the appropriate branch according to the given or randomly generated initial value, and collect the symbolic constraints associated with the input at the path branch. Finally, according to the set path

search algorithm, the constraint solver is used to reverse the path constraint to get a new input. In theory, the new input value can make the program execute along different paths. This process is then repeated until the program path is explored or the end of exploration condition is reached, or after the scheduled test time has passed. M-A-R traversal process is shown in Fig. 2. The process is briefly described as follows:

I. Enter the Initial Value. The input value is still a symbol value, but the actual value and the symbol value are executed in parallel and enter the path selection module;
II. Path Selection. Input the initial input symbol value or the result of the path constraint inversion into the path selection module to select a path for execution;
III. Generate Path Constraints. Execute the current path to the stop of the leaf node of the path, and generate the constraint conditions of the path;
IV. Solving Path Constraints. Input the path constraint conditions into the constraint solver for solution, and input the actual value of the path to simplify the constraint solution and avoid problems such as path explosion;
V. Generate New Execution Path. After the execution path of a certain path is generated through constraint solving, the constraint conditions are reversed to generate a new execution path;
VI. Generate Execution Tree. Traverse each execution path in turn to finally generate an execution tree;
VII. Attribute verification. Based on the generated execution tree, the verification of smart contract attributes is completed, and the code audit is realized.

M-A-R combines the characteristics of high precision and high coverage of dynamic symbolic execution technology, which can achieve the advantage of zero rate in theory. Therefore, it can well solve the problem of path explosion faced by existing symbolic execution methods in smart contract vulnerability detection. The technology of dynamic symbolic execution has been paid more and more attention in the automatic generation and utilization of test cases, and it also has a certain practical value in practical application.

6 Experiment and Evaluation

The M-A-R algorithm is implemented in Python. Based on the syntax environment of Manticore, the dynamic symbolic execution method is used to simulate the transaction and complete the attribute verification. The existing six smart contract detection tools that support reentry vulnerability detection and perform well are compared with M-A-R synchronously, and the key data such as vulnerability detection results and detection time are recorded, the experimental data set has been uploaded to GitHub (see footnote 1).

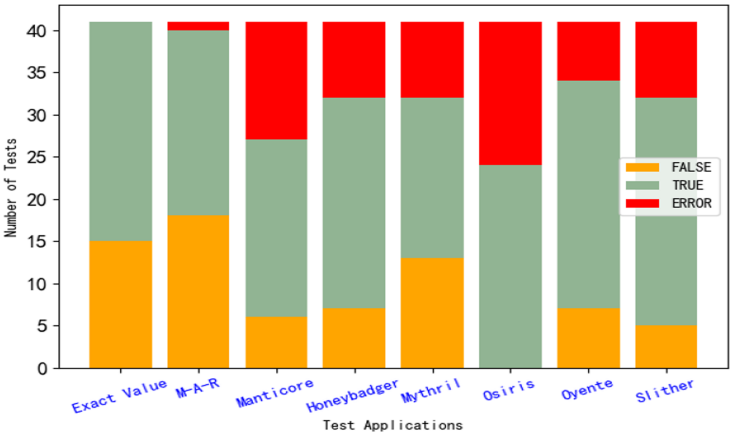


Fig. 3. Proportion of test results

6.1 Simulation Experiment

In vulnerability detection, the error rate will directly affect the test results, so the classification of test results is the most intuitive way.

As shown in Fig. 3, the red part represents the number and proportion of errors reported in the experimental results of vulnerability detection. As can be seen from the figure, Osiris has a higher error reporting rate, while M-A-R has the lowest error reporting rate, and the proportion of detection results is closest to the accurate value, indicating that M-A-R has high practicability and good scalability.

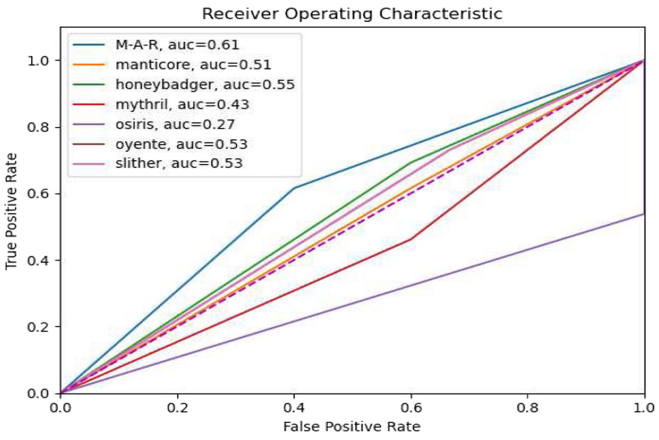


Fig. 4. ROC curve of tools

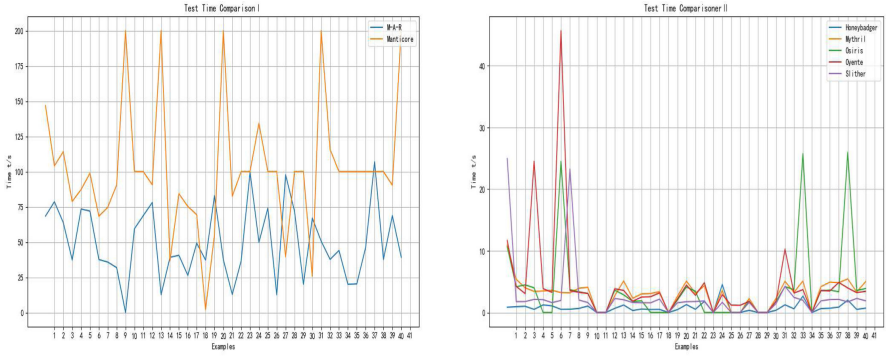
Receiver operating characteristic curve, referred to as ROC curve, can well show the same sensitivity of each point on the curve. AUC, the area under the ROC curve, is a more stable index reflecting the quality of the model. It can be seen from the figure below(see Fig. 4). that M-A-R has the best detection performance for reentry vulnerabilities and osiris model has the worst detection performance under the condition of maintaining high TPR.

6.2 Time Delay Test

The running time of vulnerability detection also has a certain impact on the practicability of vulnerability detection tools, Compared with the traditional formal verification and static analysis, the time cost of M-A-R is much higher.

In this work, will focus on the comparative test between M-A-R and Manticore tools, as M-A-R uses the same language environment as Manticore, and manticoe is also the main tool for vulnerability detection using traditional symbolic execution. It can be seen from the figure that in the face of the same experimental data set, dynamic symbolic execution well shows its advantages and characteristics compared with traditional symbolic execution. Dynamic symbolic execution greatly improves the efficiency of program operation by solving evolutionary constraints.

M-A-R ensures the high reliability of the test results and the superiority of the performance by simulating the real transaction, and adopts the method of dynamic symbolic execution to simplify the constraint solving process, which greatly improves the running time and efficiency of practical application (Fig. 5).



(a) (b)

Fig. 5. Time comparison chart

7 Conclusions and Future Works

The most important feature of M-A-R is to use simulated transaction to verify attribute proof. The experimental data show that: compared with the existing smart contract vulnerability detection tools, M-A-R model has outstanding

reentry vulnerability detection results, which is 30% higher than the average detection results. Compared with the traditional symbolic execution, M-A-R model reflects the advantages of dynamic symbolic execution, and its running speed is 2–3 times of its execution.

To sum up: M-A-R model can greatly reduce the consumption of time and resources, enhance the efficiency of M-A-R, and ensure the high reliability of test results through the characteristics of dynamic symbol execution and the way of testing while verifying when simulating reentry vulnerability transactions.

The dynamic symbolic execution method can simplify the constraint solution and improve the speed of symbolic execution. On this basis, the further works are as follows:

1. It is of great significance to solve other security problems with the idea of dynamic symbol execution for the detection of reentry attacks;
2. The speed of dynamic symbol execution is optimized by improving the path search strategy;
3. Research and implementation of blockchain security audit assistant tool.

Acknowledgements. This research has been supported by the Natural Science Foundation of Hainan Province (No. 620RC605) and Postgraduates' Innovative Research Projects of Hainan Province (No. Hys2020-332).

References

1. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
2. Crytic. Slither (2018).https://github.com/crytic/slither
3. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., et al.: Formal verification of smart contracts: short paper. In: The 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96 (2016)
4. Ferreira, C.O.: (2018). https://github.com/christoftorres/Osiris
5. ConsenSys. Mythril (2017). https://github.com/ConsenSys/mythril-classic
6. Manticore. https://github.com/trailofbits/manticore
7. SmartDec. SmartCheck (2017). https://github.com/smartdec/smartcheck
8. SRI Lab. Securify (2018). https://github.com/eth-sri/securify
9. Hirai, Y., et al.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M. (ed.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
10. Hirai, Y.: Formal verification of deed contract in ethereum name service. In: 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–6 (2019)
11. SeaHorn | A Verification Framework. https://seahorn.github.io/
12. Hildenbrandt, E., Saxena, M., Rodrigues, N., et al.: KEVM: a complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), 9–12 July 2018, pp. 204–217. IEEE, Oxford (2018)

13. Kalra, S., Goel, S., Dhawan, M., et al.: ZEUS: analyzing safety of smart contracts. In: Network and Distributed System Security Symposium, pp. 26–35 (2018)
14. Tsankov, P., Dan, A., Cohen, D.D., et al.: Securify: practical security analysis of smart contracts (2018). ArXiv180601143 Cs
15. Angr/angr. GitHub. https://github.com/angr/angr
16. Luu, L., Chu, D.-H., et al.: Making smart contracts smarter, pp. 254–269 (2016)