VeriSolid: Correct-by-Design Smart Contracts for Ethereum

Anastasia Mavridou[1], Aron Laszka[2](✉), Emmanouela Stachtiari[3], and Abhishek Dubey[1]

[1] Vanderbilt University, Nashville, USA
[2] University of Houston, Houston, USA
alaszka@uh.edu
[3] Aristotle University of Thessaloniki, Thessaloniki, Greece

Abstract. The adoption of blockchain based distributed ledgers is growing fast due to their ability to provide reliability, integrity, and auditability without trusted entities. One of the key capabilities of these emerging platforms is the ability to create self-enforcing smart contracts. However, the development of smart contracts has proven to be error-prone in practice, and as a result, contracts deployed on public platforms are often riddled with security vulnerabilities. This issue is exacerbated by the design of these platforms, which forbids updating contract code and rolling back malicious transactions. In light of this, it is crucial to ensure that a smart contract is secure before deploying it and trusting it with significant amounts of cryptocurrency. To this end, we introduce the *VeriSolid* framework for the formal verification of contracts that are specified using a transition-system based model with rigorous operational semantics. Our model-based approach allows developers to reason about and verify contract behavior at a high level of abstraction. VeriSolid allows the generation of Solidity code from the verified models, which enables the *correct-by-design* development of smart contracts.

1 Introduction

The adoption of blockchain based platforms is rising rapidly. Their popularity is explained by their ability to maintain a *distributed public ledger*, providing reliability, integrity, and auditability *without a trusted entity*. Early blockchain platforms, e.g., Bitcoin, focused solely on creating cryptocurrencies and payment systems. However, more recent platforms, e.g., Ethereum, also act as distributed computing platforms [43,45] and enable the creation of *smart contracts*, i.e., software code that runs on the platform and automatically executes and enforces the terms of a contract [10]. Since smart contracts can perform any computation[1], they allow the development of decentralized applications, whose execution is safeguarded by the security properties of the underlying platform. Due to their

[1] While the virtual machine executing a contract may be Turing-complete, the amount of computation that it can perform is actually limited in practice.

I. Goldberg and T. Moore (Eds.): FC 2019, LNCS 11598, pp. 446–465, 2019.
https://doi.org/10.1007/978-3-030-32101-7_27

unique advantages, blockchain based platforms are envisioned to have a wide range of applications, ranging from financial to the Internet-of-Things [9].

However, the trustworthiness of the platform guarantees only that a smart contract is executed correctly, not that the code of the contract is correct. In fact, a large number of contracts deployed in practice suffer from software vulnerabilities, which are often introduced due to the semantic gap between the assumptions that contract writers make about the underlying execution semantics and the actual semantics of smart contracts [25]. A recent automated analysis of 19,336 contracts deployed on the public Ethereum blockchain found that 8,333 contracts suffered from at least one security issue [25]. While not all of these issues lead to security vulnerabilities, many of them enable stealing digital assets, such as cryptocurrencies. Smart-contract vulnerabilities have resulted in serious security incidents, such as the "DAO attack," in which $50 million worth of cryptocurrency was stolen [14], and the 2017 hack of the multisignature Parity Wallet library [32], which lost $280 million worth of cryptocurrency.

The risk posed by smart-contract vulnerabilities is exacerbated by the typical design of blockchain based platforms, which does not allow the code of a contract to be updated (e.g., to fix a vulnerability) or a malicious transaction to be reverted. Developers may circumvent the immutability of code by separating the "backend" code of a contract into a library contract that is referenced and used by a "frontend" contract, and updating the backend code by deploying a new instance of the library and updating the reference held by the frontend. However, the mutability of contract terms introduces security and trust issues (e.g., there might be no guarantee that a mutable contract will enforce any of its original terms). In extreme circumstances, it is also possible to revert a transaction by performing a hard fork of the blockchain. However, a hard fork requires consensus among the stakeholders of the entire platform, undermines the trustworthiness of the entire platform, and may introduce security issues (e.g., replay attacks between the original and forked chains).

In light of this, it is crucial to ensure that a smart contract is secure before deploying it and trusting it with significant amounts of cryptocurrency. Three main approaches have been considered for securing smart contracts, including secure programming practices and patterns (e.g., Checks–Effects–Interactions pattern [40]), automated vulnerability-discovery tools (e.g., Oyente [25,42]), and formal verification of correctness (e.g., [17,21]). Following secure programming practices and using common patterns can decrease the occurrence of vulnerabilities. However, their effectiveness is limited for multiple reasons. First, they rely on a programmer following and implementing them, which is error prone due to human nature. Second, they can prevent a set of typical vulnerabilities, but they are not effective against vulnerabilities that are atypical or belong to types which have not been identified yet. Third, they cannot provide formal security and safety guarantees. Similarly, automated vulnerability-discovery tools consider generic properties that usually do not capture contract-specific requirements and thus, are effective in detecting typical errors but ineffective in detecting atypical vulnerabilities. These tools typically require security properties and patterns to be specified at a low level (usually bytecode) by security experts. Additionally, automated vulnerability-discovery tools are not precise; they often produce false positives.

On the contrary, formal verification tools are based on formal operational semantics and provide strong verification guarantees. They enable the formal specification and verification of properties and can detect both typical and atypical vulnerabilities that could lead to the violation of some security property. However, these tools are harder to automate.

Our approach falls in the category of formal verification tools, but it also provides an end-to-end design framework, which combined with a code generator, allows the *correctness-by-design* development of Ethereum smart contracts. We focus on providing usable tools for helping developers to eliminate errors early at design time by raising the abstraction level and employing graphical representations. Our approach does not produce false positives for safety properties and deadlock-freedom.

In principle, a contract vulnerability is a programming error that enables an attacker to use a contract in a way that was not intended by the developer. To detect vulnerabilities that do not fall into common types, developers must specify the intended behavior of a contract. Our framework enables developers to specify intended behavior in the form of liveness, deadlock-freedom, and safety properties, which capture important security concerns and vulnerabilities. One of the key advantages of our model-based verification approach is that it allows developers to specify desired properties with respect to high-level models instead of, e.g., bytecode. Our tool can then automatically verify whether the behavior of the contract satisfies these properties. If a contract does not satisfy some of these properties, our tool notifies the developers, explaining the execution sequence that leads to the property violation. The sequence can help the developer to identify and correct the design errors that lead to the erroneous behavior. Since the verification output provides guarantees to the developer regarding the actual execution semantics of the contract, it helps eliminating the semantic gap. Additionally, our verification and code generation approach fits smart contracts well because contract code cannot be updated after deployment. Thus, code generation needs to be performed only once before deployment.

Contributions. We build on the *FSolidM* [27,28] framework, which provides a graphical editor for specifying Ethereum smart contracts as transitions systems and a *Solidity* code generator.[2] We present the *VeriSolid* framework, which introduces *formal verification capabilities*, thereby providing an approach for correct-by-design development of smart contracts. Our contributions are:

- We extend the syntax of FSolidM models (Definition 1), provide formal operational semantics (FSolidM has no formal operational semantics) for our model (Sect. 3.3) and for supported Solidity statements ([29, Appendix A.3]), and extend the Solidity code generator ([29, Appendix E]).

[2] Solidity is the high-level language for developing Ethereum contracts. Solidity code can be compiled into bytecode, which can be executed on the Ethereum platform.

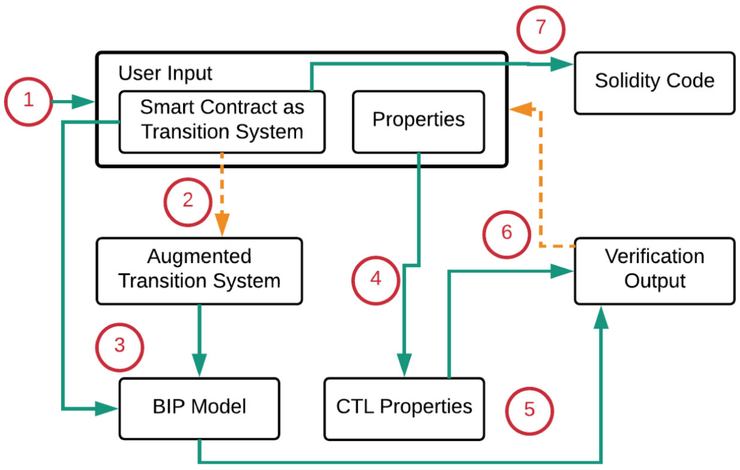

Fig. 1. Design and verification workflow.

- We design and implement developer-friendly natural-language like templates for specifying safety and liveness properties (Sect. 3.4).
- The developer input of VeriSolid is a transition system, in which each transition action is specified using Solidity code. We provide an automatic transformation from the initial system into an augmented transition system, which extends the initial system with the control flow of the Solidity action of each transition (Sect. 4). We prove that the initial and augmented transition systems are observationally equivalent (Sect. 4.1); thus, the verified properties of the augmented model are also guaranteed in the initial model.
- We use an overapproximation approach for the meaningful and efficient verification of smart-contract models (Sect. 5). We integrate verification tools (i.e., nuXmv and BIP) and present verification results.

2 VeriSolid: Design and Verification WorkFlow

VeriSolid is an open-source[3] and web-based framework that is built on top of WebGME [26] and FSolidM [27,28]. VeriSolid allows the collaborative development of Ethereum contracts with built-in version control, which enables branching, merging, and history viewing. Figure 1 shows the steps of the VeriSolid design flow. Mandatory steps are represented by solid arrows, while optional steps are represented by dashed arrows. In step ①, the developer input is given, which consists of:

- A contract specification containing (1) a graphically specified transition system and (2) variable declarations, actions, and guards specified in Solidity.

[3] https://github.com/anmavrid/smart-contracts.

– A list of properties to be verified, which can be expressed using predefined natural-language like templates.

The verification loop starts at the next step. Optionally, step ② is automatically executed if the verification of the specified properties requires the generation of an augmented contract model[4]. Next, in step ③, the Behavior-Interaction-Priority (BIP) model of the contract (augmented or not) is automatically generated. Similarly, in step ④, the specified properties are automatically translated to Computational Tree Logic (CTL). The model can then be verified for deadlock freedom or other properties using tools from the BIP tool-chain [5] or nuXmv [7] (step ⑤). If the required properties are not satisfied by the model (depending on the output of the verification tools), the specification can be refined by the developer (step ⑥) and analyzed anew. Finally, when the developers are satisfied with the design, i.e., all specified properties are satisfied, the equivalent Solidity code of the contract is automatically generated in step ⑦. The following sections describe the steps from Fig. 1 in detail. Due to space limitations, we present the Solidity code generation (step ⑦) in [29, Appendix E].

3 Developer Input: Transition Systems and Properties

3.1 Smart Contracts as Transition Systems

To illustrate how to represent smart contracts as transition systems, we use the *Blind Auction* example from prior work [27], which is based on an example from the Solidity documentation [38].

In a blind auction, each bidder first makes a deposit and submits a blinded bid, which is a hash of its actual bid, and then reveals its actual bid after all bidders have committed to their bids. After revealing, each bid is considered valid if it is higher than the accompanying deposit, and the bidder with the highest valid bid is declared winner. A blind auction contract has four main states:

1. `AcceptingBlindedBids`: bidders submit blinded bids and make deposits;
2. `RevealingBids`: bidders reveal their actual bids by submitting them to the contract, and the contract checks for each bid that its hash is equal to the blinded bid and that it is less than or equal to the deposit made earlier;
3. `Finished`: winning bidder (i.e., the bidder with the highest valid bid) withdraws the difference between her deposit and her bid; other bidders withdraw their entire deposits;
4. `Canceled`: all bidders withdraw their deposits (without declaring a winner).

This example illustrates that smart contracts have *states* (e.g., `Finished`). Further, contracts provide functions, which allow other entities (e.g., users or contracts) to invoke *actions* and change the states of the contracts. Hence, we can represent a smart contract naturally as a *transition system* [39], which comprises

[4] We give the definition of an augmented smart contract in Sect. 4.

a set of states and a set of transitions between those states. Invoking a transition forces the contract to execute the action of the transition if the *guard* condition of the transition is satisfied. Since such states and transitions have intuitive meanings for developers, representing contracts as transition systems provides an adequate level of abstraction for reasoning about their behavior.

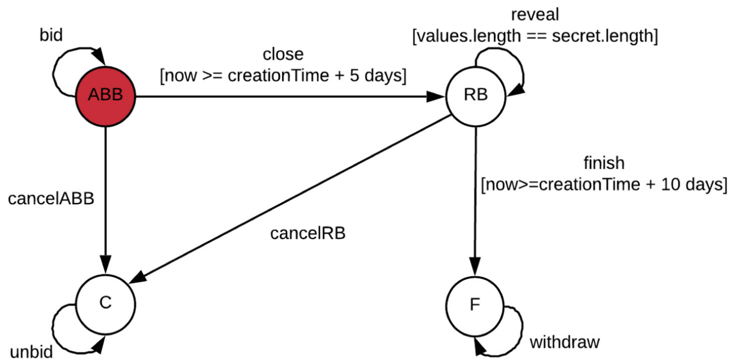
bid
ABB
close
[now >= creationTime + 5 days]
reveal
[values.length == secret.length]
RB
cancelABB
cancelRB
finish
[now>=creationTime + 10 days]
C
unbid
F
withdraw

Fig. 2. Blind auction example as a transition system.

Figure 2 shows the blind auction example in the form of a transition system. For ease of presentation, we abbreviate `AcceptingBlindedBids`, `RevealingBids`, `Finished`, and `Canceled` to `ABB`, `RB`, `F`, and `C`, respectively. The initial state of the transition system is `ABB`. To differentiate between transition names and guards, we use square brackets for the latter. Each transition (e.g., `close`, `withdraw`) corresponds to an action that a user may perform during the auction. For example, a bidding user may execute transition `reveal` in state `RB` to reveal its blinded bid. As another example, a user may execute transition `finish` in state `RB`, which ends the revealing phase and declares the winner, if the guard condition `now >= creationTime + 10 days` is true. A user can submit a blinded bid using transition `bid`, close the bidding phase using transition `close`, and withdraw her deposit (minus her bid if she won) using transitions `unbid` and `withdraw`. Finally, the user who created the auction may cancel it using transitions `cancelABB` and `cancelRB`. For clarity of presentation, we omitted from Fig. 2 the specific actions that the transitions take (e.g., transition `bid` executes—among others—the following statement: `pendingReturns[msg.sender] += msg.value;`).

3.2 Formal Definition of a Smart Contract

We formally define a contract as a transition system. To do that, we consider a subset of Solidity statements, which are described in detail in [29, Appendix A.1]. We chose this subset of Solidity statements because it includes all the essential control structures: loops, selection, and `return` statements. Thus, it is a Turing-complete subset, and can be extended in a straightforward manner to capture all other Solidity statements. Our Solidity code notation is summarized in Table 1.

Table 1. Summary of notation for Solidity code

Symbol	Meaning
$\mathbb{T}$	Set of Solidity types
$\mathbb{I}$	Set of valid Solidity identifiers
$\mathbb{D}$	Set of Solidity event and custom-type definitions
$\mathbb{E}$	Set of Solidity expressions
$\mathbb{C}$	Set of Solidity expressions without side effects
$\mathbb{S}$	Set of supported Solidity statements

Definition 1. *A transition-system* initial smart contract *is a tuple* $(D, S, S_F, s_0, a_0, a_F, V, T)$*, where*

- $D \subset \mathbb{D}$ *is a set of custom event and type definitions;*
- $S \subset \mathbb{I}$ *is a finite set of states;*
- $S_F \subset S$ *is a set of final states;*
- $s_0 \in S$*,* $a_0 \in \mathbb{S}$ *are the initial state and action;*
- $a_F \in \mathbb{S}$ *is the fallback action;*
- $V \subset \mathbb{I} \times \mathbb{T}$ *contract variables (i.e., variable names and types);*
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I}\times\mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ *is a transition relation, where each transition* $\in T$ *includes: transition name* $t^{name} \in \mathbb{I}$*; source state* $t^{from} \in S$*; parameter variables (i.e., arguments)* $t^{input} \subseteq \mathbb{I} \times \mathbb{T}$*; transition guard* $g_t \in \mathbb{C}$*; return type* $t^{output} \in (\mathbb{T} \cup \emptyset)$*; action* $a_t \in \mathbb{S}$*; destination state* $t^{to} \in S$*.*

The initial action a_0 represents the constructor of the smart contract. A contract can have *at most one constructor*. In the case that the initial action a_0 is empty (i.e., there is no constructor), a_0 may be omitted from the transition system. A constructor is graphically represented in VeriSolid as an incoming arrow to the initial state. The fallback action a_F represents the fallback function of the contract. Similar to the constructor, a contract can have *at most one fallback* function. Solidity fallback functions are further discussed in [29, Appendix C.1].

Lack of the Re-entrancy Vulnerability. VeriSolid allows specifying contracts such that the re-entrancy vulnerability is prevented by design. In particular, after a transition begins but before the execution of the transition action, the contract changes its state to a temporary one (see [29, Appendix E]). This prevents re-entrancy since none of the contract functions[5] can be called in this state.

[5] Our framework implements transitions as functions, see [29, Appendix E].

One might question this design decision since re-entrancy is not always harmful. However, we consider that it can pose significant challenges for providing security. First, supporting re-entrancy substantially increases the complexity of verification. Our framework allows the efficient verification—within seconds—of a broad range of properties, which is essential for iterative development. Second, re-entrancy often leads to vulnerabilities since it significantly complicates contract behavior. We believe that prohibiting re-entrancy is a small price to pay for security.

3.3 Smart-Contract Operational Semantics

We define the operational semantics of our transition-system based smart contracts in the form of Structural Operational Semantics (SOS) rules [37]. We let Ψ denote the state of the ledger, which includes account balances, values of state variables in all contracts, number and timestamp of the last block, etc. During the execution of a transition, the execution state $\sigma = \{\Psi, M\}$ also includes the memory and stack state M. To handle return statements and exceptions, we also introduce an execution status, which is E when an exception has been raised, $R[v]$ when a return statement has been executed with value v (i.e., `return` v), and N otherwise. Finally, we let $\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\hat{\sigma}, x), v\rangle$ signify that the evaluation of a Solidity expression Exp in execution state σ yields value v and—as a side effect—changes the execution state to $\hat{\sigma}$ and the execution status to x.[6]

A transition is triggered by providing a transition (i.e., function) $name \in \mathbb{I}$ and a list of parameter values $v_1, v_2, \ldots$. The normal execution of a transition without returning any value, which takes the ledger from state Ψ to Ψ' and the contract from state $s \in S$ to $s' \in S$, is captured by the TRANSITION rule:

$$\text{TRANSITION}\quad \frac{\begin{array}{c} t \in T, name = t^{name}, s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M) \\ \text{Eval}(\sigma, g_t) \rightarrow \langle(\hat{\sigma}, N), \texttt{true}\rangle \\ \langle(\hat{\sigma}, N), a_t\rangle \rightarrow \langle(\hat{\sigma}', N), \cdot\rangle \\ \hat{\sigma}' = (\Psi', M'), s' = t^{to} \end{array}}{\langle(\Psi, s), name\,(v_1, v_2, \ldots)\rangle \rightarrow \langle(\Psi', s', \cdot)\rangle}$$

This rule is applied if there exists a transition t whose name t^{name} is $name$ and whose source state t^{from} is the current contract state s (first line). The execution state σ is initialized by taking the parameter values $Params(t, v_1, v_2, \ldots)$ and the current ledger state Ψ (second line). If the guard condition g_t evaluates $\text{Eval}(\sigma, g_t)$ in the current state σ to `true` (third line), then the action statement a_t of the transition is executed (fourth line), which results in an updated execution state $\hat{\sigma}'$ (see statement rules in [29, Appendix A.3]). Finally, if the resulting execution status is normal N (i.e., no exception was thrown), then the updated ledger state Ψ' and updated contract state s' (fifth line) are made permanent.

We also define SOS rules for all cases of erroneous transition execution (e.g., exception is raised during guard evaluation, transition is reverted, etc.)

[6] Note that the correctness of our transformations does not depend on the exact semantics of Eval.

and for returning values. Due to space limitations, we include these rules in [29, Appendix A.2]. We also define SOS rules for supported statements in [29, Appendix A.3].

3.4 Safety, Liveness, and Deadlock Freedom

A VeriSolid model is automatically verified for deadlock freedom. A developer may additionally verify safety and liveness properties. To facilitate the specification of properties, VeriSolid offers a set of predefined natural-language like templates, which correspond to properties in CTL. Alternatively, properties can be specified directly in CTL. Let us go through some of these predefined templates. Due to space limitations, the full template list, as well as the CTL property correspondence is provided in [29, Appendix B].

```
uint amount = pendingReturns[msg.sender];
if (amount > 0) {
  if (msg.sender!= highestBidder)
    msg.sender.transfer(amount);
  else
    msg.sender.transfer(amount - highestBid);
  pendingReturns[msg.sender] = 0;
}
```

Fig. 3. Action of transition `withdraw` in Blind Auction, specified using Solidity.

$\langle$***Transitions*** $\cup$ ***Statements***$\rangle$ cannot happen after $\langle$***Transitions*** $\cup$ ***Statements***$\rangle$.

The above template expresses a safety property type. **Transitions** is a subset of the transitions of the model (i.e., **Transitions** $\subseteq T$). A statement from **Statements** is a specific inner statement from the action of a specific transition (i.e., **Statements** $\subseteq T \times \mathbb{S}$). For instance, we can specify the following safety properties for the Blind Auction example:

- **bid** cannot happen after **close**.
- **cancelABB; cancelRB** cannot happen after **finish**,

where **cancelABB; cancelRB** means **cancelABB** $\cup$ **cancelRB**.

If $\langle$***Transitions*** $\cup$ ***Statements***$\rangle$ happens, $\langle$***Transitions*** $\cup$ ***Statements***$\rangle$ can happen only after $\langle$***Transitions*** $\cup$ ***Statements***$\rangle$ happens.

The above template expresses a safety property type. A typical vulnerability is that currency withdrawal functions, e.g., `transfer`, allow an attacker to withdraw currency again before updating her balance (similar to "The DAO" attack). To check this vulnerability type for the Blind Auction example, we can specify the following property. The statements in the action of transition `withdraw` are shown in Fig. 3.

– if **withdraw.**`msg.sender.transfer(amount);` happens, **withdraw.**`msg.sender.transfer(amount);` can happen only after **withdraw.**`pendingReturns[msg.sender]=0;` happens.

As shown in the example above, a statement is written in the following form: ***Transition.Statement*** to refer to a statement of a specific transition. If there are multiple identical statements in the same transition, then all of them are checked for the same property. To verify properties with statements, we need to transform the input model into an augmented model, as presented in Sect. 4.

⟨***Transitions*** ∪ ***Statements***⟩ will eventually happen after ⟨***Transitions*** ∪ ***Statements***⟩.

Finally, the above template expresses a liveness property type. For instance, with this template we can write the following liveness property for the Blind Auction example to check the Denial-of-Service vulnerability ([29, Appendix C.2]):

– **withdraw.**`pendingReturns[msg.sender]=0;` will eventually happen after **withdraw.**`msg.sender.transfer(amount);`.

4 Augmented Transition System Transformation

To verify a model with Solidity actions, we transform it to a functionally equivalent model that can be input into our verification tools. We perform two transformations: First, we replace the initial action a_0 and the fallback action a_F with transitions. Second, we replace transitions that have complex statements as actions with a series of transitions that have only simple statements (i.e., variable declaration and expression statements). After these two transformations, the entire behavior of the contract is captured using only transitions. The transformation algorithms are discussed in detail in [29, Appendices D.1 and D.2]. The input of the transformation is a smart contract defined as a transition system (see Definition 1). The output of the transformation is an *augmented smart contract*:

Definition 2. *An* augmented contract *is a tuple* (D, S, S_F, s_0, V, T)*, where*

- $D \subset \mathbb{D}$ *is a set of custom event and type definitions;*
- $S \subset \mathbb{I}$ *is a finite set of states;*
- $S_F \subset S$ *is a set of final states;*
- $s_0 \in S$*, is the initial state;*
- $V \subset \mathbb{I} \times \mathbb{T}$ *contract variables (i.e., variable names and types);*
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I}\times\mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ *is a transition relation (i.e., transition name, source state, parameter variables, guard, return type, action, and destination state).*

Figure 4 shows the augmented `withdraw` transition of the Blind Auction model. We present the complete augmented model in [29, Appendix F]. The action of the original `withdraw` transition is shown by Fig. 3. Notice the added state `withdraw`, which avoids re-entrancy by design, as explained in Sect. 3.2.

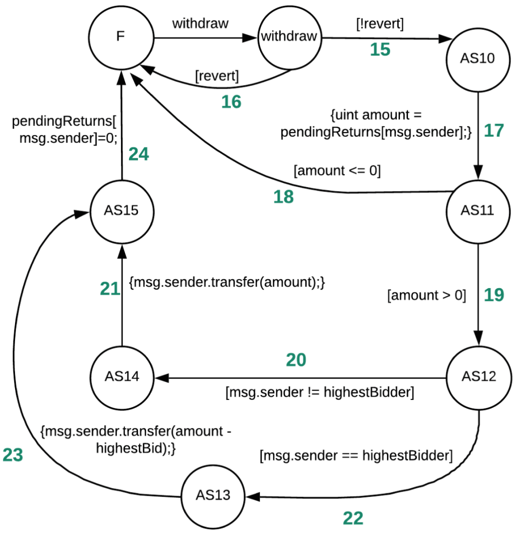
F
withdraw
withdraw
[!revert]
15
AS10
[revert]
16
{uint amount = pendingReturns[msg.sender];}
17
pendingReturns[msg.sender]=0;
24
[amount <= 0]
18
AS15
AS11
21
{msg.sender.transfer(amount);}
[amount > 0]
19
20
AS14
AS12
[msg.sender != highestBidder]
{msg.sender.transfer(amount - highestBid);}
23
[msg.sender == highestBidder]
AS13
22

Fig. 4. Augmented model of transition `withdraw`.

4.1 Observational Equivalence

We study sufficient conditions for augmented models to be behaviorally equivalent to initial models. To do that, we use observational equivalence [30] by considering non-observable $\beta-$transitions. We denote by S_I and S_E the set of states of the smart contract transition system and its augmented derivative, respectively. We show that $R = \{(q,r) \in S_I \times S_E\}$ is a weak bi-simulation by considering as observable transitions A, those that affect the ledger state, while the remaining transitions B are considered non-observable transitions. According to this definition, the set of transitions in the smart contract system, which represent the execution semantics of a Solidity named function or the fallback, are all observable. On the other hand, the augmented system represents each Solidity function using paths of multiple transitions. We assume that final transition of each such path is an α transition, while the rest are β transitions. Our weak bi-simulation is based on the fact the effect of each $\alpha \in A$ on the ledger state is equal for the states of S_I and S_E. Therefore, if $\sigma_I = \sigma_E$ at the initial state of α, then $\sigma'_I = \sigma'_E$ at the resulting state.

A weak simulation over I and E is a relation $R \subseteq S_I \times S_E$ such that we have:

Property 1. For all $(q,r) \in R$ and for each $\alpha \in A$, such that $q \xrightarrow{\alpha} q'$, there is r' such that $r \overset{\beta^\star\alpha\beta^\star}{\longrightarrow} r'$ where $(q',r') \in R$

For each observable transition α of a state in S_I, it should be proved that (i) a path that consists of α and other non-observable transitions exists in all its equivalent states in S_E, and (ii) the resulting states are equivalent.

Property 2. For all $(q,r) \in R$ and $\alpha \in A$, such that $r \xrightarrow{\alpha} r'$, there is q' such that $q \xrightarrow{\alpha} q'$ where $(q',r') \in R$.

For each observable outgoing transition in a state in S_E, it should be proved that (i) there is an outgoing observable transition in all its equivalent states in S_I, and (ii) the resulting states are equivalent.

Property 3. For all $(q, r) \in R$ and $\beta \in B$ such that $r \xrightarrow{\beta} r'$, $(q, r') \in R$
For each non observable transition, it should be proved that the the resulting state is equivalent with all the states that are equivalent with the initial state.

Theorem 1. *For each initial smart contract I and its corresponding augmented smart contract E, it holds that $I \sim E$.*

The proof of Theorem 1 is presented in the [29, Appendix D.3].

5 Verification Process

Our verification approach checks whether contract behavior satisfies properties that are required by the developer. To check this, we must take into account the effect of data and time. However, smart contracts use environmental input as control data, e.g., in guards. Such input data can be infinite, leading to infinitely many possible contract states. Exploring every such state is highly inefficient [11] and hence, appropriate data and time abstractions must be employed.

We apply data abstraction to ignore variables that depend on (e.g., are updated by) environmental input. Thus, an overapproximation of the contract behavior is caused by the fact that transition guards with such variables are not evaluated; instead, both their values are assumed possible and state space exploration includes execution traces with and without each guarded transition. In essence, we analyze a more abstract model of the contract, with a set of reachable states and traces that is a superset of the set of states (respectively, traces) of the actual contract. As an example, let us consider the function in Fig. 5.

```
void fn(int x) {
   if (x < 0) {
       ...       (1)
   }
   if (x > 0) {
       ...       (2)
   }
}
```

Fig. 5. Code example.

An overapproximation of the function's execution includes traces where both lines (1) and (2) are visited, even though they cannot both be satisfied by the same values of x. Note that abstraction is not necessary for variables that are independent of environment input (e.g. iteration counters of known range). These are updated in the model as they are calculated by contract statements.

We also apply abstraction to time variables (e.g. the `now` variable in the Blind Auction) using a slightly different approach. Although we need to know which transitions get invalidated as time increases, we do not represent the time spent in each state, as this time can be arbitrarily high. Therefore, for a time-guarded transition in the model, say from a state s_x, one of the following applies:

- if the guard is of type $t \leq t_{max}$, checking that a time variable does not exceed a threshold, a loop transition is added to s_x, with an action $t = t_{max} + 1$ that invalidates the guard. A deadlock may be found in traces where this invalidating loop is executed (e.g., if no other transitions are offered in s_x).
- if the guard is of type $t > t_{min}$, checking that a time variable exceeds a threshold, an action $t{=}t_{min}{+}1$ is added to the guarded transition. This sets the time to the earliest point that next state can be reached (e.g., useful for checking bounded liveness properties.)

This overapproximation has the following implications.

Safety Properties: *Safety properties that are fulfilled in the abstract model are also guaranteed in the actual system.* Each safety property checks the non-reachability of a set of erroneous states. If these states are unreachable in the abstract model, they will be unreachable in the concrete model, which contains a subset of the abstract model's states. This property type is useful for checking vulnerabilities in currency withdrawal functions (e.g., the "DAO attack").

Liveness Properties: *Liveness properties that are violated in the abstract model are also violated in the actual system.* Each liveness property checks that a set of states are reachable. If they are found unreachable (i.e., liveness violation) in the abstract model, they will also be unreachable in the concrete model. This property type is useful for "Denial-of-Service" vulnerabilities ([29, Appendix C.2]).

Deadlock Freedom: States without enabled outgoing transitions are identified as deadlock states. If no deadlock states are reachable in the abstract model, they will not be reachable in the actual system.

5.1 VeriSolid-to-BIP Mapping

Since both VeriSolid and BIP model contract behavior as transition systems, the transformation is a simple mapping between the transitions, states, guards, and actions of VeriSolid to the transitions, states, guards, and actions of BIP (see [29, Appendix C.3] for background on BIP). Because this is an one-to-one mapping, we do not provide a proof. Our translation algorithm performs a single-pass syntax-directed parsing of the user's VeriSolid input and collects values that are appended to the attributes list of the templates. Specifically, the following values are collected:

- variables $v \in V$, where $type(v)$ is the data type of v and $name(v)$ is the variable name (i.e., identifier);
- states $s \in S$;

– transitions $t \in T$, where t^{name} is the transition (and corresponding port) name, t^{from} and t^{to} are the outgoing and incoming states, a_t and g_t are invocations to functions that implement the associated actions and guards.

Figure 6 shows the BIP code template. We use `fixed-width` font for the generated output, and *italic* font for elements that are replaced with input.

```
                atom type Contract()
$\forall v \in V$ :      data $type(v)$ $name(v)$
$\forall t \in T$ :      export port synPort $t^{name}$()
                  places $s_0, \ldots, s_{|S|-1}$
                  initial to $s_0$
$\forall t \in T$ :      on $t^{name}$ from $t^{from}$ to $t^{to}$
                     provided ($g_t$) do {$a_t$}
                end
```

Fig. 6. BIP code generation template.

Table 2. Analyzed properties and verification results for the case study models.

Case Study	Properties	Type	Result
BlindAuction (initial) states: 54	(i) `bid` cannot happen after `close`: $\texttt{AG}\big(close \rightarrow \texttt{AG}\neg bid\big)$	Safety	Verified
	(ii) `cancelABB` or `cancelRB` cannot happen after `finish`: $\texttt{AG}\big(finish \rightarrow \texttt{AG}\neg\big(cancelRB \vee cancelABB\big)\big)$	Safety	Verified
	(iii) `withdraw` can happen only after `finish`: $\texttt{A}\big[\neg withdraw \;\texttt{W}\; finish\big]$	Safety	Verified
	(iv) `finish` can happen only after `close`: $\texttt{A}\big[\neg finish \;\texttt{W}\; close\big]$	Safety	Verified
BlindAuction (augmented) states: 161	(v) `23` cannot happen after `18`: $\texttt{AG}\big(18 \rightarrow \texttt{AG}\neg 23\big)$	Safety	Verified
	(vi) if `21` happens, `21` can happen only after `24`: $\texttt{AG}\big(21 \rightarrow \texttt{AX}\;\texttt{A}\big[\neg 21 \;\texttt{W}\; \big(24\big)\big]\big)$	Safety	Verified
DAO attack states: 9	if `call` happens, `call` can happen only after `subtract`: $\texttt{AG}\big(call \rightarrow \texttt{AX}\;\texttt{A}\big[\neg call \;\texttt{W}\; subtract\big]\big)$	Safety	Verified
King of Ether 1 states: 10	7 will eventually happen after 4: $\texttt{AG}\big(4 \rightarrow \texttt{AF}\; 7\big)$	Liveness	Violated
King of Ether 2 states: 10	8 will eventually happen after `fallback`: $\texttt{AG}\big(fallback \rightarrow \texttt{AF}\; 8\big)$	Liveness	Violated

5.2 Verification Results

Table 2 summarizes the properties and verification results. For ease of presentation, when properties include statements, we replace statements with the augmented-transition numbers that we have added to [29, Figures 9, 10, and 12]. The number of states represents the reachable state space as evaluated by nuXmv.

Blind Auction. We analyzed both the initial and augmented models of the Blind Auction contract. On the initial model, we checked four safety properties (see Properties (i)–(iv) in Table 2). On the augmented model, which allows for more fine-grained analysis, we checked two additional safety properties. All properties were verified to hold. The models were found to be deadlock-free and their state space was evaluated to 54 and 161 states, respectively. The augmented model and generated code can be found in [29, Appendix F].

The DAO Attack. We modeled a simplified version of the DAO contract. Atzei et al. [2] discuss two different vulnerabilities exploited on DAO and present different attack scenarios. Our verified safety property (Table 2) excludes the possibility of both attacks. The augmented model can be found in [29, Appendix G.1].

King of the Ether Throne. For checking Denial-of-Service vulnerabilities, we created models of two versions of the King of the Ether contract [2], which are provided in [29, Appendix G.2]. On "King of Ether 1," we checked a liveness property stating that crowning (transition 7) will happen at some time after the compensation calculation (transition 4). The property is violated by the following counterexample: *fallback* $\rightarrow$ *4* $\rightarrow$ *5*. A second liveness property, which states that the crowning will happen at some time after fallback fails in "King of Ether 2." A counterexample of the property violation is the following: *fallback* $\rightarrow$ *4*. Note that usually many counterexamples may exist for the same violation.

Resource Allocation. We have additionally verified a larger smart contract that acts as the core of a blockchain-based platform for transactive energy systems. The reachable state space, as evaluated by nuXmv, is 3, 487. Properties were verified or shown to be violated within seconds. Due to space limitations, we present the verification results in [29, Appendix G.3].

6 Related Work

Here, we present a brief overview of related work. We provide a more detailed discussion in [29, Appendix H].

Motivated by the large number of smart-contract vulnerabilities in practice, researchers have investigated and established taxonomies for common types of contract vulnerabilities [2, 25]. To find vulnerabilities in existing contracts, both verification and vulnerability discovery are considered in the literature [36]. In comparison, the main advantage of our model-based approach is that it allows developers to specify desired properties with respect to a high-level model instead of, e.g., EVM bytecode, and also provides verification results and counterexamples in a developer-friendly, easy to understand, high-level form. Further, our approach allows verifying whether a contract satisfies all desired security properties instead of detecting certain types of vulnerabilities; hence, it can detect atypical vulnerabilities.

Hirai performs a formal verification of a smart contract used by the Ethereum Name Service [20] and defines the complete instruction set of the Ethereum Virtual Machine (EVM) in Lem, a language that can be compiled for interactive theorem provers, which enables proving certain safety properties for existing contracts [21]. Bhargavan et al. outline a framework for verifying the safety and correctness of Ethereum contracts based on translating Solidity and EVM bytecode contracts into F^* [6]. Tsankov et al. introduce a security analyzer for Ethereum contracts, called Securify, which symbolically encodes the dependence graph of a contract in stratified Datalog [23] and then uses off-the-shelf solvers to check the satisfaction of properties [42]. Atzei et al. prove the well-formedness properties of the Bitcoin blockchain have also been proven using a formal model [3]. Techniques from runtime verification are used to detect and recover from violations at runtime [12,13].

Luu et al. provide a tool called Oyente, which can analyze contracts and detect certain typical security vulnerabilities [25]. Building on Oyente, Albert et al. introduce the EthIR framework, which can produce a rule-based representation of bytecode, enabling the application of existing analysis to infer properties of the EVM code [1]. Nikolic et al. present the MAIAN tool for detecting three types of vulnerable contracts, called prodigal, suicidal and greedy [33]. Fröwis and Böhme define a heuristic indicator of control flow immutability to quantify the prevalence of contractual loopholes based on modifying the control flow of Ethereum contracts [16]. Brent et al. introduce a security analysis framework for Ethereum contracts, called Vandal, which converts EVM bytecode to semantic relations, which are then analyzed to detect vulnerabilities described in the Soufflé language [8]. Mueller presents Mythril, a security analysis tool for Ethereum smart contracts with a symbolic execution backend [31]. Stortz introduces Rattle, a static analysis framework for EVM bytecode [41].

Researchers also focus on providing formal operational semantics for EVM bytecode and Solidity language [17–19,24,46]. Common design patterns in Ethereum smart contracts are also identified and studied by multiple research efforts [4,44]. Finally, to facilitate development, researchers have also introduced a functional smart-contract language [35], an approach for semi-automated translation of human-readable contract representations into computational equivalents [15], a logic-based smart-contract model [22].

7 Conclusion

We presented an end-to-end framework that allows the generation of correct-by-design contracts by performing a set of equivalent transformations. First, we generate an augmented transition system from an initial transition system, based on the operational semantics of supported Solidity statements ([29, Appendix A.3]). We have proven that the two transition systems are observationally equivalent (Sect. 4.1). Second, we generate the BIP transition system from the augmented transition system through a direct one-to-one mapping. Third, we generate the NuSMV transition system from the BIP system (shown to be observationally equivalent in [34]). Finally, we generate functionally equivalent Solidity code, based on the operational semantics of the transition system ([29, Appendix A.2]).

To the best of our knowledge, VeriSolid is the first framework to promote a model-based, correctness-by-design approach for blockchain-based smart contracts. Properties established at any step of the VeriSolid design flow are preserved in the resulting smart contracts, guaranteeing their correctness. VeriSolid fully automates the process of verification and code generation, while enhancing usability by providing easy-to-use graphical editors for the specification of transition systems and natural-like language templates for the specification of formal properties. By performing verification early at design time, we provide a cost-effective approach; fixing bugs later in the development process can be very expensive. Our verification approach can detect typical vulnerabilities, but it may also detect any violation of required properties. Since our tool applies verification at a high-level, it can provide meaningful feedback to the developer when a property is not satisfied, which would be much harder to do at bytecode level. Future work includes extending the approach to model and generate correct-by-design *systems of interacting smart contracts*.

References

1. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: a framework for high-level analysis of Ethereum bytecode. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 513–520. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_30
2. 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
3. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Meiklejohn, S., Sako, K. (eds.) FC 2018. LNCS, vol. 10957. Springer, Berlin (2018). https://doi.org/10.1007/978-3-662-58387-6_29
4. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 494–509. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_31
5. Basu, A., et al.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)

6. Bhargavan, K., et al.: Short paper: formal verification of smart contracts. In: Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in Conjunction with ACM CCS 2016, pp. 91–96, October 2016
7. Bliudze, S., et al.: Formal verification of infinite-state BIP models. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 326–343. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_25
8. Brent, L., et al.: Vandal: a scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)
9. Christidis, K., Devetsikiotis, M.: Blockchains and smart contracts for the Internet of Things. IEEE Access **4**, 2292–2303 (2016)
10. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016)
11. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
12. Colombo, C., Ellul, J., Pace, G.J.: Contracts over smart contracts: recovering from violations dynamically. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 300–315. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_23
13. Ellul, J., Pace, G.: Runtime verification of Ethereum smart contracts. In: Workshop on Blockchain Dependability (WBD), in Conjunction with 14th European Dependable Computing Conference (EDCC) (2018)
14. Finley, K.: A $50 million hack just showed that the DAO was all too human. Wired. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/ (2016)
15. Frantz, C.K., Nowostawski, M.: From institutions to code: towards automated generation of smart contracts. In: 1st IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 210–215. IEEE (2016)
16. Fröwis, M., Böhme, R.: In code we trust? In: Garcia-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) ESORICS/DPM/CBT -2017. LNCS, vol. 10436, pp. 357–372. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67816-0_20
17. 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
18. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. Technical report, TU Wien (2018)
19. Hildenbrandt, E., et al.: KEVM: a complete semantics of the Ethereum virtual machine. Technical report, UIUC (2017)
20. Hirai, Y.: Formal verification of deed contract in Ethereum name service, November 2016. https://yoichihirai.com/deed.pdf
21. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
22. Hu, J., Zhong, Y.: A method of logic-based smart contracts for blockchain system. In: Proceedings of the 4th International Conference on Data Processing and Applications (ICPDA), pp. 58–61. ACM (2018)
23. Jeffrey, D.U.: Principles of Database and Knowledge-base Systems. Computer Science Press, New york (1989)
24. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Executable operational semantics of Solidity. arXiv preprint arXiv:1804.01295 (2018)

25. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 254–269. ACM, October 2016
26. Maróti, M., et al.: Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. In: Proceedings of the MPM@ MoDELS, pp. 41–60 (2014)
27. Mavridou, A., Laszka, A.: Designing secure Ethereum smart contracts: a finite state machine based approach. In: Meiklejohn, S., Sako, K. (eds.) FC 2018. LNCS, vol. 10957. Springer, Berlin (2018). https://doi.org/10.1007/978-3-662-58387-6_28
28. 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
29. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: correct-by-design smart contracts for Ethereum. arXiv preprint arXiv:1901.01292 (2019). https://arxiv.org/pdf/1901.01292.pdf
30. Milner, R.: Communication and Concurrency, vol. 84. Prentice Hall, New York (1989)
31. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf) (2018)
32. Newman, L.H.: Security news this week: $280m worth of Ethereum is trapped thanks to a dumb bug. Wired, November 2017. https://www.wired.com/story/280m-worth-of-ethereum-is-trapped-for-a-pretty-dumb-reason/
33. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: 34th Annual Computer Security Applications Conference (ACSAC) (2018)
34. Noureddine, M., Jaber, M., Bliudze, S., Zaraket, F.A.: Reduction and abstraction techniques for BIP. In: Lanese, I., Madelaine, E. (eds.) FACS 2014. LNCS, vol. 8997, pp. 288–305. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15317-9_18
35. O'Connor, R.: Simplicity: a new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS 2017, pp. 107–120. ACM, New York (2017). https://doi.org/10.1145/3139337.3139340
36. Parizi, R.M., Dehghantanha, A., Choo, K.K.R., Singh, A.: Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: 28th Annual International Conference on Computer Science and Software Engineering (CASCON) (2018)
37. Plotkin, G.D.: A structural approach to operational semantics. Computer Science Department, Aarhus University, Denmark (1981)
38. Solidity by example: blind auction (2018). https://solidity.readthedocs.io/en/develop/solidity-by-example.html#blind-auction. Accessed 25 Sept 2018
39. Solidity documentation: common patterns (2018). http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine. Accessed 25 Sept 2018
40. Solidity documentation: security considerations - use the checks-effects-interactions pattern (2018). http://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern. Accessed 25 Sept 2018
41. Stortz, R.: Rattle - an Ethereum EVM binary analysis framework. In: REcon, Montreal (2018)
42. Tsankov, P., Dan, A., Cohen, D.D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: 25th ACM Conference on Computer and Communications Security (CCS) (2018)

43. Underwood, S.: Blockchain beyond Bitcoin. Commun. ACM **59**(11), 15–17 (2016)
44. Wöhrer, M., Zdun, U.: Design patterns for smart contracts in the Ethereum ecosystem. In: Proceedings of the 2018 IEEE Conference on Blockchain, pp. 1513–1520 (2018)
45. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Technical report, EIP-150, Ethereum Project - Yellow Paper, April 2014
46. Yang, Z., Lei, H.: Lolisa: formal syntax and semantics for a subset of the solidity programming language. arXiv preprint arXiv:1803.09885 (2018)