Modeling and Security Verification of State-Based Smart Contracts

(article starts on next page)

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 55-28 (2022) 356–362

Modeling and Security Verification of State-Based Smart Contracts [*]

Sahar Mohajerani [*] **Wolfgang Ahrendt** [**] **Martin Fabian** [*]

[*] *Department of Electrical Engineering*
[**] *Department of Computer Science and Engineering*
Chalmers University of Technology, Göteborg, Sweden
{mohajera, ahrendt, fabian}@chalmers.se

Abstract: Smart contracts are programs that are stored on a blockchain ledger with code immutable after deployment. Thus, verifying the correct behavior of smart contracts before deployment is vital. This paper demonstrates how a security vulnerability verification in a casino smart contract can be transformed to *non-blocking* verification. To this end, the contract is first modeled as interacting extended finite state machines (EFSM), with one EFSM for each function. Modeling the security vulnerability as a condition in the EFSM system, non-blocking verification reveals the system to be blocking. Investigating the counterexample produced by the verification shows that a *transfer* that is refused by its receiver may block the casino so that all remaining funds are forever locked into the contract, thus revealing a severe vulnerability. It is then demonstrated how the same technique can show the absence of this vulnerability, by verifying that the EFSM model of an improved casino contract is indeed non-blocking.

Keywords: Extended finite state machine, smart contract, security, verification, nonblocking

1. INTRODUCTION

Smart contracts are programs that are openly stored and executed in a blockchain ecosystem, enforcing a contractual agreement between mutually distrusting users, including the exchange of crypto currencies. This successfully addresses the problem of guaranteeing the execution of agreements in a setting where parties neither have to trust each other, nor have to involve a third party (like bank, lawyer, or authority). At the same time, they provide a surface for security attacks. Even if the underlying blockchain protocols cannot feasibly be compromised, a smart contract can itself allow behavior unintended by the programmer, which may be exploited to the disadvantage of some of the users. As smart contracts cannot be changed once deployed on the blockchain, it is important to guarantee absence of such unintended behavior.

One of many security attack scenarios of smart contracts is where an attacker, which might be a malicious contract, refuses to receive a payment (Atzei et al., 2017). This can prevent progress of the contract in such a way that funds intended (by the programmer) for other users may be locked forever. If a smart contract is vulnerable to this type of attack, an attacker can cause big damage for other users, at the cost of only small damage to herself. This paper demonstrates an approach to use formal verification to discover and exclude this type of vulnerability.

To formally verify a smart contract, the behavior of the contract needs to be modeled first. High level behavior of a smart contract, which mostly addresses the interaction between different parts, can be modeled as *Extended Finite State Machines* (EFSM) (Sköldstam et al., 2007). In general, explicit states result from abstracting away execution details and intermediate results. But in addition, there is a certain programming pattern popular in smart contracts, where elements of an enumeration type effectively act as explicit states (like `IDLE`, `GAME_AVAILABLE`, `BET_PLACED` in the casino contract below). Such states determine the effect of the events that users trigger on the contract.

The works in Mohajerani et al. (2015) and Teixeira et al. (2015), among others, provide verification of non-blocking behavior of composed EFSM in the framework of *supervisory control theory* (SCT) (Ramadge and Wonham, 1989), which can be applied to a variety of application domains from manufacturing systems (Leduc et al., 2006) to autonomous cars (Zita et al., 2017). This paper describes how the SCT framework can be used to verify smart contracts. This is based on the insight that the presence or absence of the aforementioned vulnerabilities can be analysed using non-blocking verification.

Related approaches to guarantee safety of smart contracts are presented by Suvorov and Ulyantsev (2019); Mavridou and Laszka (2018). These approaches synthesize from EFSM models smart contracts guaranteed to fulfill given requirements. VerX (Permenev et al., 2020) is a verification tool for smart contracts that verifies properties expressed in past LTL (with a top-level 'always'). The above works focus mainly on safety properties, and nonblocking cannot be expressed.

Variants of the casino smart contract have been verified in Ahrendt et al. (2019) and Utting and Kent (2021) using off-the-shelf verification tools. Also here, the approaches

[*] This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

are limited to safety properties, and cannot show non-blocking. Ahrendt et al. (2019) can deal with reverting transactions, but not prove the absence of harmful transaction reverts.

This paper applies formal verification to the casino contract of Ahrendt et al. (2019). The contract is first modeled as a set of interacting EFSMs. To verify the correctness of this EFSM system, the supervisor synthesis and verification tool Supremica (Åkesson et al., 2006) is used, and the contract is verified to be blocking, which reveals an unintended security vulnerability. Analysing the counterexample generated by Supremica, adjustments are made to the casino contract to remove the blocking problem. Then, the adjusted contract is modeled again and verified to be non-blocking, showing the absence of the vulnerability.

The paper is organized as follows. Section 2 presents a brief background on EFSM and smart contracts. Section 3 gives a structured way to model smart contracts as a set of EFSMs. Section 4 describes the casino contract and its model as interacting EFSMs. Next, Section 5 models the security vulnerability. Section 6 shows that the EFSM *Casino* model is blocking. Section 7 shows the adjusted system, and Section 8 gives concluding remarks.

2. PRELIMINARIES

2.1 Smart Contracts: Ethereum and Solidity

The first, and still major, blockchain framework supporting smart contracts was Ethereum (Wood, 2014), with its built-in cryptocurrency Ether. In Ethereum not only the users, but also the contracts can receive, own, and send Ether. Sending Ether to a contract, and calling the contract, is the same thing in this framework. Sending funds to a contract without passing control to the receiver is not possible. Ethereum miners look for transaction requests on the network. A transaction request contains the address of a contract to be called, the call data, and the amount of Ether to be sent. Miners are paid for their efforts with units of (Ether prised) *gas*, to be paid by the address that requested the transaction.

A transaction may not necessarily be executed successfully. It can be reverted for various reasons: running out of gas, sending of unbacked funds, failing runtime assertions, or a simple revert statement in the code. If the miner attempts to execute a top-level (i.e., externally triggered) transaction, a reverting action anywhere inside the transaction execution will *undo the entire transaction*, all the way up the call stack. All the effects so far are also undone (except for the paid gas), as if the original call never happened. For instance, consider a case where a user calls some smart contract C, which during execution of the request sends Ether to another contract D. Recall that sending funds from C to D means that control is passed, for the moment, from C to D, and C can only resume once D returns. If D aborts before returning, the entire original request from the user gets undone.

The most popular programming language for Ethereum smart contracts is Solidity[1], which follows largely an object-oriented paradigm. Each external user and each contract instance has a unique `address`. Each `address` owns Ether (possibly 0), can receive Ether, and send Ether to other addresses. For instance, a`.transfer(`v`)` transfers the amount of v Wei ($= 10^{-18}$ Ether) from the caller to a. Built-in data types include unsigned integer (`uint`), enums, structs, and mappings. Mappings associate keys with values. For instance, the declaration `mapping (address => ... uint) public m` declares a field `m` which contains a mapping from addresses to unsigned integers. The current caller, and the amount of Wei sent with the call, are always available via `msg.sender` and `msg.value`, respectively. Only `payable` functions accept payments. `require(`b`)` checks the boolean expression b, and aborts if b is false. Fields marked `public` are read-public, not write-public. Solidity offers also some cryptographic primitives, for instance the function `keccak256` computing a crypto-hash of its argument.

Solidity further features programmable *modifiers*. For instance, the Casino contract in Fig. 1 uses the modifiers `byOperator`, `inState(s)`, and `noActiveBet`. These modifiers expand to `require` statements that abort the transaction if not fulfilled. The above modifiers expand to, respectively:

- `require (msg.sender == operator);`
- `require (state == s);`
- `require (state != State.BET_PLACED);`

2.2 Finite-state machines

Finite-state machines (FSM) (Ramadge and Wonham, 1989; Cassandras and Lafortune, 1999) are useful to model the logic of state-based smart contracts.

Definition 1. A *finite-state machine (FSM)* is a tuple $G = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$, where Σ is a set of events, the *alphabet*; Q is a finite set of *states*; $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *transition relation*; $Q^\circ \subseteq Q$ is the set of *initial states*; and $Q^\omega \subseteq Q$ is the set of *marked states*.

If Q° is a singleton and $\rightarrow$ a function, the FSM is said to be *deterministic*, else it is *non-deterministic*.

Let Σ^* be the set of all *finite traces* of events from Σ, including the *empty trace* ε. The transition relation is written in infix notation $q \xrightarrow{\sigma} p$, and is extended to traces in Σ^* by $p \xrightarrow{\varepsilon} p$ for all $p \in Q$, and $p \xrightarrow{s\sigma} q$ if $p \xrightarrow{s} r$ and $r \xrightarrow{\sigma} q$ for some $r \in Q$. The transition relation is also defined for state sets $R \subseteq Q$, for example $R \xrightarrow{s} q$ means $r \xrightarrow{s} q$ for some $r \in R$ and some $s \in \Sigma^*$.

Definition 2. An FSM $G = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$ is *non-blocking* if, for every trace $t \in \Sigma^*$ and every state $q \in Q$ such that $Q^\circ \xrightarrow{t} q$, exists a trace $s \in \Sigma^*$ such that $q \xrightarrow{s} Q^\omega$.

2.3 Extended Finite-State Machines

Extended finite-state machines (EFSM) (Cheng and Krishnakumar, 1993; Sköldstam et al., 2007) are similar to FSM, but augmented with bounded, discrete *variables*, and *updates* of those variables associated to the transitions. The updates are formulas constructed from variables, integer constants, the Boolean literals *true* (**T**) and *false* (**F**), and the usual arithmetic and logic connectives.

A *variable* v is an entity associated with a bounded discrete domain $\mathrm{dom}(v)$ and an initial value $v^\circ \in \mathrm{dom}(v)$. Let $V = \{v_0, \dots, v_n\}$ be the set of variables with domain

[1] `https://docs.soliditylang.org/en/latest/`

$\mathrm{dom}(V) = \mathrm{dom}(v_0) \times \cdots \times \mathrm{dom}(v_n)$. An element of $\mathrm{dom}(V)$ is called a *valuation* and is denoted by $\hat{v} = \langle \hat{v}_0, \ldots, \hat{v}_n \rangle$ with $\hat{v}_i \in \mathrm{dom}(v_i)$, and the value associated to variable $v_i \in V$ is denoted $\hat{v}[v_i] = \hat{v}_i$. The *initial valuation* is $v^\circ = \langle v_0^\circ, \ldots, v_n^\circ \rangle$.

A second set of variables, called *next-state variables*, denoted by $V' = \{\, v' \mid v \in V \,\}$ with $\mathrm{dom}(V') = \mathrm{dom}(V)$, is used to describe the values of the variables after a transition occurs. Variables in V are referred to as *current-state variables* to differentiate them from the next-state variables in V'. The set of all update formulas using variables in V and V' is denoted by Π_V.

For an update $p \in \Pi_V$, the terms $\mathrm{vars}(p)$ and $\mathrm{vars}'(p)$ denote the set of all variables, and the set of next-state variables, respectively, that occur in p. Updates $p \in \Pi_V$ can be interpreted as predicates over their variables, evaluating to $\mathbf{T}$ or $\mathbf{F}$, i.e., $p\colon \mathrm{dom}(V) \times \mathrm{dom}(V') \to \{\mathbf{T}, \mathbf{F}\}$.

Definition 3. An *extended finite-state machine (EFSM)* is a tuple $E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$, where Σ is a set of events; Q is a finite set of *locations*; $\rightarrow \subseteq Q \times \Sigma \times \Pi_V \times Q$ is the *conditional transition relation*; $Q^\circ \subseteq Q$ is the set of *initial locations*; and $Q^\omega \subseteq Q$ is the set of *marked locations.*

A transition in E is given as $q \xrightarrow{\sigma:p} q'$, which means that if update p evaluates to $\mathbf{T}$, the system can move from location q to location q' on the occurrence of the event σ. When the transition occurs the variables in $\mathrm{vars}'(p)$ are updated while the variables not contained in $\mathrm{vars}'(p)$ are unchanged.

Usually, EFSM models consist of several interacting components. Such a model is called an *EFSM system.*

Definition 4. An *EFSM system* is a collection of interacting EFSMs, $\mathcal{E} = \{E_1, \ldots, E_n\}$.

Component interaction in an EFSM systems is modeled by *synchronous composition* Hoare (1985).

Definition 5. Given two EFSMs $E_1 = \langle \Sigma_1, Q_1, \rightarrow_1, Q_1^\circ, Q_1^\omega \rangle$ and $E_2 = \langle \Sigma_2, Q_2, \rightarrow_2, Q_2^\circ, Q_2^\omega \rangle$, the *synchronous composition* of E_1 and E_2 is $E_1 \parallel E_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega \rangle$, where:

$$\begin{aligned}
&(x_1, x_2) \xrightarrow{\sigma:p_1 \wedge p_2} (y_1, y_2) \quad \text{if } \sigma \in \Sigma_1 \cap \Sigma_2,\ x_1 \xrightarrow{\sigma:p_1}_1 y_1, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ ; \\
&(x_1, x_2) \xrightarrow{\sigma:p_1} (y_1, x_2) \quad \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } x_1 \xrightarrow{\sigma:p_1}_1 y_1\ ; \\
&(x_1, x_2) \xrightarrow{\sigma:p_2} (x_1, y_2) \quad \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ .
\end{aligned}$$

Using Def. 5, the global behavior of a system $\mathcal{E} = \{E_1, \ldots, E_n\}$ is given by $E_1 \parallel \cdots \parallel E_n$.

Non-blocking of an EFSM system, is defined on the *flattened* system (Mohajerani et al., 2015).

Definition 6. Let $E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$ be an EFSM with variable set $\mathrm{vars}(E) = V$. The *monolithic flattening* of E is $U(E) = \langle \Sigma, Q_U, \rightarrow_U, Q_U^\circ, Q_U^\omega \rangle$ where

- $Q_U = Q \times \mathrm{dom}(V)$;
- $(x, \hat{v}) \xrightarrow{\sigma}_U (y, \hat{w})$ if E contains a transition $x \xrightarrow{\sigma:p} y$ such that $p(\hat{v}, \hat{w}) = \mathbf{T}$;
- $Q_U^\circ = Q^\circ \times \{v^\circ\}$;
- $Q_U^\omega = Q^\omega \times \mathrm{dom}(V)$.

$U(E)$ is the FSM representation of the EFSM, where all the variables have been removed and their values $\hat{v}$ embedded into the state set Q_U. This ensures the correct sequencing of transitions in the FSM. The monolithic flattened EFSM system $\mathcal{E}$ is $U(\mathcal{E}) = U(E_1 \parallel \ldots \parallel E_n)$.

Definition 7. An EFSM system $\mathcal{E}$ is non-blocking if $U(\mathcal{E})$ is non-blocking.

3. MODELING SOLIDITY CODE AS EFSMS

This section presents an approach for modeling a Solidity smart contract as a set of EFSMs, in general one EFSM for each function. Since the EFSMs considered in this paper are finite, the model only contains the logic part of the code and bounded variables. The following describes step by step how the EFSM is built. Though this proof-of-concept was performed manually, the process is well-structured enough to be automated.

Generally, each line of code of a function is represented by an EFSM location; all the locations are marked and the initial location q° represents the function being idle.

The alphabet of the EFSM model of a function comprises:

- One event labeled by the name of the function; this event represents the function being called;
- One event, *functionDone*, representing function termination;
- One event for each line of the function; these events are not shared with other EFSMs.

The set of variables of the EFSM model represents the bounded variables in the functions. Moreover, as mentioned in Section 2.1, the only reason for a function to revert considered in this work is that a function being called inside of it aborts. Thus, if a function is called from another function, a variable is assigned to capture its reversion or successful termination.

- For each bounded modifier inside the function, add to the EFSM a variable with the same domain.
- For each function called inside the function, add to the EFSM a variable with domain $\{0, 1\}$; 0 for reversion and 1 for successful termination of the called function.
- For each bounded variable inside the function, add to the EFSM a variable with the same domain.

Each transition of the EFSM represents the execution of some lines of the code, which contains assignment of the bounded variables, conditional statements, called functions or return of the function.

- The transition $q^\circ \xrightarrow{\mathit{function}:p} q_1$ models the function call. The update p represents the condition that the modifier imposes on the function.
- The transition $q_n \xrightarrow{\mathit{functionDone}:p} q^\circ$ represents termination of the function, typically its last line. Moreover, if a function is called inside of the function, $v = 1$ or $v = 0$ will be a part of the update p, where variable v represents the called function reversion or successful termination.
- For each line of code there is a transition $q_i \xrightarrow{e:p} q_{i+1}$. The event e is a general description of the line and the

update p is the conditional or assignment statement on any bounded variable.

The overall behavior of a smart contract typically goes through intermediate *states*, starting from an initial state to eventually reach a final state (typically the initial state). In different states, different functions of the contract are accessible, and cannot be called from other states. To manage the states of the contract, enums are often used.

In the EFSM model of the overall behavior of the contract, the locations directly correspond to the states of the contract, the enum values. Selfloops in the EFSM model, $enum_i \xrightarrow{function} enum_i$, represent function calls possible at the particular state. If the function termination causes the contract to move from one state to another, this is modeled as $enum_i \xrightarrow{functionDone} enum_{i+1}$, see for instance CGD (Create Game Done) in Fig. 2.

4. SYSTEM DESCRIPTION AND MODELING

This section describes the contract and, following the modeling technique proposed in Section 3, presents a model of it as a set of interacting EFSMs.

4.1 Casino Contract Overview

The Solidity code for the casino, Fig. 1, has three explicit states: `IDLE`, `GAME_AVAILABLE`, `BET_PLACED`, see line 3.

Based on the modifier `inState(s)`, in the `IDLE` location the operator may create a game by invoking the `createGame` function (line 9). To ensure a fair betting, the casino must place its bet at the time of game creation. Thus, when calling `createGame`, `hashedNumber` is assigned a value to later decide the game outcome (line 11). After creating a new game, the state changes to `GAME_AVAILABLE` (line 12) where a game is now available. In this state, the player can call `placeBet` to place a bet on TAILS or HEADS up to the size of the pot (lines 16-21). This then changes the state of the contract to `BET_PLACED` (line 23).

Next, the operator may by `decideBet` submit the original secret number to resolve the bet (line 25). If the secret number is even the coin toss is HEADS, else it is TAILS (line 29). If the player wins, double the bet is transferred to the player (line 41). If the operator wins, the bet is added to the pot and then set to zero (lines 44-45).

The operator may add money to the pot at any state, `addToPot` (line 47). Also, the operator may remove money from the pot, `removeFromPot` (line 51), but only if the player has not placed a bet, that is, if the casino is not in the state `BET_PLACED`. This is ensured by the modifier `noActiveBet`.

4.2 EFSM Model of the Casino

The *Casino* model is modular, see Fig. 2, comprising one EFSM for each Solidity function, and maps the code line by line, disregarding the unbounded variables. In addition, the *Casino* EFSM keeps track of each state of the contract. Thus, the locations of *Casino* are the values of `enum State`. Location I models `IDLE`, GA models `GAME_AVAILABLE`, and BP `BET_PLACED`. Since going back to the `IDLE` state shows completion of a game, location I is the initial and the marked location in *Casino*. The models of the functions have all locations marked.

```
1  contract Casino {
2
3    enum State {IDLE, GAME_AVAILABLE, ...
         BET_PLACED}
4    State private state;
5    address public operator, player;
6    bytes32 public hashedNumber;
7    struct Wager {uint bet; Coin guess;}
8
9  function createGame(bytes32 hashNum) public
10   byOperator, inState(State.IDLE) {
11   hashedNumber = hashNum
12   state = State.GAME_AVAILABLE;}
13
14 function placeBet(Coin _guess) public payable
15   inState(State.GAME_AVAILABLE) {
16   require (msg.sender != operator);
17   require (msg.value > 0 && msg.value ≤ pot);
18   player = msg.sender;
19   wager = Wager({
20     bet: msg.value,
21     guess: _guess
22   });
23   state = State.BET_PLACED;}
24
25 function decideBet(uint secretNumber) public
26   byOperator, inState(State.BET_PLACED) {
27   require (hashedNumber ==
28         keccak256(secretNumber));
29   Coin secret = (secretNumber % 2 == 0)? ...
               Coin.HEADS : Coin.TAILS;
30   if (secret == wager.guess) {
31      playerWins();
32   } else {
33      operatorWins();
34   }
35   state = State.IDLE;}
36
37 function playerWins() private {
38   tmp = wager.bet
39   wager.bet = 0;
40   pot = pot - tmp;
41   player.transfer(tmp*2);}
42
43 function operatorWins() private {
44   pot = pot + wager.bet;
45   wager.bet = 0;}
46
47 function addToPot() public payable
48   byOperator {
49   pot = pot + msg.value;}
50
51 function removeFromPot(uint amount) public
52   byOperator, noActiveBet {
53   pot = pot - amount;
54   operator.transfer(amount);}
55 }
```

Fig. 1. Solidity-code for Casino (some details are omitted)

The variables of the model are $V = \{gc, sc, hn, s, tp, to, pwd, owd\}$. Variables gc and sc, with the domain $\{t, h\}$ (t for TAILS and h for HEADS), capture the `guess` and `secret` coin variables (lines 21 and 29). The hn variable represents the `hashedNumber` (line 11) and, since only the evenness and oddness of the hashed number is relevant (line 29), the domain of hn is $\{e, od\}$ (e for even and od for odd). The s variable, with the domain $\{o, p\}$ (o for operator and p for player), models the `sender`. Moreover, as `transfer`, called in `removeFromPot` and `playerWins`, can succeed or fail, variables

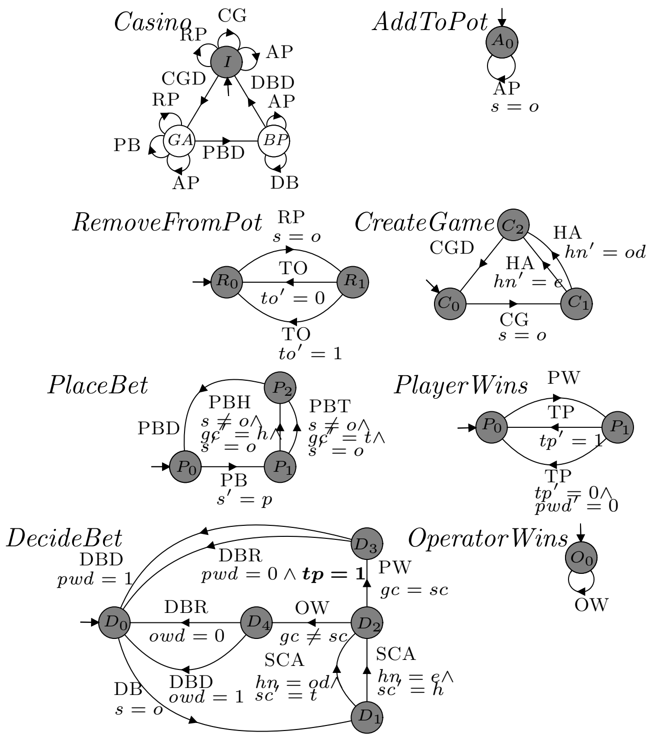

Fig. 2. EFSM model of the casino code of Fig. 1. Initial locations have small arrows pointing into them, and marked locations are shaded.

tp and *to*, both with domain $\{0, 1\}$ (0 for failure and 1 for success), are added to model the status of `transfer`. Similarly, as `playerWins` and `operatorWins` are called inside of `decideBet`, variables *pwd* and *owd*, with domain $\{0, 1\}$ (0 for reversion and 1 for success), model the status of `playerWins` and `operatorWins`, respectively. The initial value of variables *tp*, *to*, *pwd* and *owd* are 1, as initially it is assumed that the corresponding functions are not reverted. Moreover, since the operator is the one initially connected to the contract, $s^\circ = o$. The other variables have their entire domain as initial values, thus the model is non-deterministic.

When the casino is in `IDLE`, location I of the *Casino* EFSM, the operator can call functions `addToPot`, `removeFromPot`, or `createGame`, represented by the events AP, RP, and CG, respectively. Based on the occurred event, the corresponding EFSM transits from its initial location on a transition with the update $s = o$, which captures the `byOperator` modifier. Since the `pot` variable is not bounded, the `addToPot` EFSM has only a selfloop labeled by the event AP and the update $s = o$. Similar to *AddToPot*, line 53 in Fig. 1 is not modeled in the *RemoveFromPot* EFSM. Thus, after occurrence of RP, *RemoveFromPot* moves to location R_1, from where two transitions labeled by TO model line 54; the transition with the update $to' = 0$ represents a failed transfer, while $to' = 1$ represents a successful transfer to the operator. Since `addToPot` and `removeFromPot` do not change the casino state, their invocation is modeled as selfloops in *Casino*.

The *CreateGame* EFSM models the function `createGame` and occurrence of the CG event, the *CreateGame* EFSM moves to location C_1, which models calling `createGame`. In location C_1 a hashed number is assigned by event HA. Based on the evenness or oddness of the hashed number, the transition with update $hn' = e$ or $hn' = od$ occurs (compare line 11). At this point, by occurrence of the event CGD, *CreateGame* goes back to its initial location and *Casino* moves to location *GA*, which models line 12. Similar to location I, at *GA* events AP and RP can occur. At this location the player can place a bet by invoking `placeBet` (line 14), represented by the event PB. When PB occurs, *PlaceBet* moves to location P_1 and the `sender` becomes the player, by the update $s' = p$, to ensure that only the player can place a bet (line 16). At location P_1, events PBH or PBT can occur, taking *PlaceBet* to location P_2. These events, with their updates $gc' = h$ and $gc' = t$, represent that the player bets on HEADS or TAILS, modeling line 21. These transitions also update $s' = o$, as the player should not be able to invoke the other functions. At locations P_2 and *GA*, event PBD can occur, which takes *PlaceBet* to its initial location and *Casino* to the BP location, which models line 23.

In `BET_PLACED`, contrary to other states, `removeFromPot` cannot be invoked due to the `noActiveBet` modifier (line 52). Thus, the RP event is omitted at BP. The operator can invoke `addToPot` at BP, though, represented by the AP selfloop. Moreover, at BP the DB event can occur. This represents the operator resolves the winner by invoking `decideBet` (lines 25-35). Occurrence of DB takes the *DecideBet* EFSM to location D_1 with the update $s = o$. At D_1 two SCA labeled transitions with different updates can occur, which models line 29. The updates $hn = e \wedge sc' = h$ and $hn = od \wedge sc' = t$ model that `hashedNumber` is even or odd and that `secret` is assigned HEADS or TAILS, accordingly. On SCA, *DecideBet* moves to location D_2, where the events PW and OW can occur, respectively representing player or operator winning. In `decideBet`, if `secret` and `wager.guess` are equal, `playerWins` is called, else `operatorWins` is called (lines 30-33). These conditional statements are modeled as updates on the transitions $D_2 \xrightarrow{\text{PW}:gc=sc} D_3$ and $D_2 \xrightarrow{\text{OW}:gc\neq sc} D_4$, respectively, in *DecideBet*. Based on the winner, *DecideBet* will move to D_3 or D_4, each having two possible transitions to D_0. The transition labeled by DBD represents that `decideBet` is successfully terminated. In this case, line 35 is executed, so that the casino goes to its initial location, which is modeled as BP $\xrightarrow{\text{DBD}}$ I in *Casino*. Furthermore, the transitions to D_0 labeled by DBR with updates $pwd = 0$ and $owd = 0$ respectively model that `decideBet` reverts due to `playerWins` or `operatorWins` reverting. In this case `decideBet` reverts and can be called again, but line 35 is not executed. To model this, *Casino* has no transition on DBR.

The function `decideBet` calls `operatorWins`, which is modeled by the event OW in the EFSM *DecideBet*. Since, in `operatorWins` the variables `pot` and `bet` are unbounded, lines 44 and 45 are not modeled in the EFSM *OperatorWins*. Thus, *OperatorWins* is a selfloop labeled by the OW event. As no function is called inside `operatorWins`, the reverting of it is not modeled, so the variable *owd* is not affected.

Similarly, the *PlayerWins* EFSM models `playerWins`. If the player wins, the money is transferred to the player by the `transfer` (line 41). Since `transfer` is a function called inside `playerWins`, at location P_1 in *PlayerWins* two transitions labeled by the event TP can occur. The transition with the update $tp' = 0$ models transaction failure, while the transition with the update $tp' = 1$ models success. Moreover, a failed `transfer` will cause `playerWins` to revert and the update $tp' = 0 \wedge pwd' = 0$ models this situation.

5. MODEL OF THE SECURITY VULNERABILITY

With a malicious player that refuses to accept payment after a win, the casino will be prevented to progress and all funds will be locked. This section shows how this security attack can be modeled and verified to exist.

In the casino contract, refusal of `transfer` causes `playerWins` to revert and consequently `decideBet` to revert (line 31). Thus, the event DBR should occur in *DecideBet*. To verify if the attack can result in prevention of progress, an update $tp = 1$, in bold in Fig. 2, is added, which models this security attack. This update is conjuncted to the transition $D_3 \xrightarrow{\text{DBR}:pwd=0} D_0$. This ensures that `decideBet` can revert due to `playerWins` reverting, update $pwd = 0$, and be called again only if `playerWins` reverting is not the result of the malicious player refusing the payment, $tp = 1$.

6. NON-BLOCKING VERIFICATION

As the contract executes, its state is updated, to after a full round return to the initial state. This is represented by *Casino* from its initial location I firing transitions corresponding to execution of the modeled lines of code returning back to I, with all EFSMs in their respective initial locations. Thus, verifying whether the contract is vulnerable to a malicious player preventing progress and locking the funds, can be mapped to non-blocking verification in the SCT framework.

To verify the non-blockingness of the EFSM system $\mathcal{E}$ shown in Fig. 2, the software tool Supremica (Åkesson et al., 2006) that implements formal verification and synthesis of systems modeled as FSMs or EFSMs is used. When Supremica's non-blocking verification algorithm finds a blocking state, it generates a counterexample, a trace of events from the initial state to the blocking state.

Non-blocking verification of $\mathcal{E}$ shows the system to be blocking, with the following counterexample:

$$\begin{aligned}&\text{CG.HA}\{hn' = od\}\text{.CGD.PB.PBT.PBD.DB.}\\ &\text{SCA}\{hn = od\}\text{.PW}\{gc = t\}\text{.TP}\{pwd' = 0\}. \qquad (1)\end{aligned}$$

In (1), four events, HA$\{hn' = od\}$, SCA$\{hn = od\}$, PW$\{gc = t\}$, and TP$\{pwd' = 0\}$ are added to the system by the flattening. These are extended by the corresponding updates and their values when the transition occurs. For example, event HA$\{hn' = od\}$ is the event HA extended with the update $hn' = od$, representing the event where the `hashedNumber` is assigned an odd number.

The last two events of (1) show that if the player bets TAILS and wins, event PW$\{gc = t\}$, and then refuses the payment causing `playerWins` to revert, event TP$\{pwd' = 0\}$, the system blocks. This captures that the player's malicious behavior prevents progress of the contract. Regardless of the HEADS or TAILS bet, the blocking occurs if the player wins and the transfer to the player fails.

Inspection of the Solidity code, Fig. 1, maps the blocking to line 41. If `player.transfer(wager.bet*2)` fails, line 35 will not be executed. Though `decideBet` can be called again, if the malicious player refuses the `transfer` on each call, the contract will not progress to reach its `IDLE` state.

7. NON-BLOCKING CASINO MODEL

Section 6 shows that the casino contract blocks and cannot go back to its `IDLE` state if the player always rejects the transferred payment. This section proposes adjustments of the Solidity code to solve this blocking problem.

7.1 Solidity Code for Non-Blocking Casino

The blocking problem arises due to the casino contract not being able to go back to its `IDLE` state. One solution is to modify the `playerWins` function to omit the `transfer` and instead use an `account` mapping to associate the player's address to the player's account, Fig. 3. Thus, the `playerWins` function cannot fail and line 35 will always be executed. Every time that the player wins, `wager.bet*2` is removed from the pot and added to the player's account (lines 63-64). To enable the player to withdraw funds, the `withdraw` function (line 67) is added that can be called at any time. By `withdraw`, the `account` mapping looks up the account related to the address that initiated the call and stores its value in the variable *tmp* (line 68). Next, the account of the `withdraw` caller is set to zero (line 69) and the amount stored in *tmp* is transferred to the caller (line 70). If this transfer is rejected by the receiver, this reverts only the `withdraw`, while the casino contract is unaffected.

7.2 EFSM model of Non-blocking Casino

The EFSM model of the new Solidity code is shown in Fig. 4. The only change between *Casino'* of Fig. 4 and *Casino* are the selfloops labeled with event WP, which allow `withdraw` to be called at any location. When WP occurs, the EFSM *Withdraw* moves to location W_1, where two transitions, labeled with the event TP can occur. The transitions have the updates $tp' = 0$ and $tp' = 1$, respectively, modeling a successful and failed transaction, respectively. Since `withdraw` is not called inside any function, in *Withdraw* the update that models that `withdraw` failed due to a rejected `transfer` is omitted. Since the new `playerWins` function only stores the unbounded amount that the player wins, the EFSM *PlayerWins'* is a selfloop. Moreover, since the `transfer` is omitted in the function `playerWins`, the termination or reversion of the function `decideBet` is independent of failure or success of the transfer to the player. Thus, the update $tp = 1$ is not added to transition $D_3 \xrightarrow{\text{DBR}:pwd=0} D_0$ in the EFSM *DecideBet'*. The rest of the functions and their the EFSM models are not changed. The EFSM model of the adjusted casino system is:

$$\begin{aligned}\mathcal{E}' = \{&\mathit{Casino}', \mathit{AddToPot}, \mathit{RemoveFromPot},\\ &\mathit{CreateGame}, \mathit{PlaceBet}, \mathit{DecideBet}',\\ &\mathit{OperatorWins}, \mathit{PlayerWins}', \mathit{Withdraw}\}.\end{aligned} \qquad (2)$$

This system is verified to be non-blocking, thus *Casino'* is not vulnerable to the player payment refusal attack.

```
  mapping (address => uint) account;

function playerWins() private {
  pot = pot - wager.bet;
  account[player] = account[player] + ...
      wager.bet*2;
  wager.bet = 0;}

function withdraw() public {
  uint tmp = account[msg.sender];
  account[msg.sender] = 0;
  msg.sender.transfer(tmp);}
```

Fig. 3. Part of the non-blocking casino Solidity code.

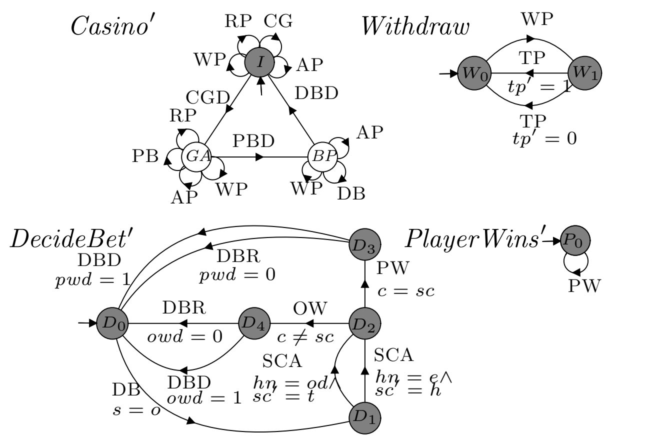

Fig. 4. EFSM model of the non-blocking casino of Fig. 3.

8. CONCLUSION

This paper investigates formal non-blocking verification of a casino smart contract. Smart contracts can often be modelled as state machines, where state transitions model function calls. In many cases, smart contracts even explicitly implement state machines. Leveraging this, the Solidity code of the casino smart contract is modeled as a set of interacting EFSMs. Applying non-blocking verification on the EFSM system shows that failure in transferring money to the player may result in the contract not able to go back to its `IDLE` state. Then, the bets are forever locked into the contract. This type of attack has been exercised on real contracts with huge financial damage. An improved casino contract removes the dependency of moving back to the `IDLE` state from the `transfer`. Verification of the adjusted model shows that this model is in fact non-blocking.

In the future, the authors will investigate automatic modeling of Solidity code as interacting EFSMs.

ACKNOWLEDGEMENTS

The authors thank Gordon Pace for providing the casino smart contract example.

REFERENCES

Åkesson, K., Fabian, M., Flordal, H., and Malik, R. (2006). Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06*, 384–385. IEEE. doi:10.1109/WODES.2006.382401.

Ahrendt, W., Bubel, R., Ellul, J., Pace, G.J., Pardo, R., Rebiscoul, V., and Schneider, G. (2019). Verification of smart contract business logic exploiting a Java source code verifier. In H. Hojjat and M. Massink (eds.), *Fundamentals of Software Engineering, FSEN, Tehran, Iran, May 2019, Proceedings*, volume 11761 of *LNCS*. Springer.

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, 164–186. Springer-Verlag, Berlin, Heidelberg.

Cassandras, C.G. and Lafortune, S. (1999). *Introduction to Discrete Event Systems*. Kluwer.

Cheng, K.T. and Krishnakumar, A.S. (1993). Automatic functional test generation using the extended finite state machine model. In *Proc. 30th ACM/IEEE Design Automation Conf.*, 86–91. doi:10.1145/157485.164585.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Leduc, R.J., Lawford, M., and Dai, P. (2006). Hierarchical interface-based supervisory control of a flexible manufacturing system. *IEEE Trans. Control Syst. Technol.*, 14(4), 654–668.

Mavridou, A. and Laszka, A. (2018). Designing secure Ethereum smart contracts: A finite state machine based approach. In S. Meiklejohn and K. Sako (eds.), *Financial Cryptography and Data Security*, 523–540. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mohajerani, S., Malik, R., and Fabian, M. (2015). A framework for compositional nonblocking verification of extended finite-state machines. *Discrete Event Dyn. Syst.* doi:10.1007/s10626-015-0217-y.

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. (2020). VerX: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, 1661–1677. doi:10.1109/SP40000.2020.00024.

Ramadge, P.J.G. and Wonham, W.M. (1989). The control of discrete event systems. *Proc. IEEE*, 77(1), 81–98.

Sköldstam, M., Åkesson, K., and Fabian, M. (2007). Modeling of discrete event systems using finite automata with variables. In *Proc. 46th IEEE Conf. Decision and Control, CDC '07*, 3387–3392.

Suvorov, D. and Ulyantsev, V. (2019). Smart contract design meets state machine synthesis: Case studies. URL `https://arxiv.org/abs/1906.02906`.

Teixeira, M., Malik, R., Cury, J.E.R., and de Queiroz, M.H. (2015). Supervisory control of DES with extended finite-state machines and variable abstraction. *IEEE Trans. Autom. Control*, 60(1), 118–129. doi:10.1109/TAC.2014.2337411.

Utting, M. and Kent, L. (2021). Verification of a smart contract for a simple casino. *Computing Research Repository (CoRR) in arXiv*. URL `https://arxiv.org/abs/2106.14457`.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 1–32.

Zita, A., Mohajerani, S., and Fabian, M. (2017). Application of formal verification to the lane change module of an autonomous vehicle. In *2017 13th IEEE Conference on Automation Science and Engineering*, 932–937.