VeriSolid: Correct-by-Design Smart Contracts for Ethereum

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

[1] Vanderbilt University
[2] University of Houston
[3] Aristotle University of Thessaloniki

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 [50,52] 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 [12]. Since smart contracts can perform any computation[4], they allow the development of decentralized applications, whose execution is safeguarded by the security properties of the underlying platform. Due to their unique advantages, blockchain based platforms are envisioned to have a wide range of applications, ranging from financial to the Internet-of-Things [11].

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

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

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 [29]. 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 [29]. 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 [16], and the 2017 hack of the multisignature Parity Wallet library [36], 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 [47]), automated vulnerability-discovery tools (e.g., Oyente [29,49]), and formal verification of correctness (e.g., [23,19]). 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* [32,33] framework, which provides a graphical editor for specifying Ethereum smart contracts as transitions systems and a *Solidity* code generator.[5] 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 (Section 3.3) and for supported Solidity statements (Appendix A.3), and extend the Solidity code generator (Appendix E).
- We design and implement developer-friendly natural-language like templates for specifying safety and liveness properties (Section 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 transfor-

[5] 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.

mation 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 (Section 4). We prove that the initial and augmented transition systems are observationally equivalent (Section 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 (Section 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[6] and web-based framework that is built on top of WebGME [30] and FSolidM [32,33]. 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.
- A list of properties to be verified, which can be expressed using predefined natural-language like templates.

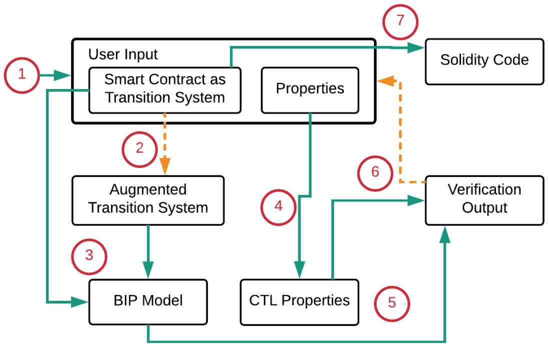

Fig. 1. Design and verification workflow.

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[7]. 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 [6] or nuXmv [9] (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

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

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

the contract is automatically generated in step ⑦. The following sections describe the steps from Figure 1 in detail. Due to space limitations, we present the Solidity code generation (step ⑦) in 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 [32], which is based on an example from the Solidity documentation [44].

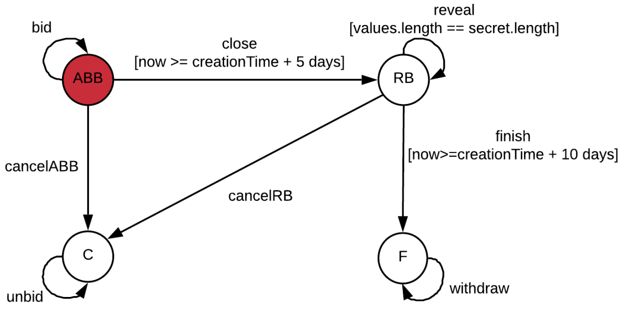
bid
close
[now >= creationTime + 5 days]
reveal
[values.length == secret.length]
ABB
RB
cancelABB
cancelRB
finish
[now>=creationTime + 10 days]
C
F
unbid
withdraw

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

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* [45], which comprises 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.

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 Figure 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

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

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 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.

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 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 Appendix E). This prevents re-entrancy since none of the contract functions[8] can be called in this state. 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 [41]. 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 $\mathrm{Eval}(\sigma, \mathrm{Exp}) \to \langle(\hat{\sigma}, x), v\rangle$ 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.[9]

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:

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

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

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

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

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

⟨***Transitions*** ∪ ***Statements***⟩ cannot happen after ⟨***Transitions*** ∪ ***Statements***⟩.

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** ∪ **cancelRB**.

If ⟨***Transitions*** ∪ ***Statements***⟩ happens, ⟨***Transitions*** ∪ ***Statements***⟩ can happen only after ⟨***Transitions*** ∪ ***Statements***⟩ 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 Figure 3.

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

```
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.

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 Section 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 (Appendix C.2):

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

4 Augmented Transition System Transformation

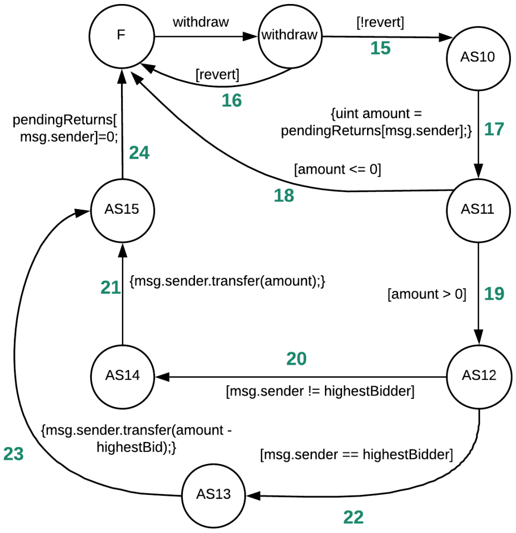

Fig. 4. Augmented model of transition `withdraw`.

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 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 Appendix F. The action of the original `withdraw` transition is shown by Figure 3. Notice the added state `withdraw`, which avoids re-entrancy by design, as explained in Section 3.2.

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 [34] 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 $q \overset{\alpha}{\rightarrow} q'$, 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 $r \overset{\alpha}{\rightarrow} r'$, there is q' such that $q \overset{\alpha}{\rightarrow} q'$ 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 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 [13] 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 Figure 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 (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

```
               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.

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 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.

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`: $\texttt{AG}\big(\mathit{close} \rightarrow \texttt{AG}\neg \mathit{bid}\big)$	Safety	Verified
	(ii) `cancelABB` or `cancelRB` cannot happen after `finish`: $\texttt{AG}\big(\mathit{finish} \rightarrow \texttt{AG}\neg\big(\mathit{cancelRB} \vee \mathit{cancelABB}\big)\big)$	Safety	Verified
	(iii) `withdraw` can happen only after `finish`: $\texttt{A}\big[\,\neg \mathit{withdraw}\ \texttt{W}\ \mathit{finish}\,\big]$	Safety	Verified
	(iv) `finish` can happen only after `close`: $\texttt{A}\big[\,\neg \mathit{finish}\ \texttt{W}\ \mathit{close}\,\big]$	Safety	Verified
BlindAuction (augmented) states: 161	(v) `23` cannot happen after `18`: $\texttt{AG}\big(\mathit{18} \rightarrow \texttt{AG}\neg \mathit{23}\big)$	Safety	Verified
	(vi) if `21` happens, `21` can happen only after `24`: $\texttt{AG}\big(\mathit{21} \rightarrow \texttt{AX}\ \texttt{A}\big[\neg \mathit{21}\ \texttt{W}\ \big(\mathit{24}\big)\,\big]\big)$	Safety	Verified
DAO attack states: 9	if `call` happens, `call` can happen only after `subtract`: $\texttt{AG}\big(\mathit{call} \rightarrow \texttt{AX}\ \texttt{A}\big[\neg \mathit{call}\ \texttt{W}\ \mathit{subtract}\big]\big)$	Safety	Verified
King of Ether 1 states: 10	`7` will eventually happen after `4`: $\texttt{AG}\big(\mathit{4} \rightarrow \texttt{AF}\ \mathit{7}\ \big)$	Liveness	Violated
King of Ether 2 states: 10	`8` will eventually happen after `fallback`: $\texttt{AG}\big(\mathit{fallback} \rightarrow \texttt{AF}\ \mathit{8}\ \big)$	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 Figures 8, 10, and 11 in Appendices F.1 and G.2. 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 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 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 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: $\mathit{fallback} \rightarrow \mathit{4} \rightarrow \mathit{5}$. 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: $\mathit{fallback} \rightarrow \mathit{4}$. 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 Appendix G.3.

6 Related Work

Here, we present a brief overview of related work. We provide a more detailed discussion in 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 [2,29]. To find vulnerabilities in existing contracts, both verification and vulnerability discovery are considered in the literature [40]. 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 [22] 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 [23]. 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^* [8]. Tsankov et al. introduce a security analyzer for Ethereum contracts, called Securify, which symbolically encodes the dependence graph of a contract in stratified Datalog [25] and then uses off-the-shelf solvers to check the satisfaction of properties [49]. 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 [15,14].

Luu et al. provide a tool called Oyente, which can analyze contracts and detect certain typical security vulnerabilities [29]. 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 codee [1]. Nikolic et al. present the MAIAN tool for detecting three types of vulnerable contracts, called prodigal, suicidal and greedy [37]. 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 [18]. 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 [10]. Mueller presents Mythril, a security analysis tool for Ethereum smart contracts with a symbolic execution backend [35]. Stortz introduces Rattle, a static analysis framework for EVM bytecode [48].

Researchers also focus on providing formal operational semantics for EVM bytecode and Solidity language [21,19,20,53,26]. Common design patterns in Ethereum smart contracts are also identified and studied by multiple research efforts [5,51]. Finally, to facilitate development, researchers have also introduced a functional smart-contract language [39], an approach for semi-automated translation of human-readable contract representations into computational equivalents [17], a logic-based smart-contract model [24].

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 (Appendix A.3). We have proven that the two transition systems are observationally equivalent (Section 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 [38]). Finally, we generate functionally equivalent Solidity code, based on the operational semantics of the transition system (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: 16th International Symposium on Automated Technology for Verification and Analysis (ATVA) (2018)
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Proceedings of the 6th International Conference on Principles of Security and Trust (POST). pp. 164–186. Springer (April 2017)
3. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC) (2018)
4. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
5. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: Platforms, applications, and design patterns. In: Proceedings of the 1st Workshop on Trusted Smart Contracts, in conjunction with the 21st International Conference of Financial Cryptography and Data Security (FC) (April 2017)
6. Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. IEEE Software **28**(3), 41–48 (2011)
7. Basu, A., Gallien, M., Lesire, C., Nguyen, T.H., Bensalem, S., Ingrand, F., Sifakis, J.: Incremental component-based construction and verification of a robotic system. In: ECAI. vol. 178, pp. 631–635 (2008)
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: 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)
9. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state BIP models. In: Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis (ATVA). pp. 326–343. Springer (2015)
10. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)
11. Christidis, K., Devetsikiotis, M.: Blockchains and smart contracts for the internet of things. IEEE Access **4**, 2292–2303 (2016)
12. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: Foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016)
13. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (Sep 1994)
14. Colombo, C., Ellul, J., Pace, G.J.: Contracts over smart contracts: Recovering from violations dynamically. In: 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA) (2018)
15. 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)
16. 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/` (June 2016)

17. 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)
18. Fröwis, M., Böhme, R.: In code we trust? In: International Workshop on Cryptocurrencies and Blockchain Technology (CBT). pp. 357–372. Springer (September 2017)
19. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: 7th International Conference on Principles of Security and Trust (POST). pp. 243–269. Springer (2018)
20. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts – Technical report. Tech. rep., TU Wien (2018)
21. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: KEVM: A complete semantics of the Ethereum virtual machine. Tech. rep., UIUC (2017)
22. Hirai, Y.: Formal verification of deed contract in Ethereum name service. `https://yoichihirai.com/deed.pdf` (November 2016)
23. Hirai, Y.: Defining the Ethereum Virtual Machine for interactive theorem provers. In: Proceedings of the 1st Workshop on Trusted Smart Contracts, in conjunction with the 21st International Conference of Financial Cryptography and Data Security (FC) (April 2017)
24. 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)
25. Jeffrey, D.U.: Principles of database and knowledge-base systems (1989)
26. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Executable operational semantics of Solidity. arXiv preprint arXiv:1804.01295 (2018)
27. K Team: K-framework. `http://www.kframework.org/index.php/` (2017), accessed on 9/25/2018.
28. Laszka, A., Eisele, S., Dubey, A., Karsai, G.: TRANSAX: A blockchain-based decentralized forward-trading energy exchange for transactive microgrids. In: Proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS) (December 2018)
29. 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)
30. Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., Lédeczi, Á.: Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. In: Proceedings of the MPM@ MoDELS. pp. 41–60 (2014)
31. Mavridou, A., Emmanouela, S., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis, J.: Architecture-based design: A satellite on-board software case study. In: Proceedings of the 13th International Conference on Formal Aspects of Component Software (FACS). pp. 260–279 (October 2016)
32. Mavridou, A., Laszka, A.: Designing secure Ethereum smart contracts: A finite state machine based approach. In: Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC) (February 2018)
33. Mavridou, A., Laszka, A.: Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In: Proceedings of the 7th International Conference on Principles of Security and Trust (POST) (April 2018)

34. Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989)
35. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. 9th Annual HITB Security Conference (HITBSecConf) (2018)
36. Newman, L.H.: Security news this week: $280m worth of Ethereum is trapped thanks to a dumb bug. WIRED, https://www.wired.com/story/280m-worth-of-ethereum-is-trapped-for-a-pretty-dumb-reason/ (November 2017)
37. 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)
38. Noureddine, M., Jaber, M., Bliudze, S., Zaraket, F.A.: Reduction and abstraction techniques for bip. In: Proceedings of the International Workshop on Formal Aspects of Component Software. pp. 288–305. Springer (2014)
39. O'Connor, R.: Simplicity: A new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. pp. 107–120. PLAS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3139337.3139340, http://doi.acm.org/10.1145/3139337.3139340
40. 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)
41. Plotkin, G.D.: A structural approach to operational semantics. Computer Science Department, Aarhus University, Denmark (1981)
42. Said, N.B., Abdellatif, T., Bensalem, S., Bozga, M.: Model-driven information flow security for component-based systems. In: From Programs to Systems. The Systems perspective in Computing, pp. 1–20. Springer (2014)
43. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Proceedings of the International Conference on Financial Cryptography and Data Security (FC). pp. 478–493. Springer (2017)
44. Solidity by Example: Blind auction. https://solidity.readthedocs.io/en/develop/solidity-by-example.html#blind-auction (2018), accessed on 9/25/2018.
45. Solidity Documentation: Common patterns. http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine (2018), accessed on 9/25/2018.
46. Solidity Documentation: Function calls. http://solidity.readthedocs.io/en/develop/control-structures.html#function-calls (2018), accessed on 9/25/2018.
47. Solidity Documentation: Security considerations – use the Checks-Effects-Interactions pattern. http://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern (2018), accessed on 9/25/2018.
48. Stortz, R.: Rattle – an Ethereum EVM binary analysis framework. REcon Montreal (2018)
49. 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)
50. Underwood, S.: Blockchain beyond Bitcoin. Communications of the ACM **59**(11), 15–17 (2016)

51. 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)
52. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Tech. Rep. EIP-150, Ethereum Project – Yellow Paper (April 2014)
53. Yang, Z., Lei, H.: Lolisa: Formal syntax and semantics for a subset of the solidity programming language. arXiv preprint arXiv:1803.09885 (2018)

A Formalisms

A.1 Supported Solidity Subset

Here, we define the subset of Solidity that VeriSolid supports. First, let us introduce the following notation:

- Let $\mathbb{T}$ denote the set of Solidity types;
- let $\mathbb{I}$ denote the set of valid Solidity identifiers;
- let $\mathbb{D}$ denote the set of Solidity event and custom type definitions;
- let $\mathbb{E}$ denote the set of Solidity expressions;
- let $\mathbb{C}$ denote the set of Solidity expressions without side effects (i.e., expression whose evaluation does not change storage, memory, balances, etc.);
- let $\mathbb{S}$ denote the set of supported Solidity statements.

We define the set of supported event ($\langle \mathit{event} \rangle$) and custom type ($\langle \mathit{struct} \rangle$) definitions $\mathbb{D}$ as follows:

$$\begin{aligned} \langle \mathit{event} \rangle ::= \texttt{event}\ @\mathit{identifier}\ \texttt{(}\ \big(@\mathit{type}\ @\mathit{identifier} \\ (\texttt{,}\ @\mathit{type}\ @\mathit{identifier})*\big)?\ \texttt{);} \end{aligned}$$

$$\langle \mathit{struct} \rangle ::= \texttt{struct}\ @\mathit{identifier}\ \texttt{\{}\ (@\mathit{type}\ @\mathit{identifier}\ \texttt{;})*\ \texttt{\}}$$

We let $\mathbb{E}$ denote the set of Solidity expressions. We let $\mathbb{C}$ denote the following subset of Solidity expressions, which do not have any side effects:

$$\begin{aligned} \langle \mathit{pure} \rangle ::= &\mid \langle \mathit{variable} \rangle \\ &\mid @\mathit{constant} \\ &\mid \texttt{(}\ \langle \mathit{pure} \rangle\ \texttt{)} \\ &\mid \langle \mathit{unary} \rangle\ \langle \mathit{pure} \rangle \\ &\mid \langle \mathit{pure} \rangle\ \langle \mathit{operator} \rangle\ \langle \mathit{pure} \rangle \end{aligned}$$

$$\begin{aligned} \langle \mathit{variable} \rangle ::= &\mid @\mathit{identifier} \\ &\mid \langle \mathit{variable} \rangle\ \texttt{.}\ @\mathit{identifier} \\ &\mid \langle \mathit{variable} \rangle\ \texttt{[}\ \langle \mathit{pure} \rangle\ \texttt{]} \end{aligned}$$

$$\begin{aligned} \langle \mathit{operator} \rangle ::= &\ \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<=} \\ &\mid \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \end{aligned}$$

$$\langle \mathit{unary} \rangle ::= \ \texttt{!} \mid \texttt{+} \mid \texttt{-}$$

VeriSolid supports the following types of statements:

- variable declarations (e.g., `int32 value = 0;` and `address from = msg.sender;`),

- expressions (e.g., `amount = balance[msg.sender];` or `msg.sender.transfer(amount);`),
- event statements (e.g., `emit Deposit(amount, msg.sender);`),
- `return` statements (e.g., `return;` and `return amount;`),
- `if` and `if` ... `else` selection statements (including `if` ... `else if` ... and so on),
- `for` and `while` loop statements,
- compound statements (i.e., `{` *`statement1 statement2`* `...` `}`).

We define the formal grammar of the subset of supported Solidity statements $\mathbb{S}$ as follows:

$$
\begin{aligned}
\langle \mathit{statement} \rangle ::= & \\
& \mid \langle \mathit{declaration} \rangle\ \texttt{;} \\
& \mid @\mathit{expression}\ \texttt{;} \\
& \mid \texttt{emit}\ @\mathit{identifier}\texttt{(}\ \big(@\mathit{expression} \\
& \qquad (\texttt{,}\ @\mathit{expression}) * \big)?\texttt{);} \\
& \mid \texttt{return}\ (@\mathit{pure})?\ \texttt{;} \\
& \mid \texttt{if}\ \texttt{(}\ @\mathit{expression}\ \texttt{)}\ \langle \mathit{statement} \rangle \\
& \qquad (\texttt{else}\ \langle \mathit{statement} \rangle)? \\
& \mid \texttt{for}\ \texttt{(}\ \langle \mathit{declaration} \rangle\ \texttt{;}\ @\mathit{expression}\ \texttt{;} \\
& \qquad @\mathit{expression}\ \texttt{)}\ \langle \mathit{statement} \rangle \\
& \mid \texttt{while}\ \texttt{(}\ @\mathit{expression}\texttt{)}\ \langle \mathit{statement} \rangle \\
& \mid \texttt{\{}\ (\langle \mathit{statement} \rangle) *\ \texttt{\}}
\end{aligned}
$$

$$
\langle \mathit{declaration} \rangle ::=\ @\mathit{type}\ @\mathit{identifier}\ (\texttt{=}\ @\mathit{expression})?
$$

where $@\mathit{expression} \in \mathbb{E}$ is a primary Solidity expression, which may include function calls, transfers, etc., while $@\mathit{pure} \in \mathbb{C}$ is a Solidity expression without side effects.

A.2 Operational Semantics of the Transition System

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 equal to 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.

A transition is triggered by providing a transition (i.e., function) $\mathit{name} \in \mathbb{I}$ 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:

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

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` without any exceptions (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 Appendix A.3). Finally, if the execution status resulting from the action 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.

The normal execution of a transition that returns a value is captured by the TRANSITION-RET rule:

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

This rule is applied if the transition action a_t finishes with a `return` v statement, resulting in execution status $R[v]$.

If the transition t by name $t^{name} = name$ exists, but its source state t^{from} is not s, then the transition is not executed, which is captured by the TRANSITION-WRO rule:

$$\text{TRANSITION-WRO}\quad \frac{t \in T, \quad name = t^{name}, \quad s \neq t^{from}}{\langle (\Psi, s), name\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, s, \cdot) \rangle}$$

Similarly, if the guard condition g_t of the transition evaluates $\text{Eval}(\sigma, g_t)$ to `false`, then the transition is reverted, , which is captured by the TRANSITION-GRD rule:

$$\text{TRANSITION-GRD}\quad \frac{\begin{array}{c} t \in T, \quad name = t^{name}, \quad s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M) \\ \text{Eval}(\sigma, g_t) \rightarrow \langle (\hat{\sigma}, N), \texttt{false} \rangle \end{array}}{\langle (\Psi, s), name\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, s, \cdot) \rangle}$$

If an exception is raised during the evaluation $\text{Eval}(\sigma, g_t)$ of the guard condition g_t (i.e., if the execution status becomes E), then the transition is reverted, which is captured by the TRANSITION-EXC1 rule:

$$\text{TRANSITION-EXC1}\quad \frac{\begin{array}{c} t \in T, \quad name = t^{name}, \quad s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M) \\ \text{Eval}(\sigma, g_t) \rightarrow \langle (\hat{\sigma}, E), x \rangle \end{array}}{\langle (\Psi, s), name\,(v_1, v_2, \ldots) \rangle \rightarrow \langle (\Psi, s, \cdot) \rangle}$$

Similarly, if an exception is raised during the execution of the transition action a_t, then the transition is reverted, which is captured by the TRANSITION-EXC2 rule:

$$\text{TRANSITION-EXC2} \quad \frac{\begin{array}{c} t \in T, \quad name = t^{name}, \quad s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M) \\ \text{Eval}(\sigma, g_t) \rightarrow \langle(\hat{\sigma}, N), \texttt{true}\rangle \\ \langle(\hat{\sigma}, N), a_t\rangle \rightarrow \langle(\hat{\sigma}', E), \cdot\rangle \end{array}}{\langle(\Psi, s), name\,(v_1, v_2, \ldots)\rangle \rightarrow \langle(\Psi, s, \cdot)\rangle}$$

On the other hand, if there exists no transition by the name *name*, then the fallback action a_F is executed, which is captured by the TRANSITION-FAL rule:

$$\text{TRANSITION-FAL} \quad \frac{\begin{array}{c} \forall t \in T : name \neq t^{name} \\ \sigma = (\Psi, \emptyset) \\ \langle(\hat{\sigma}, N), a_F\rangle \rightarrow \langle(\hat{\sigma}', x), \cdot\rangle, \quad x \neq E \\ \hat{\sigma}' = (\Psi', y) \end{array}}{\langle(\Psi, s), name\,(v_1, v_2, \ldots)\rangle \rightarrow \langle(\Psi', s, \cdot)\rangle}$$

Finally, if an exception is raised during the execution of the fallback action a_F, then the transition is reverted, which is captured by the TRANSITION-EXC3 rule:

$$\text{TRANSITION-EXC3} \quad \frac{\begin{array}{c} \forall t \in T : name \neq t^{name} \\ \sigma = (\Psi, \emptyset) \\ \langle(\hat{\sigma}, N), a_F\rangle \rightarrow \langle(\hat{\sigma}', E), \cdot\rangle \end{array}}{\langle(\Psi, s), name\,(v_1, v_2, \ldots)\rangle \rightarrow \langle(\Psi, s, \cdot)\rangle}$$

A.3 Operational Semantics of Supported Solidity Statements

We build on the small-step operational semantics for Solidity defined in [26], which enables us to reason about one computational step at a time. We have extended the semantics of [26] to support exceptions and return values.

We present the semantics of each supported Solidity statement as one or more rules. Each rule takes an execution state σ, an execution status $\in \{N, E, R[v]\}$, and a statement Stmt $\in \mathbb{S}$, and maps them to a new execution state, a new execution status, and a statement that remains to be executed (or $\cdot$ if no statements are left to be executed).

We start with basic rules that apply to every statement. If an exception has been raised or if a `return` statement has been executed, then no further statements should be executed, which is captured by the SKIP-EXC and SKIP-RET rules:

$$\text{SKIP-EXC} \quad \frac{}{\langle(\sigma, E), \text{Stmt}\rangle \rightarrow \langle(\sigma, E), \cdot\rangle}$$

$$\text{SKIP-RET} \quad \frac{}{\langle(\sigma, R[v]), \text{Stmt}\rangle \rightarrow \langle(\sigma, R[v]), \cdot\rangle}$$

A `return` statement changes the execution status to $R[\cdot]$, skipping all remaining statements, which is captured by the RETURN rule:

$$\text{RETURN} \quad \frac{}{\langle(\sigma, N), \texttt{return;}\rangle \rightarrow \langle(\sigma, R[\cdot]), \cdot\rangle}$$

To return a value v, a `return` Exp statement changes the execution status to $R[v]$, which is captured by the RETURN-VAL rule:

$$\text{RETURN-VAL}\quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\sigma', N), v\rangle}{\langle(\sigma, N), \texttt{return Exp;}\rangle \rightarrow \langle(\sigma', R[v]), \cdot\rangle}$$

If an exception is raised during the evaluation of $\text{Eval}(\sigma, \text{Exp})$, then execution status is changed to E, which is captured by the RETURN-EXC rule:

$$\text{RETURN-EXC}\quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\sigma', E), v\rangle}{\langle(\sigma, N), \texttt{return Exp;}\rangle \rightarrow \langle(\sigma', E), \cdot\rangle}$$

A compound statement (i.e., a list of statements enclosed in braces { and }) is executed by executing inner statements one after another, which is captured by the COMPOUND rule:

$$\text{COMPOUND}\quad \frac{\begin{array}{c}\langle(\sigma, N), \text{Stmt}_1\rangle \rightarrow \langle(\sigma_1, x_1), \cdot\rangle \\ \langle(\sigma_1, x_1), \text{Stmt}_2\rangle \rightarrow \langle(\sigma_2, x_2), \cdot\rangle \\ \dots \\ \langle(\sigma_{n-1}, x_{n-1}), \text{Stmt}_n\rangle \rightarrow \langle(\sigma', x), \cdot\rangle\end{array}}{\langle(\sigma, N), \{\text{Stmt}_1\ \text{Stmt}_2\ \dots\ \text{Stmt}_n\}\rangle \rightarrow \langle(\sigma', x), \cdot\rangle}$$

Loop Statements A `while` loop statement evaluates its condition Exp and if its `false`, skips the execution of the body statement Stmt, which is captured by the WHILE1 rule:

$$\text{WHILE1}\quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\sigma', N), \texttt{false}\rangle}{\langle(\sigma, N),\ \texttt{while(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle \rightarrow \langle(\sigma', N), \cdot\rangle}$$

Similarly, if the evaluation of the loop condition Exp results is an exception, then execution of the body statement Stmt is skipped, which is captured by the WHILE-EXC rule:

$$\text{WHILE-EXC}\quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\sigma', E), x\rangle}{\langle(\sigma, N),\ \texttt{while(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle \rightarrow \langle(\sigma', E), \cdot\rangle}$$

On the other hand, if the loop condition Exp is `true`, then the body statement Stmt is executed, which is captured by the WHILE2 rule:

$$\text{WHILE2}\quad \frac{\begin{array}{c}\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\hat{\sigma}, N), \texttt{true}\rangle \\ \langle(\hat{\sigma}, N), \text{Stmt}\rangle \rightarrow \langle(\hat{\sigma}', x), \cdot\rangle\end{array}}{\langle(\sigma, N),\ \texttt{while(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle \rightarrow \langle(\hat{\sigma}', x),\ \texttt{while(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle}$$

A `for` loop statement can be reduced to a `while` loop, which is captured by the FOR rule:

$$\text{FOR}\quad \frac{\langle(\sigma, N), \text{Stmt}_I\rangle \rightarrow \langle(\sigma', x), \cdot\rangle}{\begin{array}{l}\langle(\sigma, N),\ \texttt{for(}\text{Stmt}_I\,\texttt{;}\ \text{Exp}_C\,\texttt{;}\ \text{Stmt}_A\texttt{)}\ \text{Stmt}_B\rangle \\ \rightarrow \langle(\sigma', x),\ \texttt{while(}\text{Exp}_C\texttt{)}\ \ \{\text{Stmt}_B\ \text{Stmt}_A\}\rangle\end{array}}$$

Selection Statements An `if` statement is captured by the IF1, IF2, and IF-EXC rules:

$$\text{IF1}\quad \frac{\begin{array}{c}\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\hat{\sigma}, N), \texttt{true}\rangle \\ \langle(\hat{\sigma}, N), \text{Stmt}\rangle \rightarrow \langle(\hat{\sigma}', x), \cdot\rangle\end{array}}{\langle(\sigma, N),\ \texttt{if(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle \rightarrow \langle(\hat{\sigma}', x), \cdot\rangle}$$

$$\text{IF2}\quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle(\hat{\sigma}, N), \texttt{false}\rangle}{\langle(\sigma, N),\ \texttt{if(}\text{Exp}\texttt{)}\ \text{Stmt}\rangle \rightarrow \langle(\hat{\sigma}, N), \cdot\rangle}$$

$$\text{IF-EXC}\quad \frac{\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\hat{\sigma},E),x\rangle}{\langle(\sigma,N),\ \texttt{if}\,\texttt{(}\mathrm{Exp}\texttt{)}\,\mathrm{Stmt}\rangle \rightarrow \langle(\sigma,E),\cdot\rangle}$$

Similarly, an `if ... else` statement is captured by three rules, IFELSE1, IFELSE2, and IFELSE-EXC:

$$\text{IFELSE1}\quad \frac{\begin{array}{c}\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\hat{\sigma},N),\texttt{true}\rangle\\ \langle(\hat{\sigma},N),\mathrm{Stmt}_1\rangle \rightarrow \langle(\hat{\sigma}',x),\cdot\rangle\end{array}}{\langle(\sigma,N),\ \texttt{if}\,\texttt{(}\mathrm{Exp}\texttt{)}\,\mathrm{Stmt}_1\ \texttt{else}\ \mathrm{Stmt}_2\rangle \rightarrow \langle(\hat{\sigma}',x),\cdot\rangle}$$

$$\text{IFELSE2}\quad \frac{\begin{array}{c}\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\hat{\sigma},N),\texttt{false}\rangle\\ \langle(\hat{\sigma},N),\mathrm{Stmt}_2\rangle \rightarrow \langle(\hat{\sigma}',x),\cdot\rangle\end{array}}{\langle(\sigma,N),\ \texttt{if}\,\texttt{(}\mathrm{Exp}\texttt{)}\,\mathrm{Stmt}_1\ \texttt{else}\ \mathrm{Stmt}_2\rangle \rightarrow \langle(\hat{\sigma}',x),\cdot\rangle}$$

$$\text{IFELSE-EXC}\quad \frac{\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\hat{\sigma},E),x\rangle}{\langle(\sigma,N),\ \texttt{if}\,\texttt{(}\mathrm{Exp}\texttt{)}\,\mathrm{Stmt}_1\ \texttt{else}\ \mathrm{Stmt}_2\rangle \rightarrow \langle(\sigma,E),\cdot\rangle}$$

Miscellaneous Statements An expression statement is captured by the EXPRESSION rule:

$$\text{EXPRESSION}\quad \frac{\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\sigma',x),v\rangle}{\langle(\sigma,N),\mathrm{Exp}\texttt{;}\rangle \rightarrow \langle(\sigma',x),\cdot\rangle}$$

A variable declaration statement is captured by the VARIABLE or VARIABLE-ASG rule:

$$\text{VARIABLE}\quad \frac{\mathrm{Decl}(\sigma,\mathrm{Type},\mathrm{Name}) \rightarrow \langle(\sigma',x)\rangle}{\langle(\sigma,N),\mathrm{Type\ Name}\texttt{;}\rangle \rightarrow \langle(\sigma',x),\cdot\rangle}$$

$$\text{VARIABLE-ASG}\quad \frac{\mathrm{Eval}(\sigma,\mathrm{Exp}) \rightarrow \langle(\sigma',x),v\rangle}{\begin{array}{l}\langle(\sigma,N),\mathrm{Type}\ \ \mathrm{Name}\ \texttt{=}\ \mathrm{Exp}\texttt{;}\rangle\\ \quad\rightarrow \langle(\sigma',x),\texttt{\{}\ \ \mathrm{Type\ Name}\texttt{;}\ \ \mathrm{Name}\ \texttt{=}\ v\texttt{;}\ \ \texttt{\}}\rangle\end{array}}$$

where $\mathrm{Decl}(\sigma,\mathrm{Type},\mathrm{Name})$ introduces a variable into the namespace (and extends memory when necessary for memory-type variables).

An event statement is captured by the EVENT and EVENT-EXC rules:

$$\text{EVENT}\quad \frac{\begin{array}{c}\mathrm{Eval}(\sigma,\mathrm{Exp}_1) \rightarrow \langle(\sigma_1,N),v_1\rangle\\ \dots\\ \mathrm{Eval}(\sigma_{n-1},\mathrm{Exp}_n) \rightarrow \langle(\sigma_n,N),v_n\rangle\\ \mathrm{Log}(\sigma_n,(\mathit{name},v_1,\dots,v_n)) \rightarrow (\sigma',N)\end{array}}{\langle(\sigma,N),\ \texttt{emit}\ \mathit{name}\,\texttt{(}\mathrm{Exp}_1,\dots,\mathrm{Exp}_n\texttt{);}\rangle \rightarrow \langle(\sigma',x),\cdot\rangle}$$

$$\text{EVENT-EXC}\quad \frac{\begin{array}{c}\mathrm{Eval}(\sigma,\mathrm{Exp}_1) \rightarrow \langle(\sigma_1,x_1),v_1\rangle\\ \dots\\ \mathrm{Eval}(\sigma_{n-1},\mathrm{Exp}_n) \rightarrow \langle(\sigma_n,x_n),v_n\rangle\\ \mathrm{Log}(\sigma_n,(\mathit{name},v_1,\dots,v_n)) \rightarrow (\sigma',y)\\ x_1 = E \vee \dots \vee x_n = E \vee y = E\end{array}}{\langle(\sigma,N),\ \texttt{emit}\ \mathit{name}\,\texttt{(}\mathrm{Exp}_1,\dots,\mathrm{Exp}_n\texttt{);}\rangle \rightarrow \langle(\sigma',x),\cdot\rangle}$$

where Log records the specified values on the blockchain.

B Templates and CTL for Property Specification

B.1 Background on CTL

For the specification of properties, we use Computation Tree Logic (CTL). We only provide a brief overview, referring the reader to the classic textbook [4] for a complete and formal presentation. CTL formulas specify properties of execution trees generated by transitions systems. The formulas are built from atomic predicates that represent transitions and statements of the transition system, using several operators, such as $\mathtt{EX}$, $\mathtt{AX}$, $\mathtt{EF}$, $\mathtt{AF}$, $\mathtt{EG}$, $\mathtt{AG}$ (unary) and $\mathtt{E}[\cdot\,\mathtt{U}\,\cdot]$, $\mathtt{A}[\cdot\,\mathtt{U}\,\cdot]$, $\mathtt{E}[\cdot\,\mathtt{W}\,\cdot]$, $\mathtt{A}[\cdot\,\mathtt{W}\,\cdot]$ (binary). Each operator consists of a quantifier on the branches of the tree and a temporal modality, which together define when in the execution the operand sub-formulas must hold. The intuition behind the letters is the following: the branch quantifiers are $\mathtt{A}$ (for “All”) and $\mathtt{E}$ (for “Exists”); the temporal modalities are $\mathtt{X}$ (for “neXt”), $\mathtt{F}$ (for “some time in the Future”), $\mathtt{G}$ (for “Globally”), $\mathtt{U}$ (for “Until”) and $\mathtt{W}$ (for “Weak until”). A property is satisfied if it holds in the initial state of the transition systems. For instance, the formula $\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that in *all execution branches* the predicate p must hold *up to the first state* (not including this latter), where the predicate q holds. Since we used the weak until operator $\mathtt{W}$, if q never holds, p must hold forever. As soon as q holds in one state of an execution branch, p does not need to hold anymore, even if q does not hold. On the contrary, the formula $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that the subformula $\mathtt{A}[p\,\mathtt{W}\,q]$ must hold in *all branches at all times.* Thus, p must hold whenever q does not hold, i.e., $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q] = \mathtt{AG}\,(p \vee q)$.

B.2 Templates and Corresponding CTL formulas

Tables 3 and 4 contain the full list of our natural language-like templates and their corresponding CTL formulas. We use p, q, and r for simplicity, to denote the transition and statement sets, i.e., $\langle$***Transitions*** $\cup$ ***Statements***$\rangle$.

Table 3. Safety property templates

Template	CTL formula
p can never happen after **q**	$\mathtt{AG}(\mathbf{q} \rightarrow \mathtt{AG}\ (\neg \mathbf{p}))$
p can happen only after **q**	$\mathtt{A}[\neg\ \mathbf{p}\ \mathtt{W}\ \mathbf{q}]$
if **p** happens, **q** can happen only after **r** happens	$\mathtt{AG}(\mathbf{p} \rightarrow \mathtt{AX}\ \mathtt{A}\ [\neg\ \mathbf{q}\ \mathtt{W}\ \mathbf{r}])$
p can never happen	$\mathtt{AG}(\ \neg \mathbf{p})$
p can never happen before **q**	$\mathtt{A}[\neg\ \mathbf{p}\ \mid\ \mathtt{AG}(\ \neg \mathbf{q})\ \mathtt{W}\ \mathbf{q}]$

Table 4. Liveness property templates

Template	CTL formula
p will eventually happen after **q**	$\mathtt{AG}\ (\mathbf{q} \rightarrow \mathtt{AF}\ (\mathbf{p}))$
p will eventually happen	$\mathtt{AF}(\ \mathbf{p})$

C Background

C.1 Solidity Function Calls

Nested function calls in Solidity are the reason behind several identified vulnerabilities. We briefly describe how a smart contract can call a function of another contract or delegate execution. More information can be found in the Solidity documentation [46]. Firstly, a contract can call functions defined in another contract:

- *addressOfContract*`.call(`*data*`)`: Low-level call, for which the name and arguments of the invoked function must be specified in *data* according to the Ethereum ABI. The `call` method returns Boolean `true` if the execution was successful (or if there is no contract at the specified address) and `false` if it failed (e.g., if the invoked function threw an exception).
- *contract*`.`*function*`(`*arg1*`,` *arg2*`,` *...*`)`: High-level call[10], which may return a value as output on success. If the invoked method fails (or does not exist), an exception is raised in the caller, which means that all changes made by the caller are reverted, and the exception is automatically propagated up in the call hierarchy.

If the function specified for `call` does not exist, then the *fallback* function of the callee is invoked. The fallback function does not have a name[11] and arguments, and it cannot return anything. A contract can have *at most one* fallback function, and no function is executed if a fallback is not found (note that this does not constitute a failure). The fallback function is also invoked if ether[12] is sent to the contract using one of the two methods:

- *addressOfContract*`.send(`*amount*`)`: Sends the specified amount of currency to the contract, invoking its fallback function (if there exists one). If `send` fails (e.g., if the fallback function throws an exception), then it returns Boolean `false`; otherwise, it returns `true`.
- *addressOfContract*`.transfer(`*amount*`)`: Similar to `send`, but raises an exception on failure, which is handled similar to a high-level function call failure.

Finally, a contract can also "delegate" execution to another contract using *addressOfContract*`.delegatecall(`*data*`)`. Delegation is similar to a low-level `call`, but there is a fundamental difference: in this case, the function specified by *data* is executed in the context of the caller (e.g., the function will see the contract variables of the caller, not the callee). In other words, contracts may "borrow code" from other contracts using `delegatecall`, which enables the creation of libraries.

[10] Note that *contract* is a reference to a Solidity contract that is available at compile time, while *addressOfContract* is just a 160-bit address value.

[11] For ease of presentation, we will refer to the fallback function using the name "fallback" in our models.

[12] Ether is the cryptocurrency provided by the Ethereum blockchain.

C.2 Examples of Common Solidity Vulnerabilities

Here, we discuss three examples of common types of vulnerabilities in Solidity smart contracts [43,2].

Re-Entrancy When a contract calls a function in another contract, the caller is blocked until the call returns. This allows the callee, who may be malicious, to take advantage of the intermediate state in which the caller is, e.g., by invoking a function in the caller. Re-entrancy is one of the most common culprits behind vulnerabilities, and it was also exploited in the infamous "The DAO" attack [16]. In Section 3.2, we discuss how the model behind VeriSolid prevents re-entrancy.

"Denial of Service" [2] If a function involves sending ether using `transfer` or making a high-level function call to another contract, then the recipient contract can "block" the execution of this function by always throwing an exception. Such vulnerabilities can be detected with VeriSolid using a type of liveness properties (see Section 3.4), as we do for "King of Ether 2" (see Section 5.2).

Deadlocks A contract may end up in a "deadlock" state (either accidentally or through adversarial action), in which it is no longer possible to withdraw or transfer currency from the contract. This means that the currency stored in the contract is practically lost, similar to what happened to the Parity multisignature wallet contracts [36]. VeriSolid can verify if a contract model is deadlock-free, without requiring the developer to specify any property (Section 5).

C.3 Modeling and Verification with BIP and nuXmv

We recall the necessary concepts of the Behavior-Interaction-Priority (BIP) component framework [6]. BIP has been used for constructing several correct-by-design systems, such as robotic systems and satellite on-board software [42,31,7]. Systems are modeled in BIP by superposing three layers: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of components represented by transition systems. Each component transition is labeled by a *port*, which specifies the transition's unique name. Ports form the interface of a component and are used for interaction with other components. Additionally, each transition may be associated with a set of *guards* and a set of *actions*. A guard is a predicate on variables that must be true to allow the execution of the associated transition. An action is a computation triggered by the execution of the associated transition. Component interaction is described in the *interaction* and *priority* layers. We omit the explanation of these two layers, which are not used in this paper.

In order to check behavioral correctness of a system under design, formal verification is essential. While alternative approaches, such as simulation and testing, rely on the selection of appropriate test input for an adequate coverage of the program's control flow, formal verification (e.g., by model checking) guarantees full coverage of execution paths for all possible inputs. Thus, it provides a *rigorous* way to assert (or deny) that a system model meets a set of properties.

In VeriSolid, we verify deadlock-freedom using the state space exploration analysis provided by BIP. This analysis checks deadlock by default, as it is an essential correctness property. For the verification of safety and liveness properties, we use the BIP-to-NuSMV tool[13] to translate our BIP models into NuSMV, the input language of the nuXmv symbolic model checker [9]. The developer must give as input the properties to be verified directly as temporal logic formulas or by using natural language templates provided by our tool. The template input is used to generate (*Computation Tree Logic*) CTL specifications which are checked by the nuXmv tool. If a property is violated, the user gets a counterexample transition sequence that exemplifies the violation. Counterexamples help the user to locate the error back to the input model and identify its cause. The correctness of the BIP-to-NuSMV transformation based on bi-simulation was proved by Noureddine et al. [38].

D Augmentation Algorithms and Equivalence Proof

D.1 Conformance Transformation

First, we introduce Algorithm 1 for replacing the fallback and initial actions, which model the fallback function and constructor of a Solidity contract, with functionally equivalent transitions. Since the fallback function may be called in any state, the algorithm adds to each state a transition that does not change the state and whose action is the fallback action. Then, the algorithm adds a new initial state and a transition from the new to the original initial state, whose action is the initial action.

Algorithm 1 *Conformance*$(D, S, S_F, s_0, a_0, a_F, V, T)$

```
Input: model $(D, S, S_F, s_0, a_0, a_F, V, T)$
Result: model $(D, S, S_F, s_0, V, T)$
for state $s \in S$ do
    add transition from $s$ to $s$ with action $a_F$
end for
add state $s_I$
add transition from $s_I$ to $s_0$ with action $a_I$
change initial state $s_0 := s_I$
```

D.2 Augmentation Transformation

Next, we introduce algorithms for translating a model with compound, selection, loop, etc. statements into a model with only variable declaration and expression

[13] `http://risd.epfl.ch/bip2nusmv`

statements. We first describe Algorithm 2, which translates a single transition with an arbitrary statement into a set of states and internal transitions with only variable declaration, expression, and return statements. Then, we describe Algorithm 3, which translates an entire model with the help of Algorithm 2. Our augmentation algorithms are based on the small-step operational semantics of our supported Solidity Statements provided in Appendix A.3.

Algorithm 2, called *AugmentStatement*, takes as input a Solidity statement, an origin, destination, and return state, and it creates a set of states and transitions that implement the input statement using only variable declaration, expression, and return statements as actions. Note that before invoking this algorithm, Algorithm 3 removes the original transition between the origin and destination states; hence, this algorithm creates all transitions (and states) from scratch. If the statement is a variable declaration, event, or expression statement, then the algorithm simply creates a transition from the origin to the destination state without any guards and having the statement as an action. If the statement is a return statement, then it creates a transition from the origin to the return state. Note that the return state is preserved by all recursive calls to *AugmentStatement*, and it is initialized with the destination of the original transition by Algorithm 3.

If the statement is a compound, selection, or loop statement, Algorithm 2 creates a set of states and transitions. For a *compound statement* (i.e., list of statements), the algorithm creates a set of new states, each of which corresponds to the execution stage after an inner statement (except for the last one), and it invokes itself (i.e., *AugmentStatement*) for each inner statement. For a *selection statement* with an `else` (i.e., false) branch, it creates two states, which correspond to the true and false branches. Then, it creates transitions to these states with the branch condition and its negation as guards, and invokes itself for both the true and false body statements. If the selection statement does not have an `else` branch, then the false branch is replaced by a simple transition to the destination state with the negation of the condition as a guard. Finally, given a `for` *loop statement*, it creates three states, which model three stages of the loop execution: after initialization, after each time the loop condition is evaluated to true, and after each execution of the body. Then, it invokes itself with the initialization statement, creates transitions with the loop condition and its negation (leading to the second state or the destination state), and then completes the loop by invoking itself for the body and afterthought statements. For a `while` loop statement, it needs to create only one new state since there is no initialization or afterthought statement.

Algorithm 3, called *AugmentModel*, takes as input a model that can have any set of supported statements as actions, and it translates the model into one that has only variable declaration, expression, and return statements. It does so by iterating over the transitions and replacing each transition with a set of states and transitions using Algorithm 2. Furthermore, it also augments the transition to consider the possibility that the transition is *reverted* due to an exception (e.g., failure of a high-level function call or `transfer`). More specifically, for each

Algorithm 2 *AugmentStatement*(a, s_o, s_d, s_r)

```
Input: statement $a$, origin state $s_o$, destination state $s_d$, return state $s_r$
if $a$ is variable declaration statement $\vee$ $a$ is event statement $\vee$ $a$ is expression statement then
    add transition from $s_o$ to $s_d$ with action $a$
else if $a$ is return statement then
    add transition from $s_o$ to $s_r$ with action $a$
else if $a$ is compound statement $\{a_1; a_2; \ldots; a_N\}$ then
    for $i = 1, 2, \ldots, N-1$ do
        add state $s_i$
    end for
    $AugmentStatement(a_1, s_o, s_1, s_r)$
    for $i = 2, 3, \ldots, N-1$ do
        $AugmentStatement(a_i, s_{i-1}, s_i, s_r)$
    end for
    $AugmentStatement(a_N, s_{N-1}, s_d, s_r)$
else if $a$ is selection statement if ($c$) $a_T$ else $a_F$ then
    add state $s_T$
    add transition from $s_o$ to $s_T$ with guard $c$
    $AugmentStatement(a_T, s_T, s_d, s_r)$
    add state $s_F$
    add transition from $s_o$ to $s_F$ with guard $!(c)$
    $AugmentStatement(a_F, s_F, s_d, s_r)$
else if $a$ is selection statement if ($c$) $a_T$ then
    add state $s_T$
    add transition from $s_o$ to $s_T$ with guard $c$
    $AugmentStatement(a_T, s_T, s_d, s_r)$
    add transition from $s_o$ to $s_d$ with guard $!(c)$
else if $a$ is loop statement for ($a_I$; $c$; $a_A$) $a_B$ then
    add states $s_I, s_C, s_B$
    $AugmentStatement(a_I, s_o, s_I, s_r)$
    add transition from $s_I$ to $s_d$ with guard $!(c)$
    add transition from $s_I$ to $s_C$ with guard $c$
    $AugmentStatement(a_B, s_C, s_B, s_r)$
    $AugmentStatement(a_A, s_B, s_I, s_r)$
else if $a$ is loop statement while ($c$) $a_B$ then
    add state $s_L$
    add transition from $s_o$ to $s_d$ with guard $!(c)$
    add transition from $s_o$ to $s_L$ with guard $c$
    $AugmentStatement(a_B, s_L, s_o, s_r)$
end if
```

Algorithm 3 *AugmentModel*(D, S, S_F, s_0, V, T)

```
Input: model $(D, S, S_F, s_0, V, T)$
Result: model $(D, S, S_F, s_0, V, T)$
for transition $t \in T$ do
    remove transition $t$
    add state $s_{grd}$
    add transition from $t^{from}$ to $s_{grd}$ with guard $g_t$
    if action $a_t$ cannot raise exception then
        $AugmentStatement(a_t, s_{grd}, t^{to}, t^{to})$
    else
        add transition from $s_{grd}$ to $t^{from}$ with guard "revert"
        add state $s_{rvrt}$
        add transition from $s_{grd}$ to $s_{rvrt}$ with guard "!revert"
        $AugmentStatement(a_t, s_{rvrt}, t^{to}, t^{to})$
    end if
end for
```

original transition, it first removes the transition, then adds a state s_{grd} and a transition from the origin to s_{grd} with the original guard. If the action contains a statement that can result in an exception, the algorithm also adds a state s_{rvrt}, a transition from state s_{grd} to state s_{rvrt}, and a transition from state s_{grd} to the origin state. During verification, our tool considers the possibility of the entire transition being reverted using this branch. Finally, the algorithm invokes *AugmentStatement* with the original action and original destination.

D.3 Observational Equivalence Proof

Below we provide the proof of Theorem 1.

Proof. We are going to prove that all three conditions hold for some pair (q, r), for which certain criteria hold.

Before that, let us repeat a set of preliminary assumptions for the states and transitions in both systems. From the transformation algorithm, it holds that for each state q, there is exactly one corresponding state $c(q) \in S_E$, at which there can be invoked exactly the same functions as at q.

The execution semantics of a function says that α may be reverted , or that it may be executed normally (finished). There are $\alpha^{fin}, \alpha^{rev} \in A$ transitions for representing each of these cases. For each such α in the transitions of q, there is a set of outgoing paths P_a at $c(q)$, where both α and P_a represent the same execution semantics, only that paths consist of distinct transitions for each Solidity code statement in α (branching in paths is caused due to *if* and *while* constructs). Each P_a can be represented by the regular expression $\beta^{call}\beta^{\star}\alpha$, where β^{call} is the function call, each β-transition is an arbitrary code statement, and α is either α^{rev} or α^{fin}.

Fig. D.3 shows a state $q \in S_I$ (bottom) with two transitions α^{rev} and α^{fin} and its corresponding $r = c(q) \in S_E$ (top) with the outgoing $P_{\alpha^{rev}}$ and

$P_{\alpha^{fin}}$. We will prove the relationship R denoted by the dotted lines, i.e., that $(q, r), (q, r1), (q, r2), (q, 3), (q', r') \in R$ for each such $\alpha \in A$. In other words, if r is correspondent to q, then it is equivalent with q and all the other r_i that are reachable in the path, are also equivalent with q, except for r', which is equivalent with q'. If we prove this for one $\langle q, r, \alpha \rangle$ tuple, then it holds for all of them.

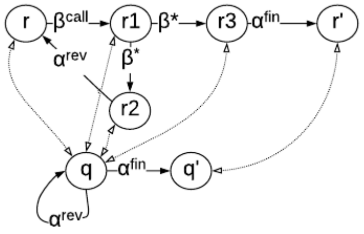

Fig. 7. Abstract representation of states in the smart contract (bottom) and the augmented system (top) (R is shown with dotted lines).

First, let us prove that $(q, r) \in R$. For each $\alpha^{fin} \in A$, such that $q \xrightarrow{\alpha^{fin}} q'$ there is a $P_{\alpha^{fin}}$, such that $r \xrightarrow{P_{\alpha^{fin}}} r'$ and $P_{\alpha^{fin}} = \beta * \alpha^{fin}$, where $\beta \in B$ and $\alpha^{fin} \in A$. Moreover, q' and r' are corresponding states just like q and r, thus, if $(q, r) \in R$ is proved, so is $(q', r') \in R$. As with each $\alpha^{fin} \in A$, also for each $\alpha^{rev} \in A$ there is a $P_{\alpha^{rev}}$, such that $r \xrightarrow{P_{\alpha^{rev}}} r$, where $\beta \in B$ and $\alpha^{rev} \in A$. Moreover, the final states being q and r are now being proved equivalent. So, far we have proved Property 1 for (q, r). Property 2 does not apply since there are no transitions of A starting from r. For Property 3, we have to prove that $(q, r1) \in R$, since $r1$ is the only state that is reachable from r through transitions of B. We will prove $(q, r1) \in R$ at a later step. Since the three Properties hold, $(q, r) \in R$ has been proved. Note that since $(q, r) \in R$, it follows that $(q', r') \in R$.

Let us prove now that $(q, r1) \in R$. For each $\alpha^{fin} \in A$, such that $q \xrightarrow{\alpha^{fin}} q'$ there is a $P_{\alpha^{fin}}$, such that $r1 \xrightarrow{P_{\alpha^{fin}}} r'$ and $P_{\alpha^{fin}} = \beta * \alpha^{fin}$, where $\beta \in B$ and $\alpha^{fin} \in A$. Moreover, it has been proved that $(q', r') \in R$. Similarly, for each $\alpha^{rev} \in A$ there is a $P_{\alpha^{rev}}$, such that $r1 \xrightarrow{P_{\alpha^{rev}}} r$, where $\beta \in B$, $\alpha^{rev} \in A$ and the final states q and r are equivalent. Property 2 does not apply. For Property 3, we have to prove that $(q, r2) \in R$ and $(q, r3) \in R$, since $r2$ and $r3$ are the only states that are reachable from $r1$ through transitions of B. We will prove $(q, r2) \in R$ and $(q, r3) \in R$ at a later step. Since the three Properties hold, $(q, r1) \in R$ has been proved.

Let us prove that $(q, r2) \in R$ and that $(q, r3) \in R$. For each $\alpha^{fin} \in A$, such that $q \xrightarrow{\alpha^{fin}} q'$ there is a α^{fin}, such that $r3 \xrightarrow{P_{\alpha^{fin}}} r'$ and $P_{\alpha^{fin}} \alpha^{fin}$, where $\beta \in B$ and $\alpha^{fin} \in A$. Moreover, it has been proved that $(q', r') \in R$. Similarly, for each

$\alpha^{rev} \in A$ there is a P_{α}^{rev}, such that $r2 \xrightarrow{\alpha^{rev}} r$, where $\beta \in B$, $\alpha^{rev} \in A$ and the final states q and r are equivalent. Property 2 holds since for each $\alpha^{fin} \in A$ and $\alpha^{rev} \in A$, such that $r3 \xrightarrow{\alpha^{fin}} r'$ and $r2 \xrightarrow{\alpha^{rev}} r$, there is an $\alpha^{fin} \in A$ (resp. $\alpha^{rev} \in A$) such that $q \xrightarrow{\alpha^{fin}} q'$ (resp. $q \xrightarrow{\alpha^{fin}} q$) and it has been proved that $(q', r') \in R$ (resp. $(q, r) \in R$). Property 3 does not apply, since there are no states that are reachable from $r2$ or $r3$ through transitions of B. Since the three Properties hold, $(q, r2) \in R$ and $(q, r3) \in R$ have been proved.

E Solidity Code Generation

The VeriSolid code generator is an extension of the FSolidM code generator [32]. The code generation takes as input the initial transition system modeled by the developer. To generate Solidity code, it follows directly the operational semantics of the transition system defined in Appendix A.2. We first provide an overview of the key differences between the two generators, and then present the VeriSolid code generator. We refer the reader to [32] for a detailed presentation of the FSolidM code generator.

Compared to FSolidM, the VeriSolid generator contains the following main differences:

- At the beginning of each transition, the value of the state variable `state` is set to `InTransition` (if the transition has a non-empty action).
- A constructor is generated from the initial action a_0.
- A fallback function is generated from the fallback action a_F.
- To maintain functional equivalence between the model and the generated code, FSolidM code-generator plugins (see [32]) are not supported.

The input of the VeriSolid code generator is a smart contract that is defined (see Definition 1) as a transition system $(D, S, S_F, s_0, a_0, a_F, V, T)$. In addition, the developer specifies the *name* of the contract. Further, for each transition $t \in T$, the developer specifies $t^{payable}$, which is true if the function implementing transition t should be payable and false otherwise.

For each contract variable or input variables (i.e., function argument) $v \in \mathbb{I} \times \mathbb{T}$, we let $name(v) \in \mathbb{I}$ and $type(v) \in \mathbb{T}$ denote the name and type of the variable, respectively. We use `fixed-width` font for generated code, and we use and *italic* font for elements that are replaced with input or specified later.

$$\begin{array}{rl}
\mathit{Contract} ::= & \texttt{contract}\ \mathit{name}\ \texttt{\{} \\
& \quad \mathit{StatesDefinition} \\
& \quad \mathit{VariablesDefinition} \\
& \quad \mathit{Constructor} \\
& \quad \mathit{Fallback} \\
& \quad \mathit{Transition}(t_1) \\
& \quad \ldots \\
& \quad \mathit{Transition}(t_{|T|}) \\
& \texttt{\}}
\end{array}$$

where $\{t_1, \ldots, t_T\}$ is the set of transitions T.

$$\begin{array}{rl}
\mathit{StatesDefinition} ::= & \texttt{enum States \{} \\
& \quad \texttt{InTransition,}\ s_0\texttt{,}\ \ldots\texttt{,} s_{|S|-1} \\
& \texttt{\}} \\
& \texttt{States private state;}
\end{array}$$

where $\{s_0, \ldots, s_{|S|-1}\}$ is the set of states S.

$$\begin{array}{rl}
\mathit{VariablesDefinition} ::= & D \\
& \mathit{type}(v_1)\ \mathit{name}(v_1)\texttt{;} \\
& \ldots \\
& \mathit{type}(v_{|V|})\ \mathit{name}(v_{|V|})\texttt{;} \\
& \texttt{uint private creationTime = now;}
\end{array}$$

where D is the set of custom event and type definitions, and $\{v_1, \ldots,\ v_{|V|}\}$ is the set of contract variables V.

$$\begin{array}{rl}
\mathit{Constructor} ::= & \texttt{constructor () public \{} \\
& \quad \mathit{Action}(a_0, \texttt{States.}s_0) \\
& \quad \texttt{state = States.}s_0\texttt{;} \\
& \texttt{\}}
\end{array}$$

where s_0 and a_0 are the initial state and action, respectively.

$$
\begin{array}{rl}
\mathit{Fallback} ::= & \texttt{function () payable public \{} \\
& \quad \texttt{State memory currentState = state;} \\
& \quad \mathit{Action}(a_F, \texttt{currentState}) \\
& \quad \texttt{state = currentState;} \\
& \texttt{\}}
\end{array}
$$

where a_F is the fallback action.

$$
\begin{array}{rl}
\mathit{Transition}(t) ::= & \texttt{function } t^{name}\texttt{(}\mathit{type}(i_1)\ \mathit{name}(i_1)\texttt{,} \\
& \qquad \ldots\texttt{, } \mathit{type}\left(i_{|t^{input}|}\right)\ \mathit{name}\left(i_{|t^{input}|}\right)\texttt{)} \\
& \qquad \texttt{public } \mathit{Payable}(t)\ \mathit{Returns}(t)\ \texttt{\{} \\
& \quad \texttt{require(state == States.}t^{from}\texttt{);} \\
& \quad \texttt{require (}g_t\texttt{);} \\
& \quad \mathit{Action}(a_t, \texttt{States.}t^{to}) \\
& \quad \texttt{state = States.}t^{to}\texttt{;} \\
& \texttt{\}}
\end{array}
$$

where t^{name} is the name of transition t, $\left\{i_1, \ldots, i_{|t^{input}|}\right\}$ is the set of parameter variables (i.e., arguments) t^{input}, g_t and a_t are the guard and action, and t^{from} and t^{to} are the source and destination states.

If $t^{payable}$ is true, then $\mathit{Payable}(t) ::= \texttt{payable}$; otherwise, $\mathit{Payable}(t)$ is empty. If return type is $t^{output} = \emptyset$, then $\mathit{Returns}(t)$ is empty. Otherwise, it is $\mathit{Returns}(t) ::=$ `returns (`t^{output}`)`

If $a = \emptyset$ (i.e., empty action statement), then $\mathit{Action}(a, s)$ is empty. Otherwise,

$$
\begin{array}{rl}
\mathit{Action}(a, s) ::= & \texttt{state = States.InTransition;} \\
& \mathit{SafeAction}(a, s)
\end{array}
$$

Finally, $\mathit{SafeAction}(a, s)$ simply means a, but replacing any

$$\texttt{return } \mathit{expression};$$

or

$$\texttt{return;}$$

statement with a

$$\texttt{\{ state = } s\texttt{; return } \mathit{expression}\texttt{; \}}$$

or

$$\texttt{\{ state = } s\texttt{; return; \}}$$

compound statement in a. Note that this applies to all inner statements within a (body statements within selection statements, loop statements, etc.).

F Blind Auction

F.1 Complete Augmented Model

Figure 8 presents the complete augmented model of the Blind Auction Contract.

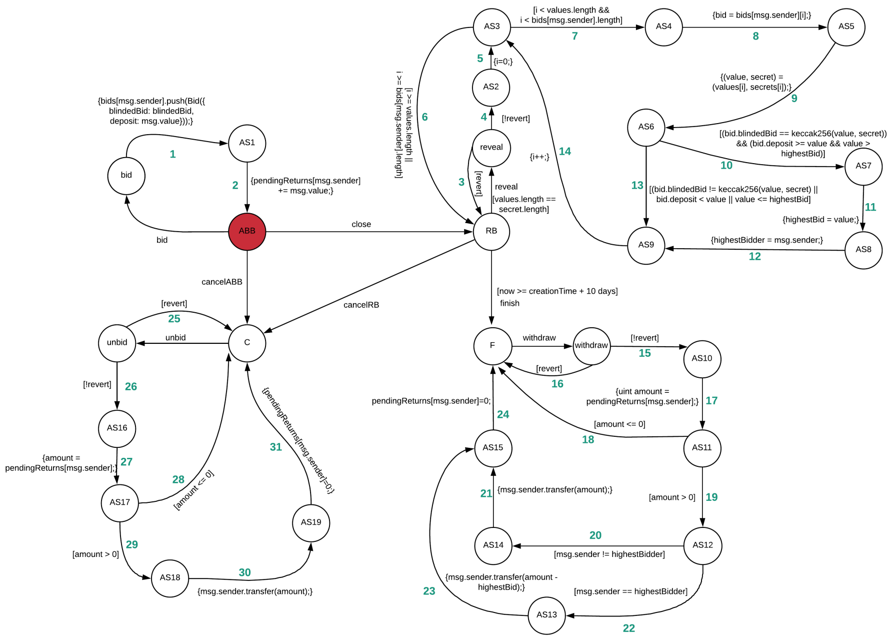
AS3
[i < values.length &&
i < bids[msg.sender].length]
7
AS4
{bid = bids[msg.sender][i];}
8
AS5
5
{i=0;}
AS2
[i >= values.length ||
i >= bids[msg.sender].length]
6
4
[!revert]
{(value, secret) =
(values[i], secrets[i]);}
9
AS6
{bids[msg.sender].push(Bid({
blindedBid: blindedBid,
deposit: msg.value}));}
AS1
1
reveal
{i++;}
14
[(bid.blindedBid == keccak256(value, secret))
&& (bid.deposit >= value && value >
highestBid)]
10
AS7
bid
2
{pendingReturns[msg.sender]
+= msg.value;}
3
[revert]
reveal
[values.length ==
secret.length]
13
[(bid.blindedBid != keccak256(value, secret) ||
bid.deposit < value || value <= highestBid]
11
{highestBid = value;}
ABB
close
RB
AS9
{highestBidder = msg.sender;}
12
AS8
bid
cancelABB
[now >= creationTime + 10 days]
finish
cancelRB
[revert]
25
unbid
unbid
C
F
withdraw
withdraw
[!revert]
15
AS10
[revert]
16
[!revert]
26
pendingReturns[msg.sender]=0;
24
{uint amount =
pendingReturns[msg.sender];}
17
[amount <= 0]
18
AS16
{amount =
pendingReturns[msg.sender];}
27
31
{pendingReturns[msg.sender]=0;}
AS15
AS11
28
[amount <= 0]
AS17
21
{msg.sender.transfer(amount);}
[amount > 0]
19
AS19
29
[amount > 0]
20
AS14
[msg.sender != highestBidder]
AS12
30
AS18
{msg.sender.transfer(amount);}
{msg.sender.transfer(amount -
highestBid);}
23
[msg.sender == highestBidder]
AS13
22

Fig. 8. Augmented model of the Blind Auction.

F.2 Solidity Code

Below we present the Solidity code generated from VeriSolid.

```
contract BlindAuction{
    //States definition
    enum States {
        InTransition,
        ABB,
        RB,
        F,
        C
    }
    States private state = States.ABB;

    //Variables definition
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }
    mapping(address => Bid[]) private bids;
    mapping(address => uint) private pendingReturns;
    address private highestBidder;
    uint private highestBid;
    uint private creationTime = now;

    //Transitions

    //Transition bid
    function bid (bytes32 blindedBid) public payable
    {
        require(state == States.ABB);
        //State change
        state = States.InTransition;
        //Actions
        bids[msg.sender].push(Bid({
            blindedBid: blindedBid,
            deposit: msg.value
        }));
        pendingReturns[msg.sender] += msg.value;
        //State change
        states = States.ABB;
    }

    //Transition close
    function close() public
    {
        require(state == States.ABB);
        //Guards
        require(now >= creationTime + 5 days);
        //State change
        state = States.RB;
    }

    //Transition reveal
    function reveal(uint[] values, bytes32[] secrets) public
    {
        require(state == States.RB);
        //Guards
        require(values.length == secrets.length);
        //State change
        state = States.InTransition;
        //Actions
        for (uint i = 0; i < values.length &&
                i < bids[msg.sender].length; i++) {
            var bid = bids[msg.sender][i];
            var (value, secret) = (values[i], secrets[i]);
            if (bid.blindedBid == keccak256(value, secret) &&
                    bid.deposit >= value &&
                    value > highestBid) {
                highestBid = value;
                highestBidder = msg.sender;
            }
        }
        //State change
        state = States.RB;
    }

    //Transition finish
    function finish() public
    {
        require(state == States.RB);
        //Guards
        require(now >= creationTime + 10 days);
        //State change
        state = States.F;
    }

    //Transition cancelABB
    function cancelABB() public
    {
        require(state == States.ABB);
        //State change
        state = States.C;
    }

    //Transition cancelRB
    function cancelRB() public
    {
        require(state == States.RB);
        //State change
        state = States.C;
    }

    //Transition withdraw
    function withdraw() public
    {
        require(state == States.F);
        //State change
        state = States.InTransition;
        // Actions
        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;
        }
        //State change
        state = States.F;
    }

    //Transition unbid
    function unbid() public
    {
        require(state == States.C);
        //State change
        state = States.InTransition;
        //Actions
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            msg.sender.transfer(amount);
            pendingReturns[msg.sender] = 0;
        }
        //State change
        state = States.C;
    }
}
```

G Further Example Models

G.1 DAO Model

The DAO contracts implemented a crowd-funding platform, which raised approximately $150 million before being attacked in June 2016. Here, we present a simplified version of the DAO contract, which allows participants to `donate` ether to fund contracts, while contracts can then `withdraw` their funds. The augmented model of the contract is presented in Figure 9.

By verifying the safety property presented in Table 1, we can guarantee that none of the two attacks presented in [2] can be successful on our contract. Both of these attacks are possible if the contract sends the amount of ether before decreasing the credit and in the meantime an attacker makes another function call, e.g., to `withdraw`. Although the former is true for our transition system, i.e., transition 6 happens after transition 5, by-design our contract changes state when the `withdraw` function is called. In particular, our contract goes from the `Initial` state to the `withdraw` state and thus, after executing transition 5, the attacker cannot make another function call. In other words, 6 will always happen right after the execution of 5.

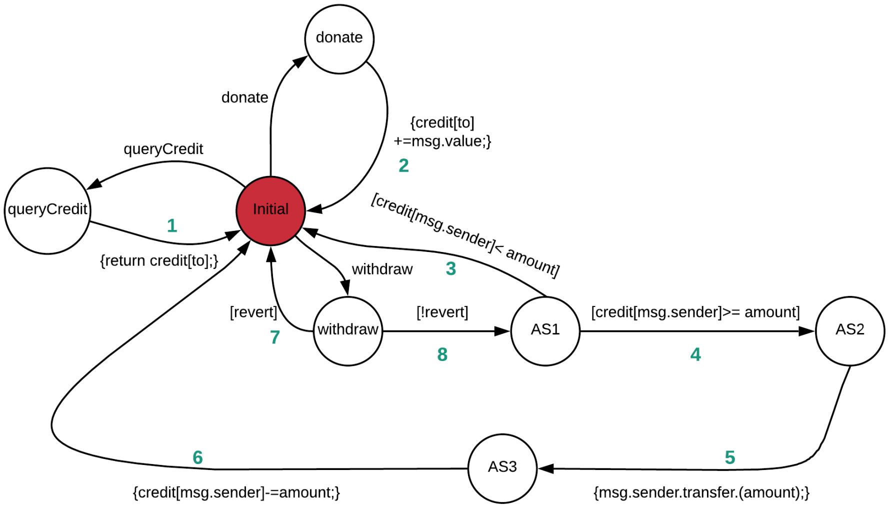

Fig. 9. Simplified model of the DAO contract.

G.2 The King Of the Ether Throne Models

The "King of the Ether Throne" is a game where players compete for acquiring the title of the King. If someone wishes to be the king, he must pay an amount of ether (which increases monotonically) to the current king. In Figures 10 and 11, we present the models of two versions of the King of the Ether Throne contract [3].

The denial of service vulnerability can be exploited in these contracts. To see why, consider an attacker Mallory, whose `fallback` just throws an exception. The adversary sends the right amount of ether, so that Mallory becomes the new king. Now, nobody else can get her crown, since every time the King of the Either Throne contract (either of the two versions) tries to send the compensation to Mallory, her fallback throws an exception, preventing the coronation to succeed. In particular, "King of Ether 1" uses `call` which is going to return `false`, while "King of Ether 2" uses `transfer` that is going to be `reverted`. We were able to check that our models have this denial of service vulnerability by model checking the liveness properties presented in Table 1.

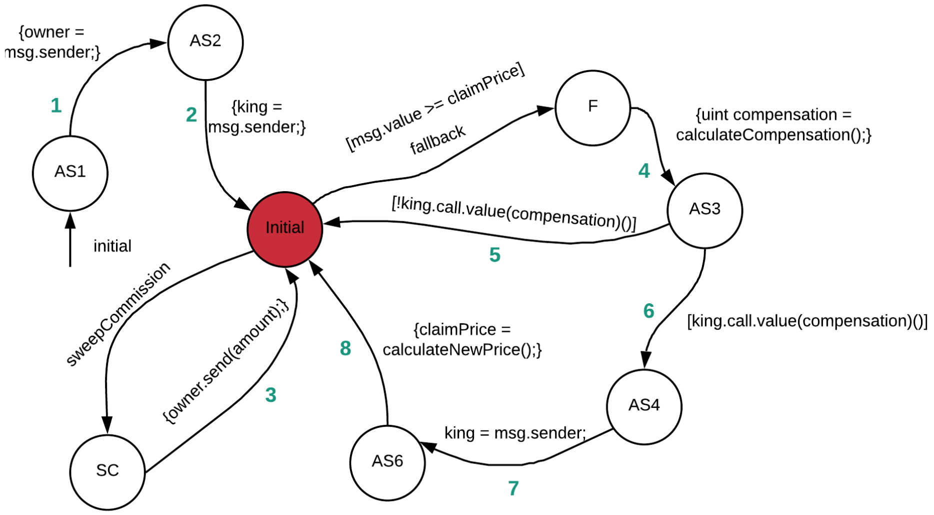

Fig. 10. King of Ether 1.

G.3 Resource Allocation Contract

TRANSAX is a blockchain-based platform for trading energy futures [28]. The core of this platform is a smart contract that allows energy producers and consumers to post offers for selling and buying energy. Since optimally matching selling offers with buying offers can be very expensive computationally, the contract relies on external solvers to compute and submit solutions to the matching problem, which are then checked by the contract.

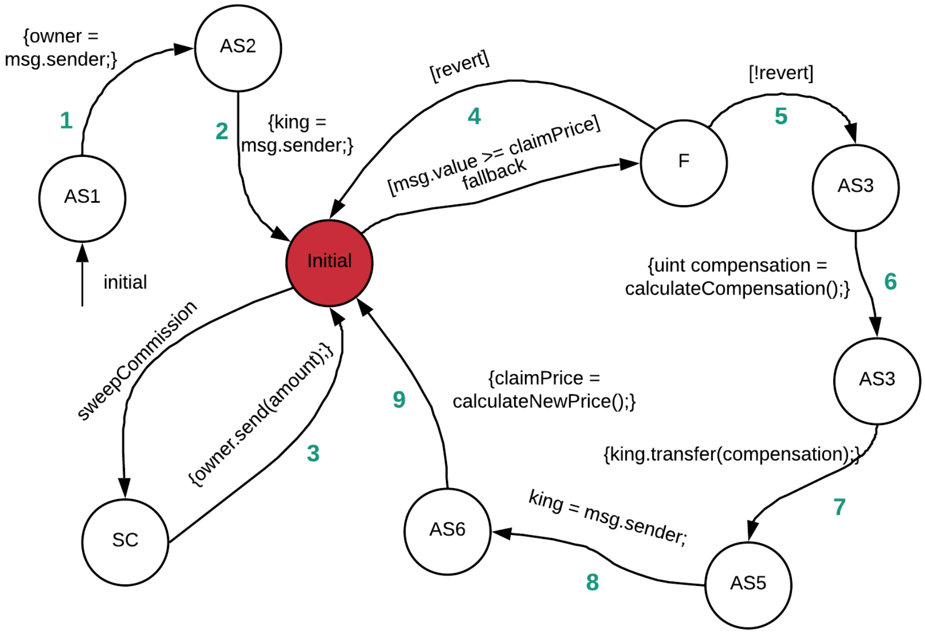

Fig. 11. King of Ether 2.

We defined a set of safety properties for this contract (Table 5 presents a subset of these properties). We were able to find a bug in the action of the `finalize` transition:

```
// action of finalize transition
if (solutions.length > 0) {
    Solution storage solution = solutions[bestSolution];
    for (uint64 i = 0; i < solution.numTrades; i++) {
        Trade memory trade = solution.trades[i];
        emit TradeFinalized(trade.sellingOfferID,
        trade.buyingOfferID, trade.power, trade.price);
    }
    solutions.length = 0;
    offers.length = 0;
}
// offers.length = 0; SHOULD HAVE BEEN HERE
cycle += 1;
```

This bug was immediately detected as a violation of our first safety property shown in Table 5.

Table 5. Analyzed properties and verification results for the Resource Allocation case study.

Case Study	Properties	Type	Result
Resource Allocation states: 3487	(i) if `close` happens, `postSellingOffer` or `postBuyingOffer` can happen only after `finalize.offers.length=0`	Safety	Violated
	(ii) `register.prosumers[msg.sender]= prosumerID` cannot happen after `setup`	Safety	Verified
	(iii) `register` cannot happen after `setup`	Safety	Verified
	(iv) if `finalize` happens `createSolution` or `addTrade` can happen only after `close`	Safety	Verified

H Extended Related Work

Vulnerability Types: Motivated by the large number of smart-contract vulnerabilities, multiple research efforts investigate and establish taxonomies of common security vulnerabilities. Atzei et al. provide a comprehensive taxonomy of Ethereum smart-contract vulnerabilities, which identifies twelve common types [2]. They show for nine of these types how an adversary can steal assets or inflict damage by exploiting a vulnerability. In another effort, Luu et al. discuss four vulnerability types—which are also identified in [2]—and they propose various techniques for mitigating them [29].

Verification and Vulnerability Discovery Both verification and vulnerability discovery are considered in the literature for identifying smart-contract vulnerabilities. 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. Parizi et al. provide a survey and comparison of existing tools for automatic security testing of smart contracts [40].

For example, Hirai performs a formal verification of a smart contract that is used by the Ethereum Name Service [22]. However, this verification proves only one particular property and it involves relatively large amount of manual analysis. In later work, Hirai defines the complete instruction set of the Ethereum Virtual Machine (EVM) in Lem, a language that can be compiled for interactive theorem provers [23]. Using this definition, certain safety properties can be proven for existing contracts. Atzei et al. propose a formal model of Bitcoin transactions, which enables formal reasoning, and they prove well-formedness properties of the Bitcoin blockchain [3]. Bhargavan et al. outline a framework for verifying the safety and correctness of Ethereum contracts [8]. The framework is built on tools for translating Solidity and EVM bytecode contracts into

F^*, a functional programming language aimed at program verification. Using the F^* representations, the framework can verify the correctness of the Solidity-to-bytecode compilation and detect certain vulnerable patterns. Tsankov et al. introduce a security analyzer for Ethereum contracts, called Securify [49]. To analyze a contract, Securify first symbolically encodes the dependence graph of the contract in stratified Datalog [25], and then it uses off-the-shelf Datalog solvers to check the satisfaction of properties, which can be described in a DSL.

Ellul and Pace use techniques from runtime verification to build the ContractLarva tool, which enables extending contracts to detect violations at runtime and to offer monetary reparations in response to a violation [15]. Colombo et al. also argue that dynamic analysis can be used not only to detect errors but also to recover from them, and they discuss how to extend the ContractLarva tool to this end [14].

Luu et al. provide a tool called Oyente, which can analyze smart contracts and detect certain typical security vulnerabilities [29]. They also recommend changes to the execution semantics of Ethereum, which would eliminate vulnerabilities of the four types that are discussed in their paper. However, these changes would need to be adopted by all Ethereum clients. Building on Oyente, Albert et al. introduce the EthIR framework for analyzing Ethereum bytecode [1]. EthIR can produce a rule-based representation of bytecode, which enables the application of existing analysis to infer properties of the EVM code. Nikolic et al. present the MAIAN tool for detecting three types of vulnerable contracts, called prodigal, suicidal and greedy [37]. MAIAN allows detecting trace vulnerabilities (i.e., vulnerabilities across a sequence of invocations of a contract) by analyzing smart contract bytecode. According to their findings, more than 30 thousand smart contracts deployed on the public Ethereum blockchain suffer from at least one vulnerability. 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 [18]. Based on an evaluation of all the contracts deployed on Ethereum, they find that two out of five contracts require trust in at least one third party. Brent et al. introduce a security analysis framework for Ethereum smart contracts, called Vandal, which converts EVM bytecode to semantic relations, which are then analyzed to detect vulnerabilities, which can be described in the Soufflé language [10]. Mueller presents Mythril, a security analysis tool for Ethereum smart contracts with a symbolic execution backend, which can be used to detect vulnerabilities [35]. Stortz introduces Rattle, a static analysis framework for EVM bytecode that can recover control flow graph, lift it into SSA / infinite register form, and optimize it, facilitating further analyses [48].

Formal Operational Semantics There are a number of research efforts that focus on defining formal operational semantics for the EVM bytecode and the Solidity language. Hildenbrandt et al. [21] formally define the semantics of EVM instructions in the K-framework [27] and validate them. In the same category falls the work of Grischchenko et al. [19,20], which presents a set of small-step semantics for the EVM bytecode. They formalized a large subset of their defined semantics in the F* proof assistant and validated them against the official Ethereum test suite. Additionally, they formally define security properties for smart contracts, such as call integrity and atomicity. Both research efforts were able to find ambiguities in the official EVM specification [52].

Yang and Hang [53] define big-step operational semantics for a large subset of the Solidity language. The work by Jiao et al. [26] defines small-step operational semantics for a subset of the Solidity language. Additionally, this work implements and validates the proposed semantics in the K-framework [27]. In our paper, we built on the small-step semantics defined in [26], which enables us to reason about one computational step at a time. We extended their Solidity statement semantics to support exceptions and return values.

Design Patterns and Development Bartoletti and Pompianu identify nine common design patterns in Ethereum smart contracts [5]. By studying the usage of patterns in publicly deployed contracts, they find that the most common one is a security pattern, called "authorization," which is found in 61% of all contracts. They also provide a taxonomy of Bitcoin and Ethereum contracts, dividing them into five categories based on their application domain, finding that the most common Ethereum contracts are financial and notary. Wöhrer and Zdun also study common design patterns in Ethereum smart contracts, based on Multivocal Literature Research [51]. They provide a taxonomy consisting of 18 patterns, and study which patterns appear commonly and how these patterns map to Solidity coding practices. O'Connnor introduces a typed, combinator-based, functional language, called Simplicity, for smart contracts [39]. Simplicity is not Turing complete, which may limit its applicability, but also makes it amenable to static analysis. Frantz and Nowostawski propose an approach for semi-automated translation of human-readable contract representations into computational equivalents [17]. They also identify smart contract components that correspond to real-world institutions and propose a mapping; however, they do not provide formal guarantees or security assurances for the generated code. Hu and Zhong propose a logic-based smart contract model, called Logic-SC, based on semantics and syntax of Active-U-Datalog with temporal extensions; however, they do not consider security properties [24].