Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach

Anastasia Mavridou[1] and Aron Laszka[2]

[1] Vanderbilt University
[2] University of Houston

Abstract. The adoption of blockchain-based distributed computation platforms is growing fast. Some of these platforms, such as Ethereum, provide support for implementing smart contracts, which are envisioned to have novel applications in a broad range of areas, including finance and the Internet-of-Things. However, a significant number of smart contracts deployed in practice suffer from security vulnerabilities, which enable malicious users to steal assets from a contract or to cause damage. Vulnerabilities present a serious issue since contracts may handle financial assets of considerable value, and contract bugs are non-fixable by design. To help developers create more secure smart contracts, we introduce *FSolidM*, a framework rooted in rigorous semantics for designing contracts as Finite State Machines (FSM). We present a tool for creating FSM on an easy-to-use graphical interface and for automatically generating Ethereum contracts. Further, we introduce a set of design patterns, which we implement as plugins that developers can easily add to their contracts to enhance security and functionality.

Keywords: smart contract, security, finite state machine, Ethereum, Solidity, automatic code generation, design patterns

1 Introduction

The adoption and importance of blockchain based distributed ledgers are growing fast. For example, the market capitalization of Bitcoin, the most-popular cryptocurrency, has exceeded $70 billion in 2017.[3] While the first generation of blockchain systems were designed to provide only cryptocurrencies, later systems, such as Ethereum, can also function as distributed computing platforms [1,2]. These distributed and trustworthy platforms enable the implementation smart contracts, which can automatically execute or enforce their contractual terms [3]. Beyond financial applications, blockchains are envisioned to have a wide range of applications, such as asset tracking for the Internet-of-Things [4]. Due to their unique advantages, blockchain based platforms and smart contracts are embraced by an increasing number of organizations and companies. For instance, the project HyperLedger[4], which aims to develop open-source blockchain

[3] `https://coinmarketcap.com/currencies/bitcoin/`
[4] `https://www.hyperledger.org/`

tools, is backed by major technology companies and financial firms, such as IBM, Cisco, J.P. Morgan, and Wells Fargo [5].

At the same time, smart contracts deployed in practice are riddled with bugs and security vulnerabilities. A recent automated analysis of 19,336 smart contracts deployed on the public Ethereum blockchain found that 8,333 contracts suffer from at least one security issue [6]. While not all of these issues lead to security vulnerabilities, many of them enable cyber-criminals to steal digital assets, such as cryptocurrencies. For example, the perpetrator(s) of the infamous 2016 "The DAO" attack exploited a combination of vulnerabilities to steal 3.6 million Ethers, which was worth around $50 million at the time of the attack [7]. More recently, $31 million worth of Ether was stolen due to a critical security flaw in a digital wallet contract [8]. Furthermore, malicious attackers might be able to cause damage even without stealing any assets, e.g., by leading a smart contract into a deadlocked state, which does not allow the rightful owners to spend or withdraw their assets.

Security vulnerabilities in smart contracts present a serious issue for multiple reasons. Firstly, smart contracts deployed in practice handle financial assets of significant value. For example, at the time of writing, the combined value held by Ethereum contracts deployed on the public blockchain is 12,205,760 Ethers, which is worth more than $3 billion.[5] Secondly, smart-contract bugs cannot be patched. By design, once a contract is deployed, its functionality cannot be altered even by its creator. Finally, once a faulty or malicious transaction is recorded, it cannot be removed from the blockchain ("code is law" principle [9]). The only way to roll back a transaction is by performing a hard fork of the blockchain, which requires consensus among the stakeholders and undermines the trustworthiness of the platform [10].

In practice, these vulnerabilities often arise due to the semantic gap between the assumptions contract writers make about the underlying execution semantics and the actual semantics of smart contracts [6]. Prior work focused on addressing these issues in existing contracts by providing tools for verifying correctness [9] and for identifying common vulnerabilities [6]. In this paper, we explore a different avenue by proposing and implementing *FSolidM*, a novel framework for creating secure smart contracts:

- We introduce a formal, finite-state machine (FSM) based model for smart contracts. We designed our model primarily to support Ethereum smart contracts, but it may be applied on other platforms as well.
- We provide an easy-to-use graphical editor that enables developers to design smart contracts as FSMs.
- We provide a tool for translating FSMs into Solidity code.[6]
- We provide a set of plugins that implement security features and design patterns, which developers can easily add to their model.

[5] `https://etherscan.io/accounts/c`

[6] Solidity is the most widely used high-level language for developing Ethereum contracts. Solidity code can be translated into Ethereum Virtual Machine bytecode, which can be deployed and executed on the platform.

Table 1. Common Smart-Contract Vulnerabilities and Design Patterns

Type	Common Name	FSolidM Plugin
Vulnerabilities	reentrancy [6,13]	locking (Section 5.1)
	transaction ordering [6] a.k.a. unpredictable state [13]	transition counter (Section 5.2)
Patterns	time constraint [14]	timed transitions (Section 5.3)
	authorization [14]	access control (Section 5.4)

Our tool is open-source and available online (see Section 6 for details).

The advantages of our approach, which aims to help developers create secure contracts rather than to fix existing ones, are threefold. First, we provide a formal model with clear semantics and an easy-to-use graphical editor, thereby decreasing the semantic gap and eliminating the issues arising from it. Second, rooting the whole process in rigorous semantics allows the connection of our framework to formal analysis tools [11,12]. Finally, our code generator—coupled with the plugins provided in our tool—enables developers to implement smart contracts with minimal amount of error-prone manual coding.

The remainder of this paper is organized as follows. In Section 2, we give a brief overview of related work on smart contracts and common vulnerabilities. In Section 3, we first present blind auction as a motivating example problem, which can be implemented as a smart contract, and then introduce our finite-state machine based contract model. In Section 4, we describe our FSM-to-Solidity code transformation. In Section 5, we introduce plugins that extend the contract model with additional functionality and security features. In Section 6, we describe our FSolidM tool and provide numerical results on computational cost. Finally, in Section 7, we offer concluding remarks and outline future work.

2 Related Work

2.1 Common Vulnerabilities and Design Patterns

Multiple studies investigate and provide taxonomies for common security vulnerabilities and design patterns in Ethereum smart contracts. In Table 1, we list the vulnerabilities that we address and the patterns that we implement in our framework using plugins.

Atzei et al. provide a detailed taxonomy of security vulnerabilities in Ethereum smart contracts, identifying twelve distinct types [13]. For nine vulnerability types, they show how an attacker could exploit the vulnerability to steal assets or to cause damage. Luu et al. discuss four of these vulnerability types in more detail, proposing various techniques for mitigating them (see Section 2.2) [6]. In this paper, we focus on two types of these common vulnerabilities:

– Reentrancy Vulnerability: Reentrancy is one of the most well-known vulnerabilities, which was also exploited in the infamous “The DAO” attack. In

Ethereum, when a contract calls a function in another contract, the caller has to wait for the call to finish. 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.
- Transaction-Ordering Dependence: If multiple users invoke functions in the same contract, the order in which these calls are executed cannot be predicted. Consequently, the users have uncertain knowledge of the state in which the contract will be when their individual calls are executed.

Bartoletti and Pompianu identify nine common design patterns in Ethereum smart contracts, and measure how many contracts use these patterns in practice [14]. Their results show that the two most common patterns are *authorization* and *time constraint*, which are used in 61% and 33% of all contracts, respectively. The also provide a taxonomy of Bitcoin and Ethereum contracts, dividing them into five categories based on their application domain. Based on their categorization, they find that the most common Ethereum contracts deployed in practice are financial, notary, and games.

2.2 Verification and Automated Vulnerability Discovery

Multiple research efforts attempt to identify and fix these vulnerabilities through verification and vulnerability discovery. For example, Hirai first performs a formal verification of a smart contract that is used by the Ethereum Name Service [15].[7] 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 in Lem, a language that can be compiled for interactive theorem provers [16]. Using this definition, certain safety properties can be proven for existing contracts.

Bhargavan et al. outline a framework for analyzing and verifying the safety and correctness of Ethereum smart contracts [9]. The framework is built on tools for translating Solidity and Ethereum Virtual Machine 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 as well as detect certain vulnerable patterns.

Luu et al. propose two approaches for mitigating common vulnerabilities in smart contracts [6]. First, they recommend changes to the execution semantics of Ethereum, which eliminate vulnerabilities from the four classes that they identify in their paper. However, these changes would need to be adopted by all Ethereum clients. As a solution that does not require changing Ethereum, they provide a tool called Oyente, which can analyze smart contracts and detect certain security vulnerabilities.

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 [17]. Based on an evaluation of all the contracts

[7] The Ethereum Name Service is a decentralized service, built on smart contracts, for addressing resources using human-readable names.

deployed on Ethereum, they find that two out of five contracts require trust in at least one third party.

3 Defining Smart Contracts as FSMs

Let us consider a blind auction (similar to the one presented in [18]), in which a bidder does not send her actual bid but only a hashed version of it. The bidder is also required make a deposit—which does not need to be equal to her actual bid—to prevent the bidder from not sending the money after she has won the auction. A deposit is considered valid if its value is higher than or equal to the actual bid. We consider that a blind auction has four main *states*:

1. `AcceptingBlindedBids`, in which blind bids and deposits are accepted by the contract;
2. `RevealingBids`, in which bidders reveal their bids, i.e., they send their actual bids and the contract checks whether the hash value is the same as the one provided during the `AcceptingBlindedBids` state and whether sufficient deposit has been provided;
3. `Finished`, in which the highest bid wins the auction. Bidders can withdraw their deposits except for the winner, who can withdraw only the difference between her deposit and bid;
4. `Canceled`, in which bidders can retract bids and withdraw their deposits.

Our approach relies on the following observations. Smart contracts have *states* (e.g., `AcceptingBlindedBids`, `RevealingBids`). Furthermore, contracts provide functions that allow other entities (e.g., contracts or users) to invoke *actions* and change the state of the smart contracts. Thus, smart contracts can be naturally represented by FSMs [19]. An FSM has a finite set of states and a finite set of transitions between these states. A transition forces a contract to take a set of actions if the associated conditions, which are called the *guards* of the transition, are satisfied. Since such states and transitions have intuitive meaning for developers, representing contracts as FSMs provides an adequate level of abstraction for reasoning about their behavior.

Figure 1 presents the blind auction example in the form of an FSM. For simplicity, we have abbreviated `AcceptingBlindedBids`, `RevealingBids`, `Finished`, and `Canceled` to `ABB`, `RB`, `F`, and `C`, respectively. `ABB` is the initial state of the FSM. Each transition (e.g., `bid`, `reveal`, `cancel`) is associated to a set of actions that a user can perform during the blind auction. For instance, a bidder can execute the `bid` transition at the `ABB` state to send a blind bid and a deposit value. Similarly, a user can execute the `close` transition, which signals the end of the bidding period, if the associated guard `now >= creationTime + 5 days` evaluates to true. To differentiate transition names from guards, we use square brackets for the latter. A bidder can reveal her bids by executing the `reveal` transition. The `finish` transition signals the completion of the auction, while the `cancelABB` and `cancelRB` transitions signal the cancellation of the auction. Finally, the `unbid` and `withdraw` transitions can be executed by

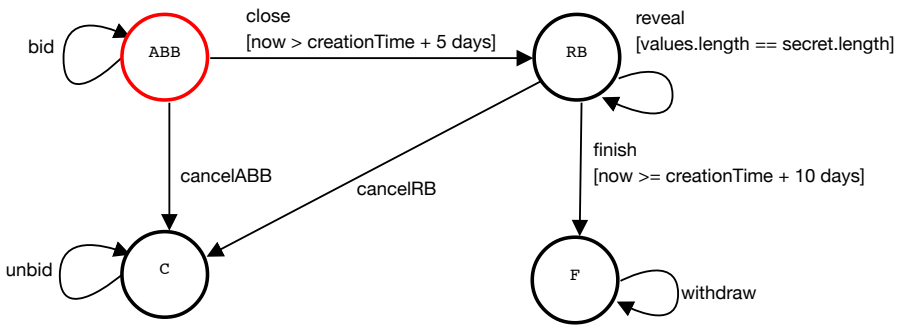

Fig. 1. Example FSM for blinded auctions.

the bidders to withdraw their deposits. For ease of presentation, we omit from Figure 1 the actions that correspond to each transition. For instance, during the execution of the `withdraw` transition, the following action is performed `amount = pendingReturns[msg.sender]`.

Guards are based on a set of variables, e.g., `creationTime`, `values`, and actions are also based on a set of variables, e.g., `amount`. These variable sets store data, that can be of type:

- *contract data*, which is stored within the contract;
- *input data*, which is received as transition input;
- *output data*, which is returned as transition output.

We denote by C, I, and O the three sets of the contract, input, and output variables of a smart contract. We additionally denote:

$\mathbb{B}[C, I]$, the set of Boolean predicates on contract and input variables;

$\mathbb{E}[C, I, O]$, the set of statements that can be defined by the full Solidity syntax.

Notice that $\mathbb{E}[C, I, O]$ represents the set of actions of all transitions. Next, we formally define a contract as an FSM.

Definition 1. *A Smart Contract is a tuple* $(S, s_0, C, I, O, \rightarrow)$*, where:*

- S *is a finite set of states;*
- $s_0 \in S$ *is the initial state;*
- C*,* I*, and* O *are disjoint finite sets of, respectively, contract, input, and output variables;*
- $\rightarrow \subseteq S \times \mathcal{G} \times \mathcal{F} \times S$ *is a transition relation, where:*
  - $\mathcal{G} = \mathbb{B}[C, I]$ *is a set of guards;*
  - $\mathcal{F}$ *is a set of action sets, i.e., a set of all ordered powersets of* $\mathbb{E}[C, I, O]$.

4 FSM-to-Solidity Transformation

To automatically generate a contract using our framework, developers can provide the corresponding FSM in a graphical form. Each transition of the FSM is

implemented as a Solidity function, where an element of $\mathcal{G}$ and a list of statements from $\mathcal{F}$ form the body. The input I and output C variables correspond to the arguments and the `return` values, respectively, of these functions. In this section, we describe the basic transformation formally, while in Section 5, we present a set of extensions, which we call *plugins*.

First, let us list the input that must be provided by the developer:

- *name*: name of the FSM;
- S: set of states;
- $s_0 \in S$: initial state;
- C: set of contract variables;
- for each contract variable $c \in C$, $access(c) \in \{\texttt{public}, \texttt{private}\}$: visibility of the variable;
- $\rightarrow$: set of transitions;
- for each transition $t \in \rightarrow$:
  - t^{name}: name of the transition;
  - $t^{guards} \in \mathcal{G}$: guard conditions of the transition;
  - $t^{input} \subseteq I$: input variables (i.e., parameters) of the transition;
  - $t^{statements} \in \mathcal{F}$: statements of the transition;
  - $t^{output} \subseteq O$: output (i.e., return values) of the transition;
  - $t^{from} \in S$: previous state;
  - $t^{to} \in S$: next state;
  - $t^{tags} \subseteq \{payable, admin, event\}$: set of transition properties specified by the developer (note that without plugins, only *payable* is supported);
- $\mathcal{T}^{custom}$: set of complex types defined by the developer in the form of `struct`s.

For any variable $v \in C \cup I \cup O$, we let $type(v) \in \mathcal{T}$ denote the domain of the variable, where $\mathcal{T}$ denotes the set of all built-in Solidity types and developer-defined `struct` types.

We use `fixed-width` font for the output generated by the transformation, and *italic* font for elements that are replaced with input or specified later. An FSM is transformed into a Solidity contract as follows:

Contract ::= `contract` *name* `{`
  StatesDefinition
  `uint private creationTime = now;`
  VariablesDefinition
  Plugins
  Transition(t_1)
  ...
  Transition($t_{|\rightarrow|}$)
`}`

where $\{t_1, \ldots, t_{|\rightarrow|}\} = \rightarrow$. Without any security extensions or design patterns added (see Section 5), *Plugins* is empty. The complete generated code for the blind-auction example presented in Figure 1 can be found in [20] (with the locking and transition-counter security-extension plugins added).

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

where $\{s_0, \ldots, s_{|S|-1}\} = S$.

Example 1. The following snippet of Solidity code presents the *StatesDefinition* code generated for the blind auction example (see Figure 1).

```
enum States {
    ABB,
    RB,
    F,
    C
}
States private state = States.ABB;
```

$$\begin{aligned} \mathit{VariablesDefinition} ::= \ & \mathcal{T}^{custom} \\ & \mathit{type}(c_1)\ \mathit{access}(c_1)\ c_1\texttt{;} \\ & \ldots \\ & \mathit{type}(c_{|C|})\ \mathit{access}(c_{|C|})\ c_{|C|}\texttt{;} \end{aligned}$$

where $\{c_1, \ldots, c_{|C|}\} = C$.

Example 2. The following snippet of Solidity code presents the *VariablesDefinition* code of the blind auction example (see Figure 1).

```
struct Bid {
    bytes32 blindedBid;
    uint deposit;
}
mapping(address => Bid[]) private bids;
mapping(address => uint) private pendingReturns;
address private highestBidder;
uint private highestBid;
```

$$\begin{aligned} \mathit{Transition}(t) ::= \ & \texttt{function } t^{name}\texttt{(}\mathit{type}(i_1)\ i_1\texttt{, } \ldots\texttt{, } \mathit{type}(i_{|t^{input}|})\ i_{|t^{input}|}\texttt{)} \\ & \quad \mathit{TransitionPlugins}(t) \\ & \quad \mathit{Payable}(t)\ \mathit{Returns}(t)\ \{ \\ & \texttt{require(state == States.}t^{from}\texttt{);} \\ & \mathit{Guards}(t) \\ & \mathit{Statements}(t) \\ & \texttt{state = States.}t^{to}\texttt{;} \\ & \} \end{aligned}$$

where $\{i_1, \ldots, i_{|t^{input}|}\} = t^{input}$. Without any security extensions or design patterns (see Section 5), *TransitionPlugins*(t) is empty, similar to *Plugins*. If $payable \in t^{tags}$, then *Payable*(t) = `payable`; otherwise, it is empty. If $t^{to} = t^{from}$ then the line `state = States.`t^{to}`;` is not generated.

If $t^{output} = \emptyset$, then *Returns*(t) is empty. Otherwise, it is as follows:

$$Returns(t) ::= \texttt{returns (}type(o_1)\ o_1\texttt{, } \ldots\texttt{, } type(o_{|t^{output}|})\ o_{|t^{output}|}\texttt{)}$$

where $\{o_1, \ldots, o_{|t^{output}|}\} = t^{output}$.

Further,

$$\begin{aligned} Guards(t) ::=&\ \texttt{require((}g_1\texttt{) \&\& (}g_2\texttt{) \&\& ... \&\& (}g_{|t^{guards}|}\texttt{));} \\ Statements(t) ::=&\ a_1 \\ &\ \ldots \\ &\ a_{|t^{statements}|} \end{aligned}$$

where $\{g_1, \ldots, g_{|t^{guards}|}\} = t^{guards}$ and $\{a_1, \ldots, a_{|t^{statements}|}\} = t^{statements}$.

Example 3. The following snippet of Solidity code shows the generated `bid` transition (see Figure 1). The `bid` transition does not have any guards and the state of the FSM does not change, i.e., it remains `ABB` after the execution of the transition.

```
// Transition bid
function bid(bytes32 blindedBid)
  payable
{
    require(state == States.ABB);
    //Actions
    bids[msg.sender].push(Bid({
        blindedBid: blindedBid,
        deposit: msg.value
    }));
}
```

Example 4. The following snippet of Solidity code shows the generated `close` transition (see Figure 1). The `close` transition does not have any associated actions but the state of the FSM changes from `ABB` to `RB` after the execution of the transition.

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

5 Security Extensions and Patterns

Building on the FSM model and the FSM-to-Solidity transformation introduced in the previous sections, we next provide extensions and patterns for enhancing the security and functionality of contracts. These extensions and patterns are implemented as plugins, which are appended to the *Plugins* and *TransitionPlugins* elements. Developers can easily add plugins to a contract (or some of its transitions) using our tool, without writing code manually.[8]

5.1 Locking

To prevent reentrancy vulnerabilities, we provide a security plugin for locking the smart contract. [9] The locking feature eliminates reentrancy vulnerabilities in a "foolproof" manner: functions within the contract cannot be nested within each other in any way.

Implementation If the locking plugin is enabled, then

$$\begin{aligned} \mathit{Plugins} \mathrel{+}= & \texttt{bool private locked = false;} \\ & \texttt{modifier locking \{} \\ & \quad \texttt{require(!locked);} \\ & \quad \texttt{locked = true;} \\ & \quad \texttt{\_;} \\ & \quad \texttt{locked = false;} \\ & \texttt{\}} \end{aligned}$$

and for every transition t,

$$\mathit{TransitionPlugins}(t) \mathrel{+}= \texttt{locking}$$

Before a transition is executed, the `locking` modifier first checks if the contract is locked. If it is not locked, then the modifier locks it, executes the transition, and unlocks it after the transition has finished. Note that the `locking` plugin must be applied before the other plugins so that it can prevent reentrancy vulnerabilities in the other plugins. Our tool always applies plugins in the correct order.

5.2 Transition Counter

Recall from Section 2.1 that the state and the values of the variables stored in an Ethereum contract may be unpredictable: when a user invokes a function (i.e., transition in an FSM), she cannot be sure that the contract does not change in

[8] Please note that we introduce an additional plugin in Appendix A.

[9] `http://solidity.readthedocs.io/en/develop/contracts.html?highlight=mutex#function-modifiers`

some way before the function is actually executed. This issue has been referred to as "transaction-ordering dependence" [6] and "unpredictable state" [13], and it can lead to various security issues. Furthermore, it is rather difficult to prevent since multiple users may invoke functions at the same time, and these function invocations might be executed in any order.

We provide a plugin that can prevent unpredictable-state vulnerabilities by enforcing a strict ordering on function executions. The plugin expects a transition number in every function as a parameter (i.e., as a transition input variable) and ensures that the number is incremented by one for each function execution. As a result, when a user invokes a function with the next transition number in sequence, she can be sure that the function is executed before any other state changes can take place (or that the function is not executed).

Implementation If the transition counter plugin is enabled, then

Plugins +=
```
uint private transitionCounter = 0;
modifier transitionCounting(uint nextTransitionNumber) {
  require(nextTransitionNumber == transitionCounter);
  transitionCounter += 1;
  _;
}
```

and for every transition t,

TransitionPlugins(t) += `transitionCounting(nextTransitionNumber)`

Note that due to the inclusion of the above modifier, t^{input}—and hence the parameter list of every function implementing a transition— includes the parameter `nextTransitionNumber` of type `uint`.

5.3 Automatic Timed Transitions

Next, we provide a plugin for implementing time-constraint patterns. We first need to extend our FSM model: a Smart Contract with Timed Transitions is a tuple $C = (S, s_0, C, I, O, \rightarrow, \overset{T}{\rightarrow})$, where $\overset{T}{\rightarrow} \subseteq S \times \mathcal{G}_T \times \mathbb{N} \times \mathcal{F}_T \times S$ is a timed transition relation such that:

- $\mathcal{G}_T = \mathbb{B}[C]$ is a set of guards (without any input data);
- $\mathbb{N}$ is the set of natural numbers, which is used to specify the time of the transition in seconds;
- $\mathcal{F}_T$ is a set of action sets, i.e., a set of all ordered powerset of $\mathbb{E}[C]$.

Notice that timed transitions are similar to non-timed transitions, but 1) their guards and assignments do not use input or output data and 2) they include a number specifying the transition time.

We implement timed transitions as a modifier that is applied to every function. When a transition is invoked, the modifier checks whether any timed transitions must be executed before the invoked transition is executed. If so, the modifier executes the timed transitions before the invoked transition.

Writing such modifiers for automatic timed transitions manually may lead to vulnerabilities. For example, a developer might forget to add a modifier to a function, which enables malicious users to invoke functions without the contract progressing to the correct state (e.g., place bids in an auction even though the auction should have already been closed due to a time limit).

Implementation For every timed transition $tt \in \overset{T}{\rightarrow}$, the developer specifies a time $tt^{time} \in \mathbb{N}$ at which the transition will automatically happen (given that the guard condition is met). This time is measured in the number of seconds elapsed since the creation (i.e., instantiation) of the contract. We let $tt_1, tt_2, \ldots$ denote the list of timed transitions in ascending order based on their specified times. When the plugin is enabled,

$$
\begin{aligned}
\mathit{Plugins} \mathrel{+}= \texttt{modifier timedTransitions } \{ \\
\quad \mathit{TimedTransition}(tt_1) \\
\quad \mathit{TimedTransition}(tt_2) \\
\quad \ldots \\
\quad \texttt{\_;} \\
\}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{TimedTransition}(t) ::= \texttt{if ((state == States.}t^{from}\texttt{)} \\
\quad \texttt{\&\& (now >= creationTime + }t^{time}\texttt{)} \\
\quad \texttt{\&\& (}\mathit{Guard}(t)\texttt{)) \{} \\
\quad \mathit{Statements}(t) \\
\quad \texttt{state = States.}t^{to}\texttt{;} \\
\}
\end{aligned}
$$

Finally, for every non-timed transition $t \in \rightarrow$, let

$$\mathit{TransitionPlugins}(t) \mathrel{+}= \texttt{timedTransitions}$$

5.4 Access Control

In many contracts, access to certain transitions (i.e., functions) needs to be controlled and restricted. [10] For example, any user can participate in a typical

[10] `http://solidity.readthedocs.io/en/develop/common-patterns.html#restricting-access`

blind auction by submitting a bid, but only the creator should be able to cancel the auction. To facilitate the enforcement of such constraints, we provide a plugin that 1) manages a list of administrators at runtime (identified by their addresses) and 2) enables developers to forbid non-administrators from accessing certain functions. This plugin implements management functions (`addAdmin`, `removeAdmin`) for only one privileged group, but it could easily be extended to support more fine-grained access control.

Implementation If the access control plugin is enabled, then

Plugins +=
```
mapping(address => bool) private isAdmin;
uint private numAdmins = 1;

function name() {
  isAdmin[msg.sender] = true;
}

modifier onlyAdmin {
  require(isAdmin[msg.sender]);
  _;
}

function addAdmin(address admin) onlyAdmin {
  require(!isAdmin[admin]);
  isAdmin[admin] = true;
  numAdmins += 1;
}

function removeAdmin(address admin) onlyAdmin {
  require(isAdmin[admin]);
  require(numAdmins > 1);
  isAdmin[admin] = false;
  numAdmins -= 1;
}
```

For transitions t such that $admin \in t^{tags}$ (i.e., transitions that are tagged "only admin" by the developer),

$$TransitionPlugins(t) \mathrel{+}= \texttt{onlyAdmin}$$

6 The FSolidM Tool

We present the FSolidM tool, which is build on top of WebGME [21], a web-based, collaborative, versioned, model editing framework. FSolidM enables collaboration between multiple users during the development of smart contracts. Changes in FSolidM are committed and versioned, which enables branching, merging, and viewing the history of a contract. FSolidM is open-source [11] and available online [12].

To use FSolidM, a developer must provide some input (see Section 4). To do so, the developer can use the graphical editor of FSolidM to specify the states, transitions, guards, etc. of a contract. The full input of the smart-contract code generator can be defined entirely through the FSolidM graphical editor. For the convenience of the developers, we have also implemented a Solidity code editor, since part of the input e.g., variable definitions and function statements, might be easier to directly write in a code editor. Figure 2 shows the two editors of the tool. We have integrated a Solidity parser[13] to check the syntax of the Solidity code that is given as input by the developers.

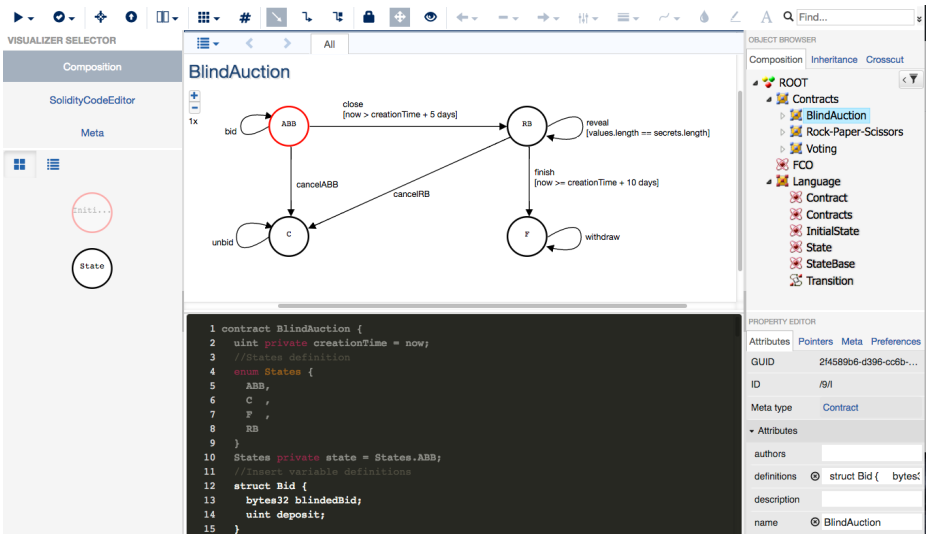

Fig. 2. The graphical and code editors provided by FSolidM.

The FSolidM code editor cannot be used to completely specify the required input. Notice that in Figure 2, parts of the code shown in the code editor are darker (lines 1-10) than other parts (lines 12-15). The darker lines of code include

[11] https://github.com/anmavrid/smart-contracts
[12] https://cps-vo.org/group/SmartContracts
[13] https://github.com/ConsenSys/solidity-parser

code that was generated from the FSM model defined in the graphical editor and are locked—cannot be altered in the code editor. The non-dark parts indicate code that was directly specified in the code editor.

FSolidM provides mechanisms for checking if the FSM is correctly specified (e.g., whether an initial state exists or not). FSolidM notifies developers of errors and provides links to the erroneous nodes of the model (e.g., a transition or a guard). Additionally, FSolidM provides an FSM-to-Solidity code generator and mechanisms for easily integrating the plugins introduced in Section 5. We present the FSolidM tool in greater detail in [20].

6.1 Numerical Results on Computational Cost

Plugins not only enhance security but also increase the computational cost of transitions. Since users must pay a relatively high price for computation performed on the public Ethereum platform, the computational cost of plugins is a critical question. Here, we measure and compare the computational cost of transitions in our blind-auction contract without and with the locking and transition counter plugins. We focus on these security feature plugins because they introduce overhead, while the design pattern plugins introduce useful functionality.

For this experiment, we use Solidity compiler version 0.4.17 with optimizations enabled. In all cases, we quantify computational cost of a transition as the gas cost of an Ethereum transaction that invokes the function implementing the transition.[14] The cost of deploying our smart contract was 504,672 gas without any plugins, 577,514 gas with locking plugin, 562,800 gas with transition counter plugin, and 637,518 gas with both plugins.[15]

Figure 3 shows the gas cost of each transition for all four combinations of the two plugins. We make two key observations. First, *computational overhead is almost constant* for both plugins and also for their combination. For example, the computational overhead introduced by locking varies between 10,668 and 10,686 gas. For the simplest transition, `unbid`, this constitutes a 54% increase in computational cost, while for the most complex transition, `reveal`, the increase is 16%. Second, the *computational overhead of the two plugins is additive*. The increase in computational cost for enabling locking, transition counter, and both are around 10,672 gas, 5,648 gas, and 16,319 gas, respectively.

7 Conclusion and Future Work

Distributed computing platforms with smart-contract functionality are envisioned to have a significant technological and economic impact in the future. However, if we are to avoid an equally significant risk of security incidents, we must ensure that smart contracts are secure. While previous research efforts focused on identifying vulnerabilities in existing contracts, we explored a different

[14] Gas measures the cost of executing computation on the Ethereum platform.

[15] At the time of writing, this cost of deployment was well below $1 (if the deployment does not need to be prioritized).

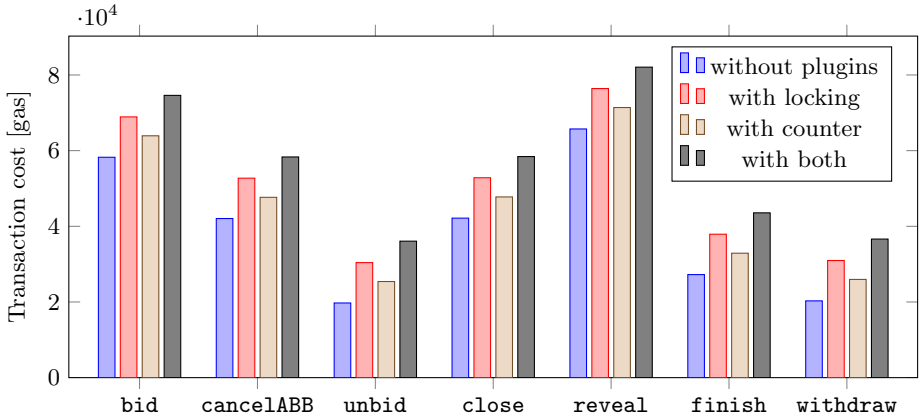

Fig. 3. Transaction costs in gas without plugins (**blue**), with locking plugin (**red**), with transition counter plugin (**brown**), and with both plugins (**dark gray**).

avenue by proposing and implementing a novel framework for creating secure smart contracts. We introduced a formal, FSM based model of smart contracts. Based on this model, we implemented a graphical editor for designing contracts as FSMs and an automatic code generator. We also provided a set of plugins that developers can add to their contracts. Two of these plugins, *locking* and *transition counter*, implement security features for preventing common vulnerabilities (i.e., reentrancy and unpredictable state). The other two plugins, *automatic timed transitions* and *access control*, implement common design patterns to facilitate the development of correct contracts with complex functionality.

We plan to extend our framework in multiple directions. First, we will introduce a number of plugins, implementing various security features and design patterns. We will provide security plugins for all the vulnerability types identified in [13] that can be addressed on the level of Solidity code. We will also provide plugins implementing the most popular design patterns surveyed in [14].

Second, we will integrate verification tools [11,12] and correctness-by-design techniques [22] into our framework. This will enable developers to easily verify the security and safety properties of their contracts. For example, developers will be able to verify if a malicious user could lead a contract into a deadlocked state. Recall that deadlocks present a serious issue since it may be impossible to recover the functionality or assets of a deadlocked contract.

Third, we will enable developers to model and verify multiple interacting contracts as a set of interacting FSMs. By verifying multiple contracts together, developers will be able to identify a wider range of issues. For example, a set of interacting contracts may get stuck in a deadlock even if the individual contracts are deadlock free.

Acknowledgements

We thank the anonymous reviewers for their invaluable suggestions and feedback.

References

1. Underwood, S.: Blockchain beyond Bitcoin. Communications of the ACM **59**(11) (2016) 15–17
2. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Technical Report EIP-150, Ethereum Project - Yellow Paper (April 2014)
3. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: Foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016)
4. Christidis, K., Devetsikiotis, M.: Blockchains and smart contracts for the internet of things. IEEE Access **4** (2016) 2292–2303
5. Vukolić, M.: Rethinking permissioned blockchains. In: Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts, ACM (2017) 3–7
6. 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), ACM (October 2016) 254–269
7. 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)
8. Qureshi, H.: A hacker stole $31m of ether – How it happened, and what it means for Ethereum. freeCodeCamp `https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce` (July 2017)
9. 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. (October 2016) 91–96
10. Leising, M.: The Ether thief. Bloomberg Markets `https://www.bloomberg.com/features/2017-the-ether-thief/` (June 2017)
11. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-Finder: A tool for compositional deadlock detection and verification. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV), Springer (2009) 614–619
12. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proceedings of the 16th International Conference on Computer Aided Verification (CAV). Volume 8559., Springer (2014) 334–342
13. 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), Springer (April 2017) 164–186
14. 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)
15. Hirai, Y.: Formal verification of deed contract in Ethereum name service. `https://yoichihirai.com/deed.pdf` (November 2016)

16. 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)
17. Fröwis, M., Böhme, R.: In code we trust? In: International Workshop on Cryptocurrencies and Blockchain Technology (CBT), Springer (September 2017) 357–372
18. Solidity by Example: Blind auction. `http://solidity.readthedocs.io/en/develop/solidity-by-example.html` Accessed on 9/5/2017.
19. Solidity Documentation: Common patterns. `http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine` Accessed on 9/5/2017.
20. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. arXiv preprint arXiv:1711.09327 (2017)
21. 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. (2014) 41–60
22. 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). (October 2016) 260–279

A Event Plugin

In this section, we introduce an additional plugin, which developers can use to notify users of transition executions. The *event plugin* uses the `event` feature of Solidity, which provides a convenient interface to the Ethereum logging facilities. If this plugin is enabled, transitions tagged with *event* emit a Solidity event after they are executed. Ethereum clients can listen to these events, allowing them to be notified when a tagged transition is executed on the platform.

Implementation If the event plugin is enabled, then

$$\begin{aligned} \mathit{Plugins} \mathrel{+}= \; & \mathit{TransitionEvent}(t_1) \\ & \mathit{TransitionEvent}(t_2) \\ & \dots \end{aligned}$$

where $\{t_1, t_2, \dots\}$ is the set of transitions with the tag *event*.

$$\begin{aligned} \mathit{TransitionEvent}(t) ::= \; & \texttt{event Event}t^{name}\texttt{;} \\ & \texttt{modifier event}t^{name}\ \texttt{\{} \\ & \quad \texttt{\_;} \\ & \quad \texttt{Event}t^{name}\texttt{();} \\ & \texttt{\}} \end{aligned}$$

For every transition t such that $\mathit{event} \in t^{tags}$ (i.e., transitions that are tagged to emit an event),

$$\mathit{TransitionPlugins}(t) \mathrel{+}= \texttt{event}t^{name}$$