On Verification of Smart Contracts via Model Checking[⋆]

Yulong Bao[1,2], Xue-Yang Zhu[1,2], Wenhui Zhang[1,2], Wuwei Shen[3], Pengfei Sun[1,2], Yingqi Zhao[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of the Chinese Academy of Sciences, Beijing, China
[3] Department of Computer Science, Western Michigan University, USA
`{baoyl, zxy, zwh}@ios.ac.cn,wuwei.shen@wmich.edu, and {sunpf, zhaoyq}@ios.ac.cn`

Abstract. Combined with smart contracts, the application of blockchain techniques has grown faster and broader. However, it is very difficult to write secure and functionally correct smart contracts because of the openness of blockchain platforms. Formal verification, such as model checking, has been proven to be an effective way of guaranteeing security and correctness of systems. In this paper, we propose a novel model checking based framework, called mcVer, to support the verification of smart contracts written in Solidity. Built on model checking tool VERDS, the mcVer framework is able to verify not only safety properties but also liveness properties of smart contracts. For the properties that are not satisfied, mcVer produces a counter example by showing a sequence of statements in the original Solidity program as a hint for fault localization. We implemented the automatic transformation from a subset of the Solidity language to the modeling language of VERDS, that therefore provides automatic verification for smart contracts. Experiments are carried out on various cases, including checking contracts for finding typical security vulnerabilities and verifying properties of an access control smart contract. The experimental results demonstrate the flexibility and efficiency of mcVer.

1 Introduction

Since Bitcoin was first introduced in 2009 [32], the blockchain has been regarded as a promising but yet challenging technology. According to a recent study by Garther [16], the market value for the blockchain-based technology can exceed $3.1 trillion by 2030. Many cloud platform giants such as Microsoft, IBM, Amazon, Oracle have proposed Blockchain-as-a-Service (BaaS) solutions to support various enterprise scenarios such as financial services and supply chains. These solutions are based on programs called smart contracts [40]. However, due to the openness of blockchain platforms, people have suffered some devastating consequence caused by the errors in smart contracts. For instance, the infamous The

[⋆] This work is partially supported by the National Natural Science Foundation of China (No. 62072443).

DAO exploit [4] resulted in the loss of almost $60 million worth of Ether, and the Parity Wallet error caused $169 million worth of Ether to be locked forever [41]. Obviously, unsafe smart contracts not only result in huge financial loss but also seriously undermine the confidence about the development of blockchain-based technologies.

Realizing the importance of safe and secure smart contracts, researchers have conducted various types of studies [27] [25] [44] to reveal the nature of unsafe smart contracts as well as to detect such problems. As reported, many common vulnerabilities in the blockchain domain are related to nondeterminism [46], which is actually caused by concurrent calls. Model checking [17] has been successfully applied in the verification for many modern software systems with the concurrency feature. Generally, model checking techniques can be used to check various properties expressed by temporal logics [28], including safety properties and liveness properties. A *safety property* specifies that something bad never happens, while a *liveness property* specifies that something good will eventually happen, playing an important role in the correctness of smart contracts. Security requirements are usually a kind of safety properties.

While model checking techniques are promising for reducing the potential threats and errors of smart contracts, they currently require the skill of writing formal models and logic formulas, which is challenging for engineers who do not have solid mathematical background. Learning how to write formal models and specifications increases the learning curve and thus reduces the applicability of these techniques. We propose in this paper a novel framework **mcVer** to ease the difficulty of using model checking techniques and deal with the diversity of smart contract properties.

Our framework **mcVer** uses model checking tool VERDS [50] as the foundation to support the verification of smart contracts written in Solidity, which is a smart contract language of Ethereum [47]. The contributions of this paper are summarized as follows.

1. The **mcVer** framework is able to verify not only safety properties but also liveness properties of smart contracts. For the properties that are not satisfied, mcVer produces a counter example by showing a sequence of statements in the original Solidity program as a hint for fault localization.
2. We provide automatic verification for smart contracts by means of the the automatic transformation from a subset of the Solidity language to the modeling language of VERDS.
3. We implement the framework and apply it to several case studies, including checking some typical security vulnerabilities and verifying properties, including liveness properties, of an access control smart contract [51].

The remainder of this paper is organized as follows. Related work is reviewed in Section 2. Solidity and VERDS are introduced in Section 3. We present the general idea of **mcVer** framework in Section 4 and illustrate the technical details in Sections 5, 6 and 7. Case studies and experiments are shown in Section 8. Section 9 concludes and discusses limitations of **mcVer** and the future work.

2 Related Work

Luu Loi [27] and Atzei [9] summarized several types of common security vulnerabilities, such as Transaction-Ordering Dependency (TOD) and Reentrancy Vulnerability. Based on these types of common security vulnerabilities, a number of research efforts have been made to ensure the safety and security for smart contracts. The most explored are testing based techniques [21] [48] and static analysis-based techniques, usually based on the symbolic execution, including Oyente [27], Securify [45], Slither [18], ETHBMC [20], Mythril [42], SolMet [22], SmartCheck [43], Ethainter [12], Manticore [30], VeriSmart [38] and FAIRCON [26]. However, these approaches heavily depend on known security patterns to detect errors and cannot guarantee the functional correctness in blockchain-based applications.

To overcome the limitation of above-mentioned approaches, formal verification techniques [15] have been proposed to verify the correctness of smart contracts. Research on formal verification of smart contracts starts with theorem proving based approaches. Hirai [24] uses theorem prover Isabelle [35] to verify smart contracts [23] and uses the Lem [31] to define a formal model for the Ethereum virtual machine (EVM). Bhargavan et al. [11] convert Solidity and EVM bytecode contracts into functional programming language F* [39]. Nehai et al. [34] translate smart contracts into models of Why3 [19]. The verification procedures in this kind of approaches generally require user's interaction and are difficult to deal with liveness properties. The verification procedure of **mcVer** is automatic and not only safety properties but also liveness properties can be verified in **mcVer**.

Model checking based techniques are also studied by many researchers. Albert et al. [7] use existing verification engines developed for C programs [10] to verify safety properties of low-level EVM code. Sukrit Kalra et al. [25] present ZEUS, which uses bounded model checking techniques to verify safety properties of smart contracts. Anton Permenev et al. [36] present VerX, which combines reduction of temporal safety verification to reachability checking, with symbolic execution and delayed abstraction. Mavridou [29], Nehai [33] and Abdellatif [8] present their work based on the model checking tool NuSMV [14]. A model-based framework VeriSolid is proposed in [29], which focuses more on the code generation procedure rather than verifying Solidity source code. Authors of [33] and [8] present work that verifies particular smart contracts. The former focuses on the contracts in the energy market field and the latter tries to verify a supply chain management smart contract. The NuSMV based methods are possible to deal with more properties, but existing work does not provide methodological techniques to support the verification of commonly developed contracts. Our framework **mcVer** is able not only to verify various properties but also implements the related tool to smooth the verification procedure that starts directly from the Solidity source code. The tool may return a counter example when a property is not satisfied, providing a further help for debugging. To the best of our knowledge, none of the formal method based work for the verification of smart contracts is able to provide such a functionality.

3 Solidity and VERDS

3.1 Solidity

```
pragma solidity ^0.4.22;
contract Auction{
    address public bene;
    address public hBidder;
    uint public hBid;
    bool ended;
    constructor(address _beneficiary)
         public {
        bene = _beneficiary;
    }
    function bid() public payable{
        require(!ended);
        require(msg.value > hBid);
        if (hBid!=0 ) {
            require(hBidder.send(hBid)
                 );
        }
        hBidder = msg.sender;
        hBid = msg.value;
    }
    function aucEnd() public{
        require(!ended);
        ended = true;
        bene.send(hBid);
    }
}
```

Fig. 1. aucSC, a smart contract for auction.

Solidity is a programming language developed to write smart contracts that run on the EVM. A Solidity smart contract mainly consists of two parts, the variable declaration and the function definition. The former defines state variables used by the contract and the latter specifies the potential behavior of the contract. Fig. 1, for example, shows partial code of our running example, aucSC, a simple auction contract modified from [1].

State variables of a smart contract are variables whose values are permanently stored in the blockchain and each has a type. For example, variables *bene* and *hBidder*, indicate addresses of the beneficiary and the current highest bidder of the auction, respectively. Variable *hBid* denotes the current highest bid and variable *ended* indicates whether an auction is ended or not.

Besides global state variables, there are implicit global variables that are defined by the EVM, including related accounts and balances. Once an account is created on the blockchain, it has an address as its identification and a variable to record the changes of its balance. When an account uses a smart contract, its address is the value of *msg.sender*, which is used but not defined in the contract. See aucSC for example. Accounts related to a smart contract include the account that deploys it (its owner), the accounts that use it (its users), and some accounts specified by the contract, e.g. the address of the beneficiary in sucSC. The balances related include its own balance and balances of the related accounts.

A function consists local variable declaration and statements. Functions can receive data via parameters, perform computation, manipulate state variables, and interact with other accounts. Functions are defined to operate on the states of the contract. The *constructor()* is a special function that is executed only once when the contract is deployed on the blockchain. The account who calls *constructor()* is the owner of the contract. Other functions can be called and executed many times during the lifecycle of the contract. For example, function *bid()*, which defines the bidding behavior, can be executed by arbitrary number of users before *aucEnd()*, which sets variable *ended* to be true. A function may also operate on the implicit global variables. An execution of a function may receive *msg.value* amount of money from its trigger *msg.sender*, and change the balances of related accounts.

We consider a subset of Solidity language that are sufficient to express most contracts. The supported subset is summarized in Fig. 2.

$$\begin{array}{rl}
\mathit{type} ::= & \mathit{address}|\mathit{bool}|\mathit{uint}|\mathit{mapping}|\mathit{enum}|\mathit{array}|\mathit{struct} \\
\mathit{operator} ::= & +|-|*|/|++|--|+=|-=|*=|/= \\
\mathit{logic\ op} ::= & || \mid \&\& \mid > \mid < \mid >= \mid <= \mid ! \\
\mathit{statement} ::= & \mathit{assignment} \mid \mathit{condition\ statement} \\
 & \mid \mathit{for\ statement} \mid \mathit{while\ statement} \\
 & \mid \textbf{continue} \mid \textbf{break} \mid \textbf{return} \mid \textbf{throw} \mid \textbf{require}
\end{array}$$

Fig. 2. Subset of Solidity language **mcVer** supports

A call to a function of a contract activates an execution. The execution terminates after successfully updating the state of blockchain, or aborting and rolling back to the state before the call. While smart contracts allow the concurrent calls, the executions of concurrent calls are sequential due to the execution model of EVM [37]. We take this into account when formalize smart contracts.

3.2 Model Checking Tool VERDS

Model checking is considered one of the most practical applications of theoretical computer science in the verification of concurrent systems. The basic idea of model checking is to use the state transition system (S) to represent the behavior of the system, and the modal formula (F) to describe the properties of the system. In this way, the question of whether the system has the desired properties can be transformed into a question of whether S satisfies F. For finite state systems, this problem is algorithmically decidable. We use VERDS as the verification engine. VERDS is a model checking tool that has been applied in many aspects, such as the verification of SystemC design [49] and the verification of multi-agent systems [13]. The input to a model checking tool usually includes a system model and a specification. The modeling language of VERDS is called VERDS modeling language (VML). A verification model specified in VML is called a VERDS verification model (VVM), including a system model defined by the *guarded command transition systems* and specifications expressed by the *computation tree logic* (CTL).

Suppose p is any propositional atom, then CTL has the syntax given as follows.

$$\begin{array}{l}
\Phi ::= p|\neg\Phi|\Phi \wedge \Phi|\Phi \vee \Phi|A\Psi|E\Psi \\
\Psi ::= X\Phi|F\Phi|G\Phi|(\Phi U \Phi)
\end{array}$$

Among them, Φ is the CTL formula and Ψ is the auxiliary path formula. The set of operators of CTL formula is divided into path operators and temporal operators. There are two kinds of path operators: $A\Psi$ indicates that on all paths Ψ should be true, and $E\Psi$ indicates that Ψ should be true on at least one path. The temporal operators are X, F, G, U. Two kinds of temporal operators are used in this paper: $F\Phi$ indicates that Φ will eventually be true on a certain state on a path; $G\Phi$ indicates that Φ should be satisfied for all the states on a path. Properties to be checked on guarded transition system can be of various kinds:

- safety: ‘something bad never happens’ is usually expressed in CTL as $AG(\neg p)$;
- liveness: ‘something good will eventually happen’ is usually expressed in CTL as $AF(p)$.

Each VVM consists of six parts: global alias definition, global variable declaration, initialization, module definition, process instantiation, and property specification. These six parts are distinguished by keywords DEFINE, VAR, INIT, MODULE, PROC and SPEC, respectively. Variables should be bounded.

A module in VVM is a template of a transition system, defined under keyword MODULE. Each module consists of four parts: module identifier and list of parameters, local variables declaration (VAR), local variables initialization (INIT), and collection of transitions (TRANS). A transition consists of two parts: logical expressions (guard) and assignments (command). When the logical expression is true, the assignment statement will be executed atomically. When logical expressions in different transitions are true at the same time, a random one of them will be executed. A process in PROC part is an instance of a module; a local variable x in a process $p0$ can be accessed using the form $p0.x$. Properties under verification are specified in SPEC part. If a property does not hold, a counterexample can be found in the CEX file returned by VERDS.

4 Overview of mcVer Framework

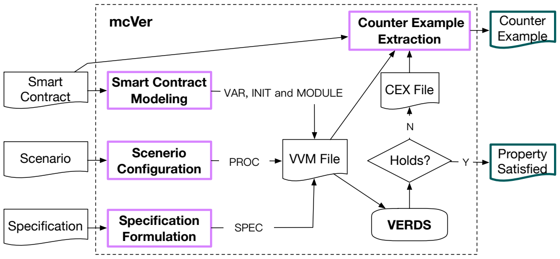
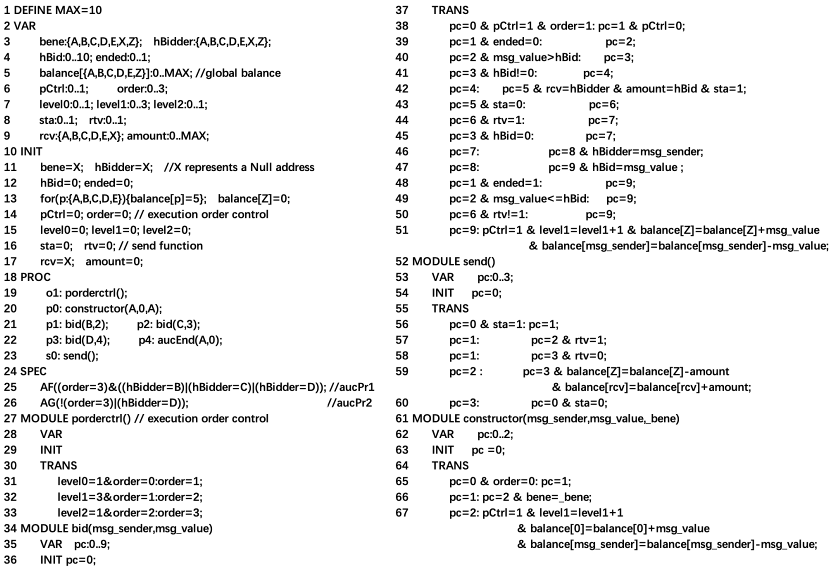


Fig. 3. Overall Framework of **mcVer**.

The framework of **mcVer** is shown in Fig. 3, in which our main contributions are in boxes with purple border. The smart contract under verification are either entirely or partly translated into the VAR, INIT, and MODULE parts of a VVM; the scenario corresponds to the PROC part, and the specification is formulated as CTL formulas in the SPEC part. The model checker VERDS is then used to verify whether the smart contract satisfies required properties. If a property is not satisfied, VERDS returns the trace indicating the problematic behavior in a CEX file. A counter example on the Solidity level is then extracted from it.

We present *Smart contract modeling* in the next section, *Scenario Configuration* and *Specification Formulation* in Section 6, and *Counter Example Extraction* in Section 7. Smart contract aucSC (Fig. 1) will be used as running example to help illustrating our ideas. Part of its corresponding VVM, aucVVM, is shown in Fig. 4. Details will be explained in the later sections.



```
1 DEFINE MAX=10
2 VAR
3       bene:{A,B,C,D,E,X,Z};   hBidder:{A,B,C,D,E,X,Z};
4       hBid:0..10; ended:0..1;
5       balance[{A,B,C,D,E,Z}]:0..MAX; //global balance
6       pCtrl:0..1;       order:0..3;
7       level0:0..1; level1:0..3; level2:0..1;
8       sta:0..1;   rtv:0..1;
9       rcv:{A,B,C,D,E,X}; amount:0..MAX;
10 INIT
11      bene=X;   hBidder=X;   //X represents a Null address
12      hBid=0; ended=0;
13      for(p:{A,B,C,D,E}){balance[p]=5};   balance[Z]=0;
14      pCtrl=0; order=0; // execution order control
15      level0=0; level1=0; level2=0;
16      sta=0;   rtv=0; // send function
17      rcv=X;   amount=0;
18 PROC
19        o1: porderctrl();
20        p0: constructor(A,0,A);
21        p1: bid(B,2);       p2: bid(C,3);
22        p3: bid(D,4);       p4: aucEnd(A,0);
23        s0: send();
24 SPEC
25      AF((order=3)&((hBidder=B)|(hBidder=C)|(hBidder=D)); //aucPr1
26      AG(!(order=3)|(hBidder=D));                          //aucPr2
27 MODULE porderctrl() // execution order control
28      VAR
29      INIT
30      TRANS
31          level0=1&order=0:order=1;
32          level1=3&order=1:order=2;
33          level2=1&order=2:order=3;
34 MODULE bid(msg_sender,msg_value)
35      VAR   pc:0..9;
36      INIT pc=0;
37      TRANS
38          pc=0 & pCtrl=1 & order=1: pc=1 & pCtrl=0;
39          pc=1 & ended=0:              pc=2;
40          pc=2 & msg_value>hBid:      pc=3;
41          pc=3 & hBid!=0:            pc=4;
42          pc=4:      pc=5 & rcv=hBidder & amount=hBid & sta=1;
43          pc=5 & sta=0:               pc=6;
44          pc=6 & rtv=1:               pc=7;
45          pc=3 & hBid=0:              pc=7;
46          pc=7:                  pc=8 & hBidder=msg_sender;
47          pc=8:                  pc=9 & hBid=msg_value ;
48          pc=1 & ended=1:              pc=9;
49          pc=2 & msg_value<=hBid:     pc=9;
50          pc=6 & rtv!=1:              pc=9;
51          pc=9: pCtrl=1 & level1=level1+1 & balance[Z]=balance[Z]+msg_value
                           & balance[msg_sender]=balance[msg_sender]-msg_value;
52 MODULE send()
53      VAR      pc:0..3;
54      INIT     pc=0;
55      TRANS
56          pc=0 & sta=1: pc=1;
57          pc=1:             pc=2 & rtv=1;
58          pc=1:             pc=3 & rtv=0;
59          pc=2 :           pc=3 & balance[Z]=balance[Z]-amount
                                  & balance[rcv]=balance[rcv]+amount;
60          pc=3:             pc=0 & sta=0;
61 MODULE constructor(msg_sender,msg_value,_bene)
62      VAR      pc:0..2;
63      INIT     pc =0;
64      TRANS
65          pc=0 & order=0: pc=1;
66          pc=1: pc=2 & bene=_bene;
67          pc=2: pCtrl=1 & level1=level1+1
                         & balance[0]=balance[0]+msg_value
                         & balance[msg_sender]=balance[msg_sender]-msg_value;
```

Fig. 4. aucVVM, the corresponding VVM of aucSC.

5 Smart Contract Modeling

The behavior of smart contracts is defined by the *guarded command transition system*. To be intuitive, we describe the semantics of smart contracts directly with the VML. We discuss key points of mapping behavior of the EVM, variables, functions and function call in smart contracts to VVMs in this section.

Behavior of EVM Due to the execution model of EVM, transactions are executed in a single-threaded manner. A boolean variable *pCtrl* is declared to control the execution of processes and is false by default. Only when *pCtrl* is true, a process except for that of *constructor* can move to the next step. When a process ends its execution, it releases the control by setting *pCtrl* to be true. Function *constructor* executes only once before all other functions. It needs not to check *pCtrl* to go. For example, a process instantiated from module *bid* in Fig. 4 is waiting until the control variable *pCtrl* becoming true (Line 38) to start its execution and change *pCtrl* to be false to block other process. It sets *pCtrl* to be true at the end of its execution (Line 51).

Related balances are explicitly defined as a global array *balance* in VVM. They are defined at Line 5 and initialized at Line 13 in Fig. 4, for example. The trigger address *msg.sender* and the amount of money *msg.value* are modeled as two parameters of each module in the VVM. See Line 34 in Fig. 4 for example. Changes of balances of related accounts are coded in the last transition of the module of each function.

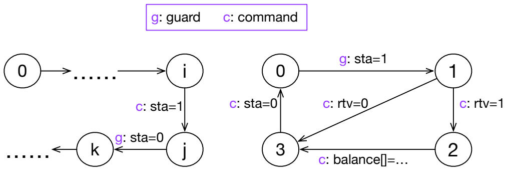
State Variables State variables of smart contracts are accordingly defined as bounded integer (in the VAR part) and initialized (in the INIT part) in the

VVM. For readability, we use characters to represent the values of variables with address type. The address of contract under verification is set to be 'Z' by default. For example, Lines 3-4 and 11-12 in Fig. 4 are definition and initialization of state variables of aucSC in aucVVM. Constants is defined in the DEFINE part.

Functions A function in a contract is modeled by a module in VVM, which is a guarded command transition system. By this we in fact define a transition system semantics for the behavior of contracts. Let S be the set of statements in a function and P_s be the label of statement s. Each statement in S is modeled as one or multiple transitions in VVM. Each module in VVM has an extra local variable pc to model the change of P_s. The mapping of statements in the subset of Solidity (Fig. 2) to VML is shown in Table. 1. The module in VVM of *bid()* in aucSC (Fig. 1) is shown in Fig. 4 (Lines 34-51). Line 34 declares the module identifier.

Table 1. Statements Mapping

statement	solidity	VML
assignment	P_{s1} $y = e;$ P_{s2} $\ldots$	$pc = P_{s1} : y = e \,\&pc = P_{s2};$
condition	P_{s1} $if\ (cond)\ \{$ P_{s2} $\ldots$ $\}else\{$ P_{s3} $\ldots\}$	$pc = P_{s1}\ \&cond : pc = P_{s2};$ $pc = P_{s1}\ \&!cond : pc = P_{s3};$
while	P_{s1} $while(cond)\{$ P_{s2} $\ldots$ P_{s3} $\}$ P_{s4} $\ldots$	$pc = P_{s1}\ \&cond : pc = P_{s2};$ $pc = P_{s1}\ \&!cond : pc = P_{s4};$ $pc = P_{s3} : pc = P_{s1};$
for	P_{s1} $for(i = 0; cond; st)\{$ P_{s2} $\ldots$ P_{s3} $\}$ P_{s4} $\ldots$	$pc = P_{s1}\ \&cond : pc = P_{s2};$ $pc = P_{s1}\ \&!cond : pc = P_{s4};$ $pc = P_{s3} : st\ \&pc = P_{s1};$
break	$while(cond_1)\{$ $if(cond_2)\{$ P_{s1} **break**; } $\ldots\}$ P_{s2} $\ldots$	$pc = P_{s1} : pc = P_{s2};$
continue	P_{s1} $while(cond_1)\{$ $if(cond_2)\{$ P_{s2} **continue**; } $\ldots\}$	$pc = P_{s2} : pc = P_{s1};$
throw **retrun**	function $foo()\{$ $if(cond)\{$ P_{s1} **throw**; } $\ldots$ P_{s2} $\}$	$pc = P_{s1} : pc = P_{s2};$
require	function $foo()\{$ P_{s1} $require(cond);$ P_{s2} $\ldots$ P_{s3} $\}$	$pc = P_{s1}\ \&cond : pc = P_{s2};$ $pc = P_{s1}\ \&!cond : pc = P_{s3};$



(a) a process of *bid()* (Caller)

(b) a process of *send()* (Callee)

Fig. 5. Illustration of function call.

Function Call We use shared variables to handle calls between functions, which may be in the same contract or not. The caller and callee can execute in parallel, with the blocking feature. Once one function is active, the other one is blocked. Some callees may return uncertain values because they are defined outside and controlled by the environment, such as functions *transfer()* and *send()*. We model this uncertainty by using non-deterministic choice of transitions in a module.

A module to model function *send()* called in *bid()* in aucSC is shown in Lines 52-60 in Fig. 4, for example. Auxiliary variables *sta*, *rtv*, *rcv* and *amount* are defined as global variables (Lines 8-9) to model starting or ending the process, returned value, the address of the sender and the amount of money sent, respectively. A process of module *send()* starts when $sta = 1$ and sets *sta* to 0 and returns *rtv* when it finishes. At Lines 57 and 58, VERDS randomly chooses one transition to execute and so is the value of *rtv*. This makes the returned value non-deterministic. When *bid()* is trying to call *send()* (Line 42), it sets *sta* to 1 and then the execution does not resume until *sta* becomes 0. The interaction between the caller and the callee in a function call is depicted in Fig. 5.

6 Scenario Configuration and Specification Formulation

6.1 Scenario Configuration

A scenario defines how a contract is used. Since smart contracts are deployed on the blockchain, which is an open environment, the order of calls to its functions are not deterministic. This uncertainty is modeled as concurrency in **mcVer**. In some cases, e.g. sequential transactions submitted by the same account, the corresponding functions, i.e. callees, should be scheduled to execute sequentially.

We propose a scenario definition language (SDL), which is simple but capable of expressing both sequential and concurrent executions of function. Sequential execution of functions is separated by line shift and concurrent execution of functions is separated by symbol '|'. For example, in contract aucSC shown in Fig. 1, users can only call function *bid()* after the contract has been deployed. After its deployment (*constructor()*); there may be multiple users bidding at the same time; the owner may call function *aucEnd()* to end it. Suppose the owner is A and there are three bidders B, C and D bidding with amount 2, 3, 4, respectively. After the owner has deployed the contract on the blockchain, the three bidders can bid in any order. Then the owner ends the auction. The scenario, named aucSNR, is formulated in SDL as:

$$\begin{array}{l} p0 : constructor(A, 0, A) \\ p1 : bid(B, 2) \mid p2 : bid(C, 3) \mid p3 : bid(D, 4) \\ p4 : aucEnd(A, 0) \end{array}$$

This scenario is then translated into the PROC part of the VVM (Lines 18-23 in Fig. 4). The three biddings can be executed in any order, but can only be executed sequentially due to the execution model of the EVM. The execution order is controlled by variable *order*. The variable *level**i*** indicates how many processes have been executed so far under *order=i*. Intuitively, variable *order* controls line execution, while *level**i*** controls the execution of processes in the same line. For scenario aucSNR, for example, we have $level\mathbf{0} \in [0, 1]$ and $level\mathbf{1} \in [1, 3]$ because only one process $p0$ is executed under $order = 0$ but three concurrent processes $p1$, $p2$, $p3$ are executed under $order = 1$. A process control module *porderctrl()* is used to help execution order (Lines 27-33 in Fig. 4).

6.2 Specification

Specifications of smart contracts are defined under scenarios. For example there are some basic requirements for the aucSC contract:

1. Eventually there is a bidder win (liveness property);
2. The winner is always the bidder with the highest bid (safety property).

The requirements demand that after the auction is ended, 1) there is a winner and the winner must be one of the bidders, and 2) the winner must be the bidder with highest bid. Consider above mentioned scenario. There are three bidders. One of them will win. That is, $hBidder = B$, C or D. The bidder D pays the highest bid. When the value of the *order* reaches the maximum, 3 in this scenario, all processes are executed and the scenario finishes. We use a keyword *End* to indicate that the involved scenario is ended. Therefore, the requirements formulated under aucSNR are expressed by the following CTL formulas:

1. aucPr1: $AF((End)\&((hBidder = B)|(hBidder = C)|(hBidder = D)));$
2. aucPr2: $AG(!(End)|(hBidder = D)).$

Where $!p|q$ is equivalent to *p implies q*. The specifications written in CTL formulas are used in the SPEC part of the VVM, shown in Fig. 4 (Lines 25-26).

7 Verification and Counter Example Extraction

The generated VVM is then verified by VERDS. If a property is satisfied, we can guarantee that the contract meets the corresponding requirement under the given scenario. Otherwise, VERDS may return a CEX file that records the trace of a counter example. A trace is a sequence of states generated by an execution of the contract. The procedure of counter example extraction starts from the trace. By the values of variables in states, we find the corresponding transitions in the VVM. Then according to Table. 1, a reverted procedure of contract modeling is used to map the transitions to the Solidity contract. As such we have a counter example on the Solidity level. Below we illustrate the procedure by an example.

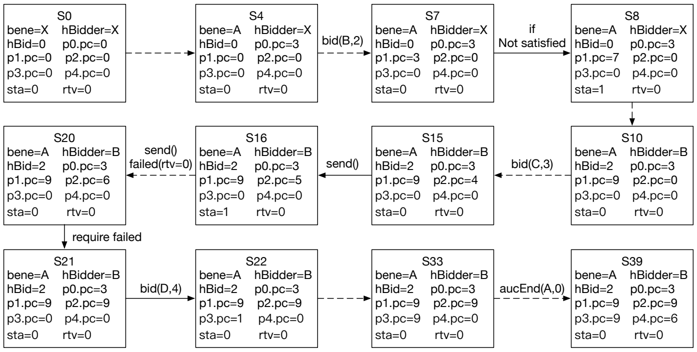


Fig. 6. A Trace Returned by VERDS.

By manually checking the aucSC contract (Fig. 1), it seems that it meets the second requirement (aucPr2). The result of the verification under scenario aucSNR, however, shows otherwise. That is, aucSC does not always choose the bidder with the highest bid as a winner. The trace returned by VERDS is shown in Fig. 6. The trace shows that when the three bidders bid sequentially in the order of bidder B, bidder C and then bidder D, the winner is bidder B. The corresponding order of calls on aucSC is shown in Fig. 7. Statements with process identifiers, i.e., p0, ..., p3 and p4, on the left are executed but the gray statements are not in a trace for a counter example.

```
p0  constructor(address _beneficiary)
     public {
p0      bene = _beneficiary;
p0  }
p1  call bid() public payable{
p1      require(!ended);
p1      require(msg.value > hBid);
p1      if (hBid!=0 ) {
            require(hBidder.send(hBid));
p1      }
p1      hBidder = msg.sender;
p1      hBid = msg.value;
p1  }
p2  call bid() public payable{
p2      require(!ended);
p2      require(msg.value > hBid);
p2      if (hBid!=0 ) {
p2          require(hBidder.send(hBid));
        }
        hBidder = msg.sender;
        hBid = msg.value;
p2  }
p3  call bid() public payable{
p3      require(!ended);
p3      require(msg.value > hBid);
p3      if (hBid!=0 ) {
p3          require(hBidder.send(hBid));
        }
        hBidder = msg.sender;
        hBid = msg.value;
p3  }
p4  call aucEnd() public{
p4      require(!ended);
p4      ended = true;
p4      bene.transfer(hBid);
p4  }
```

Fig. 7. A Counter Example of aucSC.

When bidder B bids, $p1 : \mathit{bid}(B, 2)$ is called. At state S7, where $p1.pc = 3$ and $\mathit{hBid} = 0$, the condition of transition at Line 45 of aucVVM (Fig. 4) is satisfied, then $p1.pc$ is changed to 7, corresponding to the transition of Line 46 of aucVVM. By the information of the transitions in aucVVM, we then can find that after Line 13 of aucSC (Fig. 1), the execution go to Line 16, skipping Line 14, where is the gray line in Fig. 7.

The execution trace of $\mathit{bid}(C, 3)$ and $\mathit{bid}(D, 4)$ from the states shown in Fig. 6 to the execution path shown in Fig. 7 can be similarly explained.

In the counter example, after the first bidder, say, with bid 2, other higher biddings 3 and 4 may be aborted because the *send* operation fails. Recall that the return value of *send()* depends on what happens outside the contract and may return *true* or *false* nondeterministically. The subtle potential error is revealed by the counter example.

8 Case Studies and Experiments

We implemented **mcVer** and applied it to several case studies, including some reported contracts with typical security vulnerabilities [6] [2] [3] and an access control smart contract [51]. Our experiments are carried out on a machine with 3.10GHz CPU, 512GB RAM. The dataset is available on [5].

8.1 Security Vulnerabilities Checking

We show in this section some examples to illustrate how vulnerabilities such as Transaction-Ordering Dependency (TOD) and Reentrancy Vulnerability are revealed in **mcVer** .

Transaction-Ordering Dependency Assume two transactions invoke a contract at the same time. If a final state depends on the order of these transactions, then a TOD vulnerability may exist. An attacker can enforce a specific execution order to make profit [27].

```
pragma solidity ^0.4.16;
contract EthTxOrderDependenceMinimal{
    address public owner;
    bool public claimed;
    uint public reward;
    constructor() public{
        owner = msg.sender;
    }
    function setReward() public payable{
        require(!claimed);
        require(msg.sender == owner);
        owner.transfer(reward);
        reward = msg.value;
    }
    function claimReward(uint256
         submission) {
        require(!claimed);
        require(submission<10);
        msg.sender.transfer(reward);
        claimed = true;
    }
}
```

Fig. 8. A Contract with TOD [6], todSC

An example of a contract with TOD vulnerability [6] is shown in Fig. 8. Suppose the reward is first set to 3 by the owner A after the contract is deployed. Then user B sees the reward and tries to claim, and at the same time the owner resets the reward to 1. These two transactions may be executed in any order. The contract is first translated into a VVM with a module *setReward(msg_sender, msg_value)* to model function *setReward()*, a module *claimReward(msg_sender, msg_value, submission)* to model function *claimReward(uint submission)* and a module to model the constructor. The aforementioned scenario is configured as:

$$\begin{array}{l} p0 : EthTxOrderDependenceMinimal(A, 0) \\ p1 : setReward(A, 3) \\ p2 : setReward(A, 1) | p3 : claimReward(B, 0, 3) \end{array}$$

Suppose user B has a balance 5 before a transaction. After that, he should expect a balance to be 8, since the reward is 3 when he submits the transaction. Therefore, the property is denoted as follows.

$$AG(!(End)|(balance[B] = 8))$$

VERDS exhaustively explores the state space of the model and finds violation of the property. Therefore, a trace where the user's balance is only 6 at the end of the scenario is returned by VERDS as a counter-example. The trace shows that *setReward()* is executed before *claimReward()* and the reward is set to 1. Then, should B always have balance of 6 after the scenario? Verification of the following property shows otherwise, because *claimReward()* may also be executed first.

$$AG(!(End)|(balance[B] = 6))$$

The results show that a different execution order may lead to different balance of B. The TOD vulnerability is detected.

Reentrancy Vulnerability If a function of a contract is called again before its previous invocations complete execution and the next call leads to inconsistency of balances of related accounts, a Reentrancy Vulnerability exists in the contract. The reason of TheDAO event is exactly that the Reentrancy Vulnerability in the DAO was exploited by attackers. A contract with Reentrancy Vulnerability is shown in Fig. 9. When function $withDraw()$ is called, the statement $msg.sender.call.value(amount)()$ in Line 8 will transfer some money to another contract and trigger its fallback function. When there is a callback to $withDraw()$ in the fallback function, a reentrancy is formed and an attacker may withdraw more amount of money than his balance in the contract.

```
 1  pragma solidity ^0.4.19;
 2  contract Victim{
 3    mapping(address => uint) public
          userBalance;
 4    uint public amount=0;
 5    function withDraw(){
 6      uint amount = userBalance[msg.sender];
 7      if(amount>0){
 8          msg.sender.call.value(amount)();
 9          userBalance[msg.sender] = 0;
10      }
11    }
12    function receiveEther() payable{
13      if(msg.value>0){
14          userBalance[msg.sender] += msg.
                value;
15      }
16    }
17  }
```

Fig. 9. Contract with Reentrancy [3], reSC

The contract is translated into a VVM with a set of modules. We set the number of callbacks to 1, because if there is a re-entry it will be found in one callback. An extra module $fallback()$ is used to model $fallback$ function in the external contract which is triggered automatically by $withDraw()$, and a module $withDraw\_back()$ is used to model the re-invoked $withDraw()$ in $fallback$ function. Suppose a user B first deposits 2 *wei* (a unit of Ether) into the contract by invoking *receiveEther()* and then withdraws his money. The scenario is configured as follows.

$$\begin{array}{l} p0 : receiveEther(B, 2) \\ p1 : withDraw(B, 0) \end{array}$$

Suppose the initial balance of the user is 5, the required property is that his balance after the execution of the scenario is still 5. This property is formulated as follows.

$$AG(!(End)|(balance[B] = 5))$$

Checking this property with the above scenario, **mcVer** returns with *false*, meaning that the balance of the user is inconsistent before and after the scenario. The Re-entrancy Vulnerability is then detected.

Table 2. Checking Time for Security Vulnerabilities.

Contract	Vul.	Bound		Execution Time (second)			
		MaxD	MaxI	Modeling	Veri.	Counter Ex.	Total
todSC	TOD	2	8	0.071	3.41	0.003	3.484
reSC	Reentrancy	2	8	0.051	4.099	0.002	4.152

Experimental Results Table 2 shows the time required for **mcVer** to detect the vulnerabilities contained in the above four cases. The bound of number of addresses, MaxD, and the bound of integer, MaxI, are shown in the third colomn. The procedures of model extraction and counter example generation are very fast, and the times required for verification of them are within several seconds.

8.2 Access Control Contract

The previously discussed properties related to security are all safety properties. In this section we study an access control contract [51] to show that **mcVer** can also deal with liveness properties.

Problem Description The ubiquitous interconnection of physical objects has significantly accelerated data collection, aggregation, and sharing, making Internet of Things (IoT) one of the most basic architectures for applications in the smart health-care, smart transportation, and home automation domains. However, such interconnection may also bring serious security problems to IoT systems. If a system does not have secure access control, through intrusion into the system, unauthorized entities (attackers) can illegally access existing IoT devices by simply deploying their own resources. Therefore, the access control issue of the IoT has received extensive attention from academia and industry. The access control system should satisfy the following four requirements:

PR1. Regardless of whether a user has rights or not, the system should return a result;
PR2. Users who have no right to access can't get access rights;
PR3. Users with access rights can always obtain access rights;
PR4. Only specific users (such as administrators) can modify users' access rights.

```
contract AccessControlMethod{
 address public owner;
 address public subject;
 address public object;
 mapping(bytes32=>PolicyItem) policies;
 mapping (bytes32 => BehaviorItem)
      behaviors;

 constructor (address _subject) public{}
 function policyAdd(bytes32 _action, bool
      _permission, uint minInterval, uint
      _threshold) public{}
 function policyUpdate(bytes32 _action,
      bool _newPermission) public{}
 function accessControl(bytes32 _action,
      uint _time) public{uint err;...}
}
```

Fig. 10. Access Control Contract [51].

We outline the contract and show it in Fig. 10. There are five global variables, among which *owner*, *subject* and *object* indicate the address who deployed the contract, the address of the accessing user and IoT device bound to the contract, respectively; the variables *policies* with mapping type are used to record the access policies. There are also three main functions in this contract. Function $AcessControlMethod()$ is the constructor, initializing *owner*, *subject* and *object*. Function *policyAdd*() is used to add new access policy. Function *policyUpdate*() is used to update the access permissions in an access policy. Function *accessControl*() is used to get access rights. There is a local variable *err* in *accessControl*() representing whether a user can get access right. Different values of *err* represent different return results as shown in the following list.

1. $err = 0$ means right is granted;
2. $err = 1$ means punishment isn't ended and right is not granted;

3. $err = 2$ means user has no permission and right is not granted;
4. $err = 3$ means that although the user has the permission but he will be punished and right isn't granted, because he visits too frequently;
5. $err = 4$ means user has no permission and visit too frequently in the minimal interval, will be punished, right isn't granted;
6. $err = 5$ means the device that the user wants to visit is not current device and right isn't granted.

Modeling For the verification of aforementioned four basic properties, we design three different scenarios. The first scenario, denoted by SNR1, is defined as:

$$\begin{array}{l} p0 : constructor(A, 0, B) \\ p1 : policyAdd(A, 0, R, Y, 2, 2) \\ p2 : policyUpdate(A, 0, R, N) | p3 : accessControl(B, 0, R, 2) \end{array}$$

In this scenario, administrator A deploys a contract for user B. He then sends a transaction calling *policyAdd* to add a new policy which allows B to read (R) the data on resource with identifier 3. This policy allows the user to visit the resource twice in interval 2 units of time. Then B tries to get access right and A tries to withdraw the permission for user B at the same time. Scenario SNR1 can be used to check properties PR1 and PR3, which are concretized and specified respectively by SNR1.pr1 and SNR1.pr2, shown as follows.

SNR1.pr1: $AF((End)\&(p3.err = 0|p3.err = 1|p3.err = 2|p3.err = 3|p3.err = 4|p3.err = 5))$
SNR1.pr2: $AG(!(End)|(p3.err = 0))$

The second scenario, SNR2, is defined as:

$$\begin{array}{l} p0 : constructor(A, 0, B) \\ p1 : policyAdd(A, 0, R, Y, 2, 2) \\ p2 : policyUpdate(A, 0, R, N) \\ p3 : accessControl(B, 0, R, 2) \end{array}$$

After *policyAdd* finishes, *policyUpdate* withdraws the permission of user B. The B tries to access the resource. Scenario SNR2 can be used to check properties PR1 and PR2, which are concretized and specified respectively by SNR2.pr1 and SNR2.pr2, shown as follows.

SNR2.pr1: $AF((End)\&(p3.err = 0|p3.err = 1|p3.err = 2|p3.err = 3|p3.err = 4|p3.err = 5))$
SNR2.pr2: $AG(!(End)|(p3.err = 2))$

The last scenario, SNR3, describes the situation that a user B with no permission tries to update his own permission and then tries to get access right:

$$\begin{array}{l} p0 : constructor(A, 0, B) \\ p1 : policyAdd(A, 0, R, N, 2, 2) \\ p2 : policyUpdate(B, 0, R, Y) \\ p3 : accessControl(B, 0, R, 2) \end{array}$$

Scenario SNR3 can be used to check properties PR1 and PR4, which are concretized and specified respectively by SNR3.pr1 and SNR3.pr2, shown as follows.

SNR3.pr1: $AF((End)\&(p3.err = 0|p3.err = 1|p3.err = 2|p3.err = 3|p3.err = 4|p3.err = 5))$
SNR3.pr2: $AG(!(End)|(p3.err = 2))$

The changes of permission can only be checked by the returned error code of *accessControl*(), so in this scenarios *p*3 is added and the formula SNR3.pr2 is the same as SNR2.pr2.

Table 3. Verification Results for Access Control Contract.

Requirements	Properties	Verification Results		Time(second)
PR1	SNR1.pr1	True	True	17.155
	SNR2.pr1	True		18.977
	SNR3.pr1	True		22.329
PR2	SNR2.pr2	True	True	17.367
PR3	SNR1.pr2	**False**	**False**	21.201
PR4	SNR3.pr2	**False**	**False**	19.019

Verification After the contract, the scenarios and the property specification are translated into VVM, we verified the VVM using VERDS. The results are shown in Table 3. The first column list the names of requirements, the second column lists the related CTL formula, the third and forth columns are verification results, and the last column shows verification time for each property. The result for PR1 is true only when the verification results of SNR1.pr1, SNR2.pr1 and SNR3.pr1 are all true. From the results, we know that the contract doesn't meet PR3 and PR4. And the counter-example returned when verifying SNR3.pr2 shows that user B, who is not an administrator, successfully changes his own permission by calling function *policyUpdate*(), and gets the access right which violates PR4. All six properties are verified in about two minutes.

9 Conclusion and Future Work

In this paper, the model checking based framework, **mcVer**, has been proposed to support the verification of smart contracts written in Solidity. **mcVer** is able to verify a variety of properties of smart contracts. For the properties that are not satisfied, **mcVer** produces a counter example by showing a sequence of statements in the original Solidity program as a hint of where a faulty statement may be located. We have implemented **mcVer** and applied it to automatically checking various types of security vulnerabilities and properties of an access control smart contract. The results show that the proposed framework is flexible and efficient and can facilitate software development in the blockchain domain in terms of the diversity of detecting software breaches.

The limitations of **mcVer** framework come from two folds. The first is from the model checking technique itself, which can only deal with bounded systems and may suffer from state explosion issue when the model scales up. The second is that we have to configure the scenarios with particular values of parameters, which confines the space to be explored and limits the ability of **mcVer**. In the future, we will study the property-based contract modeling technique to reduce the size of the model to be verified and consider a better way to model the environment and user behaviors to broaden the scope of verification. Also, we will consider the impact of gas limitation to the behavior of contracts.

References

1. https://solidity-cn.readthedocs.io/zh/develop/solidity-by-example.html
2. https://bitcoinist.com/smart-contract-bug-disable-icon-icx-transfers/
3. https://blog.csdn.net/programmer_cjc/article/details/85987234
4. Analysis of the dao exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
5. Dataset for mcver, https://gitee.com/fmpa/dataset-for-mcVer
6. Transaction order dependence. https://swcregistry.io/docs/swc-114
7. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: SAFEVM: a safety verifier for ethereum smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 386–389 (2019)
8. Alqahtani, S., He, X., Gamble, R., Mauricio, P.: Formal verification of functional requirements for smart contract compositions in supply chain management systems. In: Proceedings of the 53rd Hawaii International Conference on System Sciences (2020)
9. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Maffei, M., Ryan, M. (eds.) Principles of Security and Trust (POST). pp. 164–186. Springer (2017)
10. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: International Conference on Computer Aided Verification. pp. 184–190. Springer (2011)
11. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96 (2016)
12. Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 454–469 (2020)
13. Chen, R., Zhang, W.: Checking multi-agent systems against temporal-epistemic specifications. In: the 24th International Conference on Engineering of Complex Computer Systems. pp. 21–30. IEEE (2019)
14. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**(4), 410–425 (2000)
15. Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR) **28**(4), 626–643 (1996)
16. Costello, K.: Gartner predicts 90% of current enterprise blockchain platform implementations will require replacement by 2021 (2019), https://www.gartner.com/en/newsroom/press-releases/2019-07-03-gartner-predicts-90–of-current-enterprise-blockchain
17. Edmund M. Clarke, J., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking, Second Edition. MIT Press (2018)
18. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 8–15. IEEE (2019)
19. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European symposium on programming. pp. 125–128. Springer (2013)

20. Frank, J., Aschermann, C., Holz, T.: ETHBMC: A bounded model checker for smart contracts. In: 29th USENIX Security Symposium. pp. 2757–2774 (2020)
21. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 557–560 (2020)
22. Hegedűs, P.: Towards analyzing the complexity landscape of solidity based ethereum smart contracts. Technologies **7**(1), 6 (2019)
23. Hirai, Y.: Formal verification of deed contract in ethereum name service. November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf (2016)
24. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography and Data Security. vol. LNCS 10323, pp. 520–535. Springer (2017)
25. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: Analyzing safety of smart contracts. In: Network and Distributed Systems Security (NDSS) Symposium. pp. 1–12 (2018)
26. Liu, Y., Li, Y., Lin, S.W., Zhao, R.: Towards automated verification of smart contract fairness. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 666–677 (2020)
27. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 254–269 (2016)
28. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, New York
29. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: International Conference on Financial Cryptography and Data Security. pp. 446–465. Springer (2019)
30. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1186–1189 (2019)
31. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: the 19th ACM SIGPLAN international conference on Functional programming. pp. 175–188 (2014)
32. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Manubot (2019)
33. Nehaï, Z., Piriou, P., Daumas, F.: Model-checking of smart contracts. In: IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 980–987
34. Nehai, Z., Bobot, F.: Deductive proof of ethereum smart contracts using why3. arXiv preprint arXiv:1904.11281 (2019)
35. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
36. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1661–1677 (2020)
37. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Financial Cryptography and Data Security. vol. LNCS 10323, pp. 478–493. Springer (2017)

38. So, S., Lee, M., Park, J., Lee, H., Oh, H.: Verismart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1678–1694 (2020)
39. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270 (2016)
40. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997), https://firstmonday.org/ojs/index.php/fm/article/view/548
41. Thomson, I.: Parity: The bug that put $169m of ethereum on ice? yeah, it was on the todo list for months (2017), https://www.theregister.com/2017/11/16/parity_flaw_not_fixed/
42. Thomson, I.: Mythril classic: Security analysis tool for ethereum smart contracts (2018), https://github.com/ConsenSys/mythril
43. 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 (2018)
44. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Computing Surveys **54**(7), 1–38 (2022)
45. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82 (2018)
46. Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in ethereum smart contracts. Proc. ACM Program. Lang. **3**, Article 189 (2019)
47. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
48. Wüstholz, V., Christakis, M.: Harvey: A greybox fuzzer for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1398–1409 (2020)
49. Zeng, N., Zhang, W.: An executable semantics of systemc transaction level models and its applications with verds. In: the 19th International Conference on Engineering of Complex Computer Systems. pp. 198–201 (2014)
50. Zhang, W.: Verds: Verification of hierarchical discrete systems by symbolic techniques. Manuscript, available at webpage http://lcs. ios. ac. cn/~zwh/verds (2013)
51. Zhang, Y., Kasahara, S., Shen, Y., Jiang, X., Wan, J.: Smart contract-based access control for the internet of things. IEEE Internet of Things Journal **6**(2), 1594–1605 (2018)