

Evaluation of Tools for Analyzing Smart Contracts in Distributed Ledger Technologies

Denis Kirillov(✉), Oleg Iakushkin, Vladimir Korkhov, and Vadim Petrunin

Saint-Petersburg State University, St. Petersburg, Russia
kirillovdenand@gmail.com, {o.yakushkin,v.korkhov,v.petrunin}@spbu.ru

Abstract. Despite the fact that the extent of interest in distributed ledger technologies has slightly decreased after the peak of Bitcoin popularity this area continues to evolve. One of the popular areas is the development of smart contracts which introduces a new paradigm of writing programs. This inflicts additional difficulties associated primarily with the high costs of error. This paper reviews the typical vulnerabilities that are widespread during development in the Solidity language. It also presents an analysis of existing tools to help identify software bugs. It is shown that there is no universal technique at the moment and if the risks are high, one should not solely check the code with available instruments but also conduct a manual audit with help of an expert.

Keywords: Smart contracts · Blockchain · Ethereum · Distributed ledger technologies

1 Introduction

The smart contract is a protocol designed for conclusion and performance of commercial contracts inside the blockchain technology. Its verification is necessary in order to ensure that it is correct before installing the contract into the network. This is a crucial point because the code logic of the work operates mainly with financial data and an error can lead to the loss of a huge amount of money. Furthermore, after finding an error it cannot be eliminated because smart contracts are immutable after they are loaded to the network[1]. This approach is very different from the common idea of companies to release a product as soon as possible even if it is still "not ready", to not miss profits. In this case, updates and bug fixes are delivered during a support period. Despite the fact that the risks in the traditional approach can be high as well, smart contracts often store money directly and thus are bug bounty themselves. This means, if an error is found in the code of the smart contract an attacker can immediately exploit it and get the funds.

[1] There was a well known incident with The DAO when about $ 50 million was stolen due to a program error.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11620, pp. 522–536, 2019.
https://doi.org/10.1007/978-3-030-24296-1_41

Next, we will consider contracts written in the Solidity language for the Ethereum platform since it has the most extensive infrastructure for development. Also this language is most popular among new users. We want to provide an analysis of existing tools that can help check smart contracts for vulnerabilities.

The rest of this paper is organized as follows. Section 2 describes existing approaches to verify smart contracts. Classification of possible problems with smart contracts is given in Sect. 3. The detailed analysis of existing tools is presented in Sect. 4. Section 5 concludes the paper and points out directions of future work.

2 Related Work

The security of smart contracts is a fairly popular area that is actively developed. One possibility is to perform an extensive analysis of smart contracts to detect errors and unsafe patterns of source code as well as propose some classification [1]. Other researchers provide own tools (based on different approaches) for identifying such vulnerabilities.

One approach is to use a static analyzers [2]. An intermediate representation is generated from Solidity code as an XML tree. Afterwards, previously known vulnerability patterns can be found using XPath. The advantage of this approach is full code coverage, which is provided.

Furthermore, in [3] it was suggested that correct-by-design contracts should be developed using a finite-state machine. The authors present the framework FSolidM [4] which allows constructing contracts in terms of finite automata and then generate Solidity code that is based on it.

In [5] an opposite approach is suggested. Here, a framework is presented that allows converting the Solidity code (a restricted subset) or the EVM bytecode to F* code (f star is a functional language that is intended for formal verification using the SMT and manual proofs combination).

NUS[2] researchers also propose their approach based on symbolic execution and using Z3 as solver [6]. They also introduced their open source framework Oyente [7].

In opposition to this, scientists from IBM released ZEUS [8] and showed that their approach is better than the NUS solution (significantly fewer false positives, more vulnerable smart contracts were found by the solution). This tool gets the source code and the specification (that this code must comply with) as input. Then it performs static analysis atop the smart contract code and inserts the policy predicates as assert statements at correct program points. After this, the received data is transferred to LLVM bitcode for subsequent verification of violation of assert policies. However, their solution is commercial.

3 Problem Classification

Before proceeding to the classification of vulnerabilities it is necessary to describe some architectural features of code execution for Solidity. First of all, a smart

[2] National University of Singapore.

contract is a regular computer program that is executed by a virtual machine and is executed on all network nodes. To accomplish this, the user first has to write contract code and then send a transaction that establishes it to the network at a specific address. There is one more limitation: since the contract will run on every node in order to reduce network load each virtual machine operation carries a price which is expressed in Gas. Therefore, for installing a smart contract and also for subsequent calls of it, the author of the transaction is forced to pay a certain number of ethers for each unit of Gas that will be spent. Also, the transaction contains the number of maximum Gas that is allowed to spend. This is done to ensure the absence of some infinite loop in the function of a smart contract which could spend all the funds of the user calling this function.

Two approaches for classification are considered now. It should be kept in mind that the classification is rather conditional and there might be other views that will also be consistent.

1. The first approach is based on the levels that are available from the Ethereum blockchain.
   (a) Solidity – code level.
      i. Call to unknown. During implementation, a bug can occur which calls a contract function, which does not exist. In this case, the fallback function (the function that is called when money comes to the address) will be called which can lead to unpredictable behavior.
      ii. Gasless send. Let you not have enough money to call someone's fallback function by sending money. Then, if you do not check what the send function returns before continuing with the execution of the code, someone will be deprived of the according reward.
      iii. Exception disorder. The appearance of an exception does not always roll back the states of all participants in the call chain. Therefore as in the previous case, it is necessary to check return values of functions before continuing.
      iv. Type cast. There is no exception when wrong type casts are made, thus the calling code is not aware of any of these errors which could occur.
      v. Reentrancy. Details are discussed below.
      vi. Keeping secrets. In contracts some fields should be set private but their values are changed through a public transaction. Here, everyone can find out the new value of the private field.
   (b) EVM – Ethereum Virtual Machine Level.
      i. Immutable Bugs. These bugs are related to the fact that contracts that are loaded into the blockchain cannot be changed if it turns out that it contains errors.
      ii. Ether lost in the transfer. It is possible to send money to an address that does not belong to anyone, hence it will be irretrievably lost.
      iii. Stack size limit. After exceeding the call limit an exception is thrown.

(c) Blockchain technology level.
   i. Unpredictable state. It is impossible to be sure the contract which has got the transaction is in the same state as at the moment when this transaction was initiated.
   ii. Time constraints. The miner which broadcasts a block with transactions to the network has possibility of slightly changing the timestamp on this block. It can lead to vulnerabilities in contracts that use that timestamp.

2. Another approach is based on common, blockchain-specific and networking errors.
   (a) Common errors.
      i. Integer overflow. We work with money, we don‘t want a person who has a lot of money to be left with nothing because of an overflow.
      ii. Array length underflow. Details are discussed below.
   (b) Blockchain-specific.
      i. Reentrancy. Details are discussed below.
      ii. Timestamp dependency. Similar to time constraints in the previous classification.
   (c) Networking.
      i. Reachable exception. Similar to exception disorder in the previous classification.
      ii. Failed send. It is possible to constantly reject the transfer of money.
      iii. Transaction ordering. Similar to unpredictable state in the previous classification.

3.1 More Detailed Description of the Problems

Below is a fairly detailed description of the main vulnerabilities that are widespread in smart contracts at various levels. It also provides examples of code that contains an error and code that exploits this vulnerability.

Array Length Underflow. Since Solidity uses positive integers in the range $[0; 2^{256})$, then subtracting one from zero we get is $2^{256} - 1$. It can be used to change any value in the memory of the contract. This is possible due to the fact that the amount of memory allocated for the contract is 2^{256} slots in which information can be stored. And the addresses of these slots can loop (writing something to the address 2^{256} we change the 0 slot). Thus if we have got a dynamic array that contains 2^{256} elements it will cover the entire memory of the contract. By making a record at a specific index in this array we can change an arbitrary value in the contract. It remains to obtain such an array. This is possible to by subtracting 1 from the length of an empty array. Example of a vulnerable contract:

```
contract Pwnable {
  address public owner;
  address[] a1;

  function Pwnable() public {
      owner = msg.sender;
      a1.length -= 1;
    }
  modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }
    function fun(uint offset, address addr) public
        returns (uint) {
          a1[offset] = addr;
    }
    function transferOwnership(address newOwner) public
        onlyOwner {
        require(newOwner != address(0));
        owner = newOwner;
    }
}
```

Reachable Exception. Suppose we have field A in a contract that contains someone's address A_{old}. Now we want to change this address to another address A_{new} after sending A_{old} some money. But if an exception is thrown in the fallback function of the A_{old} the state will be rolled back (the money will not be recorded on its balance). However, the caller will not get an exception but the value false will be returned. If it is not checked, address A_{new} is assigned to A immediately. A_{old} will not receive the money and will lose the position in the contract and thus the preferred treatment. Contract sample (Fig. 1):

```
contract King {
address public king;
    uint public prize;

    function King() public payable {
        king = msg.sender;
        prize = msg.value;
    }
    function() payable {
        require(msg.value > prize || msg.sender == owner)
            ;
        uint compensation = calculateCompensation();
        king.send(compensation);
        king = msg.sender;
        prize = msg.value;
    }
}
```

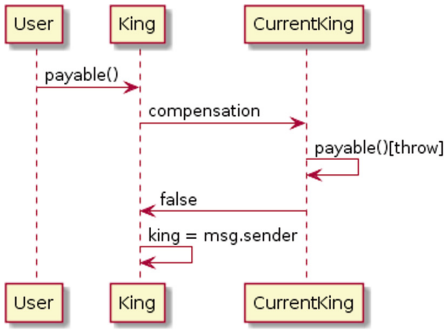


Fig. 1. Reachable exception flow.

Transaction Ordering. Suppose a smart contract which rewards clients if they solve a difficult task. The owner of this contract still has the opportunity to change the compensation of the work. Now, while someone solves the puzzle and agrees to the current reward, the owner changes the amount of the compensation. The transaction of the reward change could reach the contract before the solution of the task is submitted. Then, because of the ordering of the incoming transactions, the resources and time which were needed to solve the puzzle are not rewarded according to the appointed amount (Fig. 2).

```
1  contract Puzzle {
2      address public owner;
3      bool public locked;
4      uint public reward;
5      bytes32 public diff;
6      bytes public solution;
7
8      constructor() public payable {
9          owner = msg.sender;
10         reward = msg.value;
11         locked = false;
12         diff = bytes32 (11111);
13     }
14     function solve(bytes sol) public {
15         require (!locked);
16         if ( sha256(sol) < diff ) {
17             solution = sol;
18             locked = true;
19             msg.sender.transfer(reward);
20         }
21     }
22     function update_reward() public payable {
23         require (msg.sender == owner);
24         require (!locked);
25         owner.transfer(reward);
```

```
26          reward = msg.value;
27      }
28  }
```

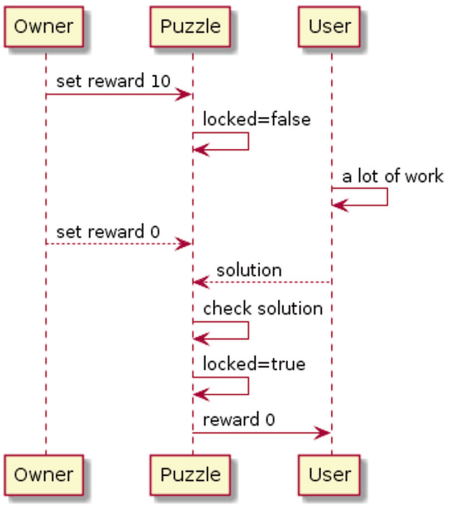


Fig. 2. Transaction ordering flow.

Reentrancy. Let a certain contract keep the money of users and allow them to withdraw their money. This function:

1. Checks the availability of the requested amount of money on the balance,
2. Sends the required amount to the calling user,
3. Reduces the balance.

When receiving money a fallback function is called which may contain some code. So if the withdrawal is called again in this function (even if the first call withdraws all the money) the balance has not decreased. That gives the possibility to receive money until subtraction from the balance occurs at the very end. This will continue until the user removes all the money or reaches the stack size limit (Fig. 3).

Vulnerable contract:

```
1  contract Wallet {
2      mapping (address => uint) private userBalances;
3      address private owner = address(0);
4
5      constructor() public payable {owner = msg.sender;}
6      function deposit(address to, uint val) public {
7          require (msg.sender == owner);
8          userBalances[to] += val;
```

```
    }
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            msg.sender.call.value(amountToWithdraw)();
            userBalances[msg.sender] = 0;
        }
        }
}
```

Attacking contract:

```
contract User {
    function () public payable {
        if (address(wallet).balance >= msg.value){
            wallet.withdrawBalance();
        }
    }
}
```

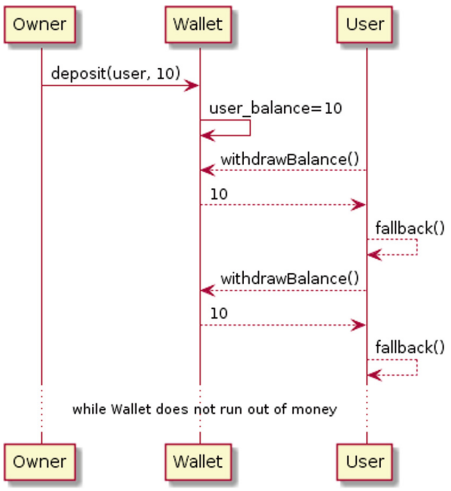


Fig. 3. Reentrancy flow.

Access Control. Often during implementation of smart contract logic you need to verify the sender of a transaction in order to grant certain privileges. However if the *tx.origin* property is used it can lead to issues since this property contains the address of the initiator of the call chain. So if a withdrawal is called that is available only to the contract owner it will not be successful because *tx.origin* will contain our address, not the owner. But by making the same call in the fallback function which will start to be executed if the contract owner transfers some money to us *tx.origin* will contain the owner's address. In such a way we will receive other people's money.

Vulnerable contract:

```
contract Wallet{
  address public owner;

    constructor (address  _owner) { owner = _owner; }
    function () public payable {}
    function withdrawAll(address _luckyAddress) public {
        require(tx.origin == owner);
        _luckyAddress.transfer(this.balance);
    }
}
```

Attacking contract:

```
contract User{
  Wallet poorWallet;
    address attacker;

    constructor (Wallet _poorWallet, address
        _attackerAddress) {
      poorWallet = _poorWallet;
        attacker = _attackerAddress;
    }
    function () payable {
        poorWallet.withdrawAll(attacker);
    }
}
```

Timestamp Dependency. In many business applications it is necessary to operate with precise time. For these purposes one should not use the timestamp on the block. Miners put this mark and in some cases they are free to change the time value up to a few minutes. It is better to use block numbers as timestamps. Therefore, constructions like $(now\%2 == 0)$ should be avoided since an attacker can adjust the time so that this condition will always be fulfilled.

4 Tools

For carrying out a formal verification a corresponding specification is required whereby the validation should be conducted. However, the preparation of such a specification is a laborious and expensive process which may not be justified with frequent changes in requirements. Therefore most of the tools below do not use this approach. We present a list of tools with a brief description and then describe some of them in more detail (Fig. 4).

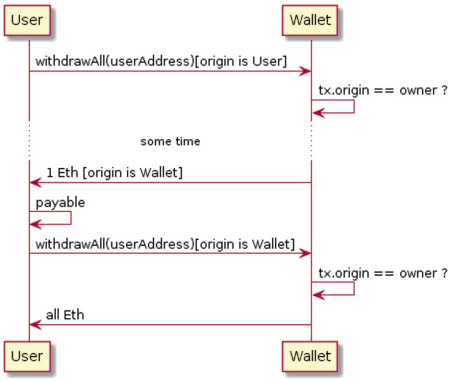


Fig. 4. Access control flow.

- Mythril Classic – uses symbolic execution, taint analysis and control flow checking in order to find security holes.
- Slither – framework for static analysis, allows taint and value tracking.
- Echidna – testing by entering unexpected or incorrect data.
- Manticore – symbolic execution over smart contract bytecode.
- Oyente – symbolic execution over control flow graph.
- Securify – decompiling EVM bytecode and automatic inferencing semantic facts. And using the previously obtained data is looking for patterns of vulnerabilities.
- SmartCheck – static code analyzer using an XML parse tree as an intermediate representation.
- Octopus - using symbolic execution.
- Zeus – using LLVM with policies which is analysed using Seahorn [9].
- Truffle Security – extension for Truffle is based on MythX.
- Mythos – instrument for MythX api.

These tools can be categorized by their kind of approach as followed:

Symbolic Execution. In symbolic execution, the data is replaced by symbolic values with a set of expressions, one expression per output variable. Thus it is possible to receive not just the output of the program depending on certain input but information about the reachable ways of program execution. Also, there is an opportunity for getting the input data which lead to these execution paths. However, there are a number of issues that face symbolic calculations:

1. Unable to handle infinite loops,
2. Problem with handling an external code call,
3. The problem with the rapidly increasing number of possible paths in real applications.

In our case, the following tools fall into this category:

- Mythril,
- Oyente,
- Manticore,
- Octopus.

Using Specification. The use of specifications moves a tool closer to formal verification. So these systems receive some set of conditions (policies) which the code being checked must correspond to. There are some obvious disadvantages of this approach:

1. Check only those conditions that have been set,
2. The need to change policies when changing the code,
3. The difficulty and high development costs.

Tools:

- Zeus,
- Echidna.

Fuzzing. Fuzzing is based on the analysis of the behavior of the program depending on the input data. It involves to use massive amounts of random data as input, called fuzz, in an attempt to make the program crash.

Tools:

- Echidna.

Automatic Inference. This technique allows the use of certain rules to obtain additional facts about the available data automatically.

Tools:

- Securify.

Lexical and Syntactic Analysis. Such programs parse the code received at the input. They transform it into so-called lexemes using some formal language and grammar. Then a certain data structure is formed. It contains syntactic information about the source data with which it is convenient to work in the future.

Tools:

- SmartCheck.

Hybrid Approach. It can use incorporation of symbolic execution, static analysis and dynamic analysis.

Tools:

- Truffle Security,
- Mythos.

4.1 Comparing Tools

Several of the aforementioned tools are open source and were chosen by us for conducting a comparative analysis based on various indicators. Below is a table (Table 1) that contains information whether these systems can determine if a particular type of smart contract vulnerability exists or not.

Table 1. Tools and their efficiency for vulnerability detection. "+" – is detected, "−" – is not detected, "+/−" – it depends, "?" – the vulnerability is detected but it cannot be mapped with the expected type.

	Underflow	Reentrancy	Ordering	Control access	Reachable exception	Timestamp
Mythril	+	+/−	−	+	+	−
SmartCheck	+	?	−	+	+	−
Oyente	−	+	−	−	+	−
Slither	−	+	−	+	−	−
Manticore	−	+	−	+	+	+
Securify	−	+	+	+	−	−

Based on obtained results some conclusions can be drawn. First, it is clear that no tool could detect all vulnerabilities. This is quite obvious since different approaches have their own advantages that have to be paid. In general, this is reflected in limited functionality (the ability to detect only certain types of vulnerabilities). Further, it can be noted that despite the use of identical approaches (for example, symbolic execution) the results may be quite different. This is probably due to the fact that each team of researchers that develops an analyzer tries to solve an individual task and adds specific functionality without trying to make the tool universal. In addition, you can see that the most difficult vulnerability to determine is the dependence on the order of transactions and the dependence on time. In the first case, it is quite clear: there is a dependence on time that adds additional complexity to the analysis. Regarding the use of the timestamp, it can be assumed that a simple pattern that depends on time (for example, $now\%2 == 0$) will give a large number of false positives and thus such patterns are not added. However it is worth noting that false positives are not so terrible since the developer will simply have to manually double-check the code. And even then just time is lost in this case. The presence of false negatives has a more harmful impact. Actually, the presence of such situations is indeed equivalent to the absence of verification. This happens if we do not include patterns with false positive in the verification rules. Thus, it is an indirect argument for that in some cases the developers of analysis tools are trying to achieve certain indicators that in general may not depend on reliability and security of smart contracts. It is an indirect argument for some cases, when developers of analysis

tools are trying to achieve certain indicators that in general may not be associated with reliability and security of smart contracts. However, in matters of money security, it is better to be on the safe side.

Table 2 contains data that shows the analysis tools comparison for other criteria. In particular, it is shown that almost all the presented analysis systems are currently maintained and under active development. This indicates that the tools will improve and soon the previous indicators (Table 1) can be extended. It is also worth noting the most popular approach is to use symbolic execution. Teams following this approach, more often choose Z3 as the SMT solver [6]. Almost all tools were presented in academic papers. This may indicate that there is a fairly strong relationship between the academic environment and business. It is an emphatic success for the area of the distributed ledger technologies since it allows a decreased time period from appearance of a theoretical result to its implementation and active deployment in the industrial sector. The most popular language for implementation is Python.

Table 2. Tools and their common properties.

	Maintain	Type	Paper	Language	Using components
Mythril	Yes	Symbolic execution	Yes	Python	Z3
SmartCheck	Yes	Static analysis	Yes	Java	XML parse tree
Oyente	No	Symbolic execution	Yes	JavaScript	Z3
Slither	Yes	Taint and value tracking	No	Python	LLVM
Manticore	Yes	Symbolic execution	No	Python	Z3
Securify	Yes	Automatic inference	Yes	Java	Z3

As mentioned above many tools using symbolic execution contain some theorem provers or SMT solver for internal computations.

Some of these tools are:

1. Coq is a cross-platform interactive tool for theorem proving that uses its own functional language.
2. CVC3 is an automatic theorem prover. It can process first-order formulas.
3. Z3 is an SMT solver presented by the Microsoft researchers. It is the most popular tool.
4. F* (f star) is a functional language that combines proof assistant and SMT solver. It was also introduced by the Microsoft researchers.
5. Isabelle is a general-purpose proof assistance that enables process a high-order logic.

The comparative Table 3 displays some of the characteristics of these tools. The “extraction” and “code generation” columns should be explained. The first

one is responsible for the possibility of translating code written using the presented tools into any other language. The last one is responsible for the ability to generate code from any languages. Why3 is a platform for program verification that allows using third-party provers. The language for writing code is WhyML.

Table 3. Tools for theorem proving and their common properties. "?" – no data. "-" – no extraction.

	IDE	Extraction	OS	Language	API	Code generation
Z3	No	?	L/M/W	C/C++	C/C++, .NET, OCaml, Python, Java	Why3
Coq	Yes	Haskell, Scheme, OCaml	L/M/W	OCaml	?	Why3
CVC3	No	-	L	C/C++	C/C++	Why3
F*	No	OCaml, F#, C	L/M/W	F*	?	Solidity*
Isabelle	Yes	SML, Scala, OCaml, Haskell	L/M/W	Standard ML, Scala	?	Why3

5 Conclusion and Future Work

In this paper we review the existing problems and vulnerabilities which are quite common in smart contracts on the Ethereum platform, in particular for the Solidity language. This topic is important because a growing number of people are coming to this area and they need to start thinking in a somewhat different paradigm in order to avoid losing a lot of money and reputation. Formal verification of programs is a rather complicated and expensive process which is why most of the existing tools do not use this approach. We analyzed several of the most popular tools for searching for vulnerabilities and came to the conclusion that, at the moment, there is no completely versatile and balanced tool. Therefore if there is a need to ensure the reliability and security of a smart contract you should either use all available tools or approach a specialized company that examines smart contracts. And it is better to do both.

As the blockchain technology field evolves and new projects appear that allow smart contracts to be written in general purpose languages (Java, Go, Javascript, Python), it is planned to consider formal verification tools that are suitable for these languages and determine the specifics arising from such analysis.

References

1. 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
2. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: SmartCheck: static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 9–16. ACM, Gothenburg (2018). https://doi.org/10.1145/3194113.3194115
3. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: a finite state machine based approach. In: 22nd International Conference on Financial Cryptography and Data Security (2018)
4. Mavridou, A., Laszka, A.: Tool demonstration: FSolidM for designing secure ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 270–277. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_11
5. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, Vienna, Austria, 24 October 2016, pp. 91–96 (2016). https://doi.org/10.1145/2993600.2993611
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 254–269 (2016). https://doi.org/10.1145/2976749.2978309
8. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: 25th ISOC Symposium on Network and Distributed System Security (NDSS 2018), San Diego, CA (2018). https://doi.org/10.14722/ndss.2018.23092
9. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 54–70. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_4
10. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_34
11. Paulson, L.C.: Isabelle: the next 700 theorem provers. In: Odifreddi, P. (ed.) Logic and Computer Science, pp. 361–386. Academic Press, London (1990)