A formal model of Algorand smart contracts

Massimo Bartoletti[1], Andrea Bracciali[2], Cristian Lepore[2],
Alceste Scalas[3], Roberto Zunino[4]

[1] Università degli Studi di Cagliari, Cagliari, Italy
[2] Stirling University, Stirling, UK
[3] Technical University of Denmark, Lyngby, Denmark
[4] Università degli Studi di Trento, Trento, Italy

Abstract. We develop a formal model of Algorand stateless smart contracts (stateless ASC1). We exploit our model to prove fundamental properties of the Algorand blockchain, and to establish the security of some archetypal smart contracts. While doing this, we highlight various design patterns supported by Algorand. We perform experiments to validate the coherence of our formal model w.r.t. the actual implementation.

1 Introduction

Smart contracts are agreements between two or more parties that are automatically enforced without trusted intermediaries. Blockchain technologies reinvented the idea of smart contracts, providing trustless environments where they are incarnated as computer programs. However, writing secure smart contracts is difficult, as witnessed by the multitude of attacks on smart contracts platforms (notably, Ethereum) — and since smart contracts control assets, their bugs may directly lead to financial losses.

Algorand [20] is a late-generation blockchain that features a set of interesting features, including high-scalability and a no-forking consensus protocol based on Proof-of-Stake [7]. Its smart contract layer (ASC1) aims to mitigate smart contract risks, and adopts a non-Turing-complete programming model, natively supporting atomic sets of transactions and user-defined assets. These features make it an intriguing smart contract platform to study.

The official specification and documentation of ASC1 consists of English prose and a set of templates to assist programmers in designing their contracts [1,3]. This conforms to standard industry practices, but there are two drawbacks:

1. Algorand lacks a mathematical model of contracts and transactions suitable for formal reasoning on their behaviour, and for the verification of their properties. Such a model is needed to develop techniques and tools to ensure that contracts are correct and secure;
2. furthermore, even preliminary informal reasoning on non-trivial smart contracts can be challenging, as it may require, in some corner cases, to resort to experiments, or direct inspection of the platform source code.

Given these drawbacks, we aim at developing a formal model that:

o1. is high-level enough to simplify the design of Algorand smart contracts and enable formal reasoning about their security properties;
o2. expresses Algorand contracts in a simple declarative language, similar to PyTeal (the official Python binding for Algorand smart contracts) [5];
o3. provides a basis for the automatic verification of Algorand smart contracts.

Contributions. This paper presents:

- a *formal model of stateless ASC1* providing a solid theoretical foundation to Algorand smart contracts (§2). Such a model formalises both Algorand accounts and transactions (§2.1 – §2.4, §2.6), and smart contracts (§2.5);
- a validation of our model through experiments [4] on the Algorand platform;
- the formalisation and proof of some *fundamental properties of the Algorand state machine*: no double spending, determinism, value preservation (§2.7);
- an *analysis of Algorand contract design patterns* (§3.2), based on several non-trivial contracts (covering both standard use cases, and novel ones). Quite surprisingly, we show that stateless contracts are expressive enough to encode arbitrary finite state machines;
- the proof of relevant *security properties of smart contracts* in our model;
- a *prototype tool* that compiles smart contracts (written in our formal declarative language) into executable TEAL code (§4).

Our formal model is faithful to the actual ASC1 implementation; by objectives o1–o3, it strives at being high-level and simple to understand, while covering the most commonly used primitives and mechanisms of Algorand, and supporting the specification and verification of non-trivial smart contracts (§3.2, §4). To achieve these objectives, we introduce minor high-level abstractions over low-level details: e.g., since TEAL code has the purpose of accepting or rejecting transactions, we model it using expressions that evaluate to *true* or *false* (similarly to PyTEAL); we also formalise different transaction types by focusing on their function, rather than their implementation. Our objectives imply that we do *not* aim at covering all the possible TEAL contracts with bytecode-level accuracy, and our Algorand model is *not* designed as a full low-level formalisation of the behavior of the Algorand blockchain. We discuss the differences between our model and the actual Algorand platform in §5. Due to space constraints, we provide the proofs of our statements in a separate technical report [14].

2 The Algorand state machine

We present our formal model of the Algorand blockchain, including its smart contracts (stateless ASC1), incrementally. We first define the basic transactions that generate and transfer assets (§2.1–§2.3), and then add atomic groups of transactions (§2.4), smart contracts (§2.5), and authorizations (§2.6). We discuss the main differences between our model and Algorand in §5.

$\mathsf{a}, \mathsf{b}, \ldots$	Users (key pairs)	$T_{\mathsf{lv}} \subseteq \mathbb{T}$	Transactions in last Δ_{max} rounds
$\mathbf{x}, \mathbf{y}, \ldots \in \mathbb{X}$	Addresses	$f_{asst} \in \mathbb{A} \to \mathbb{X}$	Asset manager
$\tau, \tau', \ldots \in \mathbb{A}$	Assets	$f_{lx} \in (\mathbb{X} \times \mathbb{N}) \rightharpoonup \mathbb{N}$	Lease map
$v, w, \ldots \in 0..2^{64}-1$	Values	$f_{frz} \in \mathbb{X} \rightharpoonup 2^{\mathbb{A}}$	Freeze map
$\sigma, \sigma' \in \mathbb{A} \rightharpoonup \mathbb{N}$	Balances	$\Gamma, \Gamma', \ldots$	Blockchain states
$\mathbf{x}[\sigma]$	Accounts	$\models \sigma$	Valid balance
$t, t', \ldots \in \mathbb{T}$	Transactions	$f_{lx}, r \models t$	Valid time constraint
$e, e', \ldots$	Scripts	$\mathcal{W} \models \mathcal{T}, i$	Authorized transaction in group
$r, r' \ldots \in \mathbb{N}$	Rounds	$[\![e]\!]^{\mathcal{W}}_{\mathcal{T},i}$	Script evaluation

Table 1: Summary of notation.

2.1 Accounts and transactions

We use $\mathsf{a}, \mathsf{b}, \ldots$ to denote public/private key pairs $(k^p_{\mathsf{a}}, k^s_{\mathsf{a}})$. Users interact with Algorand through pseudonymous identities, obtained as a function of their public keys. Hereafter, we freely use a to refer to the public or the private key of a, or to the user associated with them, relying on the context to resolve the ambiguity. The purpose of Algorand is to allow users to exchange ***assets*** $\tau, \tau', \ldots$ Besides the Algorand native cryptocurrency Algo, users can create custom assets.

We adopt the following notational convention:

- lowercase letters for single entities (e.g., a user a);
- uppercase letters for *sets* of entities (e.g., a set of users A);
- calligraphic uppercase letters for *sequences* of entities (e.g., list of users $\mathcal{A}$).

Given a sequence $\mathcal{L}$, we write $|\mathcal{L}|$ for its length, $set(\mathcal{L})$ for the set of its elements, and $\mathcal{L}.i$ for its i^{th} element ($i \in 1..|\mathcal{L}|$); ε denotes the empty sequence. We write:

- $\{x \mapsto v\}$ for the function mapping x to v, and having domain equal to $\{x\}$;
- $f\{x \mapsto v\}$ for the function mapping x to v, and y to $f(y)$ if $y \neq x$;
- $f\{x \mapsto \bot\}$ for the function undefined at x, and mapping y to $f(y)$ if $y \neq x$.

Accounts. An ***account*** is a deposit of one or more crypto-assets. We model accounts as terms $\mathbf{x}[\sigma]$, where $\mathbf{x}$ is an ***address*** uniquely identifying the account, and σ is a ***balance***, i.e., a finite map from assets to non-negative 64-bit integers. In the concrete Algorand, an address is a 58-characters word; for mathematical elegance, in our model we represent an address as either:

- a ***single user*** a. Performing transactions on $\mathsf{a}[\sigma]$ requires a's authorization;
- a pair $(\mathcal{A}, n)$, where $\mathcal{A}$ is a sequence of users, and $1 \leq n \leq |\mathcal{A}|$, are ***multisig (multi-signature)*** addresses. Performing transactions on $(\mathcal{A}, n)[\sigma]$ requires that at least n users out of those in $\mathcal{A}$ grant their authorization;[5]
- a ***script***[6] e. Performing transactions on $e[\sigma]$ requires e to evaluate to *true*.

[5] W.l.o.g., we consider a single-user address a equivalent to $(\mathcal{A}, n)$ with $\mathcal{A} = \langle \mathsf{a} \rangle$, $n = 1$.
[6] We formalize scripts (i.e., smart contracts) later on, in §2.5.

pay	snd, rcv, val, asst	snd transfers val units of asst to rcv (possibly creating rcv)
close	snd, rcv, asst	snd gives asst to rcv and removes it (if Algo, closes snd)
gen	snd, rcv, val	snd mints val units of a new asset, managed by rcv
optin	snd, asst	snd opts in to receive units of asset asst
burn	asst	asst is removed from the creator (if sole owner)
rvk	snd, rcv, val, asst	asst's manager transfers val units of asst from snd to rcv
frz	snd, asst	asst's manager freezes snd's use of asset asst
unfrz	snd, asst	asst's manager unfreezes snd's use of asset asst
delegate	snd, asst, rcv	asst's manager delegates asst to new manager rcv

Fig. 1: Transaction types. Fields type, fv, lv, lx are common to all types.

Each balance is required to own Algos, have at least 100000 micro-Algos for each owned asset, and cannot control more than 1000 assets. Formally, we say that σ is a ***valid balance*** (in symbols, $\models \sigma$) when:[7]

$$\mathsf{Algo} \in \mathrm{dom}\,(\sigma) \ \wedge \ \sigma(\mathsf{Algo}) \geq 100000 \cdot |\,\mathrm{dom}\,(\sigma)| \ \wedge \ |\,\mathrm{dom}\,(\sigma)| \leq 1001$$

Transactions. Accounts can append various kinds of ***transactions*** to the blockchain, in order to, e.g., alter their balance or set their usage policies. We model transactions as records with the structure in Fig. 1. Each transaction has a type, which determines which of the other fields are relevant.[8] The field snd usually refers to the subject of the transaction (e.g., the *sender* in an assets transfer), while rcv refers to the *receiver* in an assets transfer. The fields asst and val refer, respectively, to the affected asset, and to its amount. The fields fv ("first valid"), lv ("last valid") and lx ("lease") are used to impose time constraints.

Algorand groups transactions into ***rounds*** $r = 1, 2, \ldots$ To establish *when* a transaction t is valid, we must consider both the current round r, and a ***lease map*** f_{lx} binding pairs (address, lease identifier) to rounds: this is used to enforce mutual exclusion between two or more transactions (see e.g. the *periodic payment* contract in §3). Formally, we define the ***temporal validity of a transaction*** t by the predicate $f_{lx}, r \models t$, which holds whenever:

$$t.\mathsf{fv} \leq r \leq t.\mathsf{lv} \quad \text{and} \quad t.\mathsf{lv} - t.\mathsf{fv} \leq \Delta_{max} \quad \text{and}$$
$$\big(t.\mathsf{lx} = 0 \quad \text{or} \quad (t.\mathsf{snd}, t.\mathsf{lx}) \notin \mathrm{dom}\,(f_{lx}) \quad \text{or} \quad r > f_{lx}(t.\mathsf{snd}, t.\mathsf{lx})\big)$$

First, the current round must lie between $t.\mathsf{fv}$ and $t.\mathsf{lv}$, whose distance cannot exceed Δ_{max} rounds[9]. Second, t must have a null lease identifier, or the identifier has not been seen before (i.e., $f_{lx}(t.\mathsf{snd}, t.\mathsf{lx})$ is undefined), or the lease has expired (i.e., $r > f_{lx}(t.\mathsf{snd}, t.\mathsf{lx})$). When performed, a transaction with non-null lease identifier acquires the lease on $(t.\mathsf{snd}, t.\mathsf{lx})$, which is set to $t.\mathsf{lv}$.

[7] Since the codomain of σ is $\mathbb{N}$, the balance entry $\sigma(\mathsf{Algo})$ represents micro-Algos.

[8] In Algorand, the actual behaviour of a transaction may depend on both its type and other conditions, e.g., which optional fields are set. For instance, *pay* transactions may also close accounts if the CloseRemainderTo field is set. For the sake of clarity, in our model we prefer to use a richer set of types; see §5 for other differences.

[9] Δ_{max} is a global consensus parameter, set to 1000 at time of writing.

2.2 Blockchain states

We model the evolution of the Algorand blockchain as a labelled transition system. A ***blockchain state*** Γ has the form:

$$\mathbf{x}_1[\sigma_1] \mid \cdots \mid \mathbf{x}_n[\sigma_n] \mid r \mid T_{\mathsf{lv}} \mid f_{asst} \mid f_{lx} \mid f_{frz} \tag{1}$$

where all addresses $\mathbf{x}_i$ are distinct, $\mid$ is commutative and associative, and:

- r is the current round;
- T_{lv} is the set of performed transactions whose "last valid" time lv has not expired. This set is used to avoid double spending (see Theorem 1);
- f_{asst} maps each asset to the addresses of its ***manager*** and ***creator***;
- f_{lx} is the ***lease map*** (from pairs (address, integer) to integers), used to ensure mutual exclusion between transactions;
- f_{frz} is a map from addresses to sets of assets, used to ***freeze assets***.

We define the ***initial state*** Γ_0 as $\mathsf{a}_0[\{\mathsf{Algo} \mapsto v_0\}] \mid 0 \mid \emptyset \mid f_{asst} \mid f_{lx} \mid f_{frz}$, where $\mathrm{dom}\,(f_{asst}) = \mathrm{dom}\,(f_{lx}) = \mathrm{dom}\,(f_{frz}) = \emptyset$, a_0 is the initial user address, and $v_0 = 10^{16}$ (which is the total supply of 10 billions Algos).

We now formalize the ASC1 state machine, by defining how it evolves by single transactions (§2.3), and then including atomic groups of transactions (§2.4), smart contracts (§2.5), and the authorization of transactions (§2.6).

2.3 Executing single transactions

We write $\Gamma \xrightarrow{t}_1 \Gamma'$ to mean: *if* the transaction t is performed in blockchain state Γ, then the blockchain evolves to state Γ'.[10] We specify the transition relation $\to_1$ through a set of inference rules (see [14, Fig. 5 in Appendix] for the full definition): each rule describes the effect of a transaction t in the state Γ of eq. (1). We now illustrate all cases, depending on the transaction type ($t.\mathsf{type}$).

When $\tau \in \mathrm{dom}\,(\sigma)$, we use the shorthand $\sigma + v{:}\tau$ to update balance σ by adding v units to token τ; similarly, we write $\sigma - v{:}\tau$ to decrease τ by v units:

$$\sigma + v{:}\tau \equiv \sigma\{\tau \mapsto \sigma(\tau) + v\} \qquad \sigma - v{:}\tau \equiv \sigma\{\tau \mapsto \sigma(\tau) - v\}$$

Pay to a new account. Let $t.\mathsf{snd} = \mathbf{x}_i$ for some $i \in 1..n$, let $t.\mathsf{rcv} = \mathbf{y} \notin \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ (i.e., the sender account $\mathbf{x}$ is already in the state, while the receiver $\mathbf{y}$ is not), and let $t.\mathsf{val} = v$. The rule has the following preconditions:

c1. t does not cause double-spending ($t \notin T_{\mathsf{lv}}$);
c2. the time interval of the transaction, and its lease, are respected ($f_{lx}, r \models t$);
c3. the updated balance of $\mathbf{x}_i$ is valid ($\models \sigma_i - v : \mathsf{Algo}$);
c4. the balance of the new account at address $\mathbf{y}$ is valid ($\models \{\mathsf{Algo} \mapsto v\}$).

[10] Note that $\Gamma \xrightarrow{t}_1 \Gamma'$ does not imply that transaction t *can* be performed in Γ: in fact, t might require an authorization. We specify the required conditions in §2.6.

If these conditions are satisfied, the new state Γ' is the following:

$$\mathbf{x}_i[\sigma_i - v{:}\mathsf{Algo}] \mid \mathbf{y}[\{\mathsf{Algo} \mapsto v\}] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

In the new state, the Algo balance of $\mathbf{x}_i$ is decreased by v units, and a new account at $\mathbf{y}$ is created, containing exactly the v units taken from $\mathbf{x}_i$. The balances of the other accounts are unchanged. The updated lease mapping is:

$$\mathit{upd}(f_{lx}, t) = \begin{cases} f_{lx}\{(t.\mathsf{snd}, t.\mathsf{lx}) \mapsto t.\mathsf{lv}\} & \text{if } t.\mathsf{lx} \neq 0 \\ f_{lx} & \text{otherwise} \end{cases}$$

Note that all transaction types check conditions c1 and c2 above; further, all transactions check that updated account balances are valid (as in c3 and c4).

Pay to an existing account. Let $t.\mathsf{snd} = \mathbf{x}_i$, $t.\mathsf{rcv} = \mathbf{x}_j$, $t.\mathsf{val} = v$, and $t.\mathsf{asst} = \tau$. Besides the common checks, performing t requires that $\mathbf{x}_j$ has "opted in" τ (formally, $\tau \in \mathrm{dom}\,(\sigma_j)$), and τ must not be frozen in accounts $\mathbf{x}_i$ and $\mathbf{x}_j$ (formally, $\tau \notin f_{frz}(\mathbf{x}_i) \cup f_{frz}(\mathbf{x}_j)$). If $\mathbf{x}_i \neq \mathbf{x}_j$, then in the new state the balance of τ in $\mathbf{x}_i$ is decreased by v units, and that of τ in $\mathbf{x}_j$ is increased by v units:

$$\mathbf{x}_i[\sigma_i - v{:}\tau] \mid \mathbf{x}_j[\sigma_j + v{:}\tau] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

where all accounts but $\mathbf{x}_i$ and $\mathbf{x}_j$ are unchanged. Otherwise, if $\mathbf{x}_i = \mathbf{x}_j$, then the balance of $\mathbf{x}_i$ is unchanged, and the other parts of the state are as above.

Close. Let $t.\mathsf{snd} = \mathbf{x}_i$, $t.\mathsf{rcv} = \mathbf{x}_j \neq \mathbf{x}_i$, and $t.\mathsf{asst} = \tau$. Performing t has two possible outcomes, depending on whether τ is Algo or a user-defined asset. If $\tau = \mathsf{Algo}$, we must check that σ_i contains *only* Algos. If so, the new state is:

$$\mathbf{x}_j[\sigma_j + \sigma_i(\mathsf{Algo}){:}\mathsf{Algo}] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

where the new state no longer contains the account $\mathbf{x}_i$, and all the Algos in $\mathbf{x}_i$ are transferred to $\mathbf{x}_j$. Instead, if $\tau \neq \mathsf{Algo}$, performing t requires to check only that $\mathbf{x}_i$ actually contains τ, and that $\mathbf{x}_j$ has "opted in" τ. Further, τ must not be frozen for addresses $\mathbf{x}_i$ and $\mathbf{x}_j$, i.e. $\tau \notin f_{frz}(\mathbf{x}_i) \cup f_{frz}(\mathbf{x}_j)$. The new state is:

$$\mathbf{x}_i[\sigma_i\{\tau \mapsto \bot\}] \mid \mathbf{x}_j[\sigma_j + \sigma_i(\tau){:}\tau] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

where τ is removed from $\mathbf{x}_i$, and all the units of τ in $\mathbf{x}_i$ are transferred to $\mathbf{x}_j$.

Gen. Let $t.\mathsf{snd} = \mathbf{x}_i$, $t.\mathsf{rcv} = \mathbf{x}_j$, and $t.\mathsf{val} = v$. Performing t requires that $\mathbf{x}_i$ has enough Algos to own another asset, i.e. $\models \sigma_i\{\tau \mapsto v\}$, where τ is the (fresh) identifier of the new asset. In the new state, the balance of $\mathbf{x}_i$ is extended with $\{\tau \mapsto v\}$, and f_{asst} is updated, making $\mathbf{x}_j$ the manager of τ. The new state is:

$$\mathbf{x}_i[\sigma_i\{\tau \mapsto v\}] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst}\{\tau \mapsto (\mathbf{x}_j, \mathbf{x}_i)\} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

Opt in. Let $t.\mathsf{snd} = \mathbf{x}_i$ and $t.\mathsf{asst} = \tau$. Performing t requires that τ already occurs in Γ, and that $\mathbf{x}_i$ has enough Algos to store it. If the balance σ_i does not have an entry for τ, in the new state σ_i is extended with a new entry for τ:

$$\mathbf{x}_i[\sigma_i\{\tau \mapsto 0\}] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

Otherwise, if $\mathbf{x}_i$'s balance has already an entry for τ, then σ_i is unchanged.

Burn. Let $t.\mathsf{snd} = \mathbf{x}_i$ and $t.\mathsf{asst} = \tau$. Performing t requires that $\mathbf{x}_i$ is the creator of τ, and that $\mathbf{x}_i$ stores *all* the units of τ (i.e., there are no units of τ in other accounts). In the resulting state, the token τ no longer exists:

$$\mathbf{x}_i[\sigma_i\{\tau \mapsto \bot\}] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst}\{\tau \mapsto \bot\} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

Note that this transaction requires an authorization by the asset manager of τ, which is recorded in f_{asst}. (We address this topic in §2.6.)

Revoke. Let $t.\mathsf{snd} = \mathbf{x}_i$ and $t.\mathsf{rcv} = \mathbf{x}_j$. Performing t requires that both $\mathbf{x}_i$ and $\mathbf{x}_j$ are already storing the asset τ, and that τ is not frozen for $\mathbf{x}_i$ and $\mathbf{x}_j$. In the new state, the balance of $\mathbf{x}_i$ is decreased by $v = t.\mathsf{val}$ units of the asset $\tau = t.\mathsf{asst}$, and the balance of $\mathbf{x}_j$ is increased by the same amount:

$$\mathbf{x}_i[\sigma_i - v{:}\tau] \mid \mathbf{x}_j[\sigma_j + v{:}\tau] \mid \cdots \mid r \mid T_{\mathsf{lv}} \cup \{t\} \mid f_{asst} \mid \mathit{upd}(f_{lx}, t) \mid f_{frz}$$

The effect of a *rvk* transaction is essentially the same as *pay*. The difference is that *rvk* must be authorized by the manager of the asset τ, while *pay* must be authorized by the sender $\mathbf{x}_i$ (see §2.6).

Freeze and unfreeze. A *frz* transaction t with $t.\mathsf{snd} = \mathbf{x}_i$ and $t.\mathsf{asst} = \tau$ updates the mapping f_{frz} into f'_{frz}, such that $f'_{frz}(\mathbf{x}_i) = f_{frz}(\mathbf{x}_i) \cup \{\tau\}$, whenever the asset τ is owned by $\mathbf{x}_i$. This effectively prevents any transfers of the asset τ to/from the account $\mathbf{x}_i$. The dual transaction *unfrz* updates the mapping f_{frz} into f'_{frz} such that $f'_{frz}(\mathbf{x}_i) = f_{frz}(\mathbf{x}_i) \setminus \{\tau\}$.

Delegate. A *delegate* transaction t with $t.\mathsf{snd} = \mathbf{x}_i$, $t.\mathsf{rcv} = \mathbf{x}_j$ and $t.\mathsf{asst} = \tau$ updates the manager of τ, provided that $f_{asst}(\tau) = (\mathbf{x}_i, \mathbf{x}_i)$, for some $\mathbf{x}_k$. In the updated mapping $f_{asst}\{\tau \mapsto (\mathbf{x}_j, \mathbf{x}_k)\}$, the manager of τ is $\mathbf{x}_j$.

Initiating a new round. We model the advancement to the next round of the blockchain as a state transition $\Gamma \xrightarrow{\checkmark}_1 \Gamma'$. In the new state Γ', the round is increased, and the set T_{lv} is updated as $T'_{\mathsf{lv}} = \{t \in T_{\mathsf{lv}} \mid t.\mathsf{lv} > r\}$. The other components of the state are unchanged.

2.4 Executing atomic groups of transactions

Atomic transfers allow state transitions to atomically perform *sequences* of transactions. To atomically perform a sequence $\mathcal{T} = t_1 \cdots t_n$ from a state Γ, we must check that all the transactions t_i can be performed *in sequence*, i.e. the following precondition must hold (for some $\Gamma_1, \ldots, \Gamma_n$):

$$\Gamma \xrightarrow{t_1}_1 \Gamma_1 \quad \cdots \quad \Gamma_{n-1} \xrightarrow{t_n}_1 \Gamma_n$$

If so, the state Γ can take a *single*-step transition labelled $\mathcal{T}$. Denoting the new transition relation with $\rightarrow$, we write the atomic execution of $\mathcal{T}$ in Γ as follows:

$$\Gamma \xrightarrow{\mathcal{T}} \Gamma_n$$

e	$::= v$	constant
	$\mid e \circ e$	arithmetic ($\circ \in \{+, -, <, \leq, =, \geq, >, *, /, \%, \mathsf{and}, \mathsf{or}\}$)
	$\mid \mathsf{not}\ e$	negation
	$\mid \mathsf{txlen}$	number of transactions in the atomic group
	$\mid \mathsf{txpos}$	index of current transaction in the atomic group
	$\mid \mathsf{txid}(n)$	identifier of n-th transaction in the atomic group
	$\mid \mathsf{tx}(n).\mathsf{f}$	value of field f of n-th transaction in the atomic group
	$\mid \mathsf{arg}(n)$	n-th argument of the current transaction
	$\mid \mathsf{H}(e)$	hash
	$\mid \mathsf{versig}(e, e, e)$	signature verification

Syntactic sugar: $\mathit{false} ::= 1 = 0$ $\mathit{true} ::= 1 = 1$ $\mathsf{tx.f} ::= \mathsf{tx}(\mathsf{txpos}).\mathsf{f}$ $\mathsf{txid} ::= \mathsf{txid}(\mathsf{txpos})$
$\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 ::= (e_0\ \mathsf{and}\ e_1)\ \mathsf{or}\ ((\mathsf{not}\ e_0)\ \mathsf{and}\ e_2)$

Fig. 2: Smart contract scripts (inspired by PyTeal [5]).

2.5 Executing smart contracts

In Algorand, custom authorization policies can be defined with a smart contract language called TEAL [6]. TEAL is a bytecode-based stack language, with an official programming interface for Python (called PyTeal): in our formal model, we take inspiration from the latter to abstract TEAL bytecode scripts as terms, with the syntax in Fig. 2. Besides standard arithmetic-logical operators, TEAL includes operators to count and index all transactions in the current atomic group, and to access their id and fields. When firing transaction involving scripts, users can specify a sequence of ***arguments***; accordingly, the script language includes operators to know the number of arguments, and access them. Further, scripts include cryptographic operators to compute hashes and verify signatures.

The ***script evaluation function*** $[\![e]\!]^{\mathcal{W}}_{\mathcal{T},i}$ (Fig. 3) evaluates e using 3 parameters: a sequence of arguments $\mathcal{W}$, a sequence of transactions $\mathcal{T}$ forming an atomic group, and the index $i < |\mathcal{T}|$ of the transaction containing e. The script $\mathsf{tx}(n).\mathsf{f}$ evaluates to the field f of the n^{th} transaction in group $\mathcal{T}$. The size of $\mathcal{T}$ is given by txlen, while txpos returns the index i of the transaction containing the script being evaluated. The script $\mathsf{arg}(n)$ returns the n^{th} argument in $\mathcal{W}$. The script $\mathsf{H}(e)$ applies a public hash function H to the evaluation of e. The script $\mathsf{versig}(e_1, e_2, e_3)$ verifies a signature e_2 on the message obtained by concatenating the enclosing script and e_1, using public key e_3. All operators in Fig. 3 are *strict*: they fail if the evaluation of any operand fails.

2.6 Authorizing transactions, and user-blockchain interaction

As noted before, the mere existence of a step $\Gamma \xrightarrow{t}_1 \Gamma'$ does not imply that t can actually be issued. For this to be possible, users must provide a sequence $\mathcal{W}$ of ***witnesses***, satisfying the ***authorization predicate*** associated with t; such

$$[\![v]\!]^{\mathcal{W}}_{\mathcal{T},i} = v \qquad [\![e \circ e']\!]^{\mathcal{W}}_{\mathcal{T},i} = [\![e]\!]^{\mathcal{W}}_{\mathcal{T},i} \circ_\bot [\![e']\!]^{\mathcal{W}}_{\mathcal{T},i} \qquad [\![\mathsf{not}\ e]\!]^{\mathcal{W}}_{\mathcal{T},i} = \neg_\bot [\![e]\!]^{\mathcal{W}}_{\mathcal{T},i}$$
$$[\![\mathsf{tx}(n).\mathsf{f}]\!]^{\mathcal{W}}_{\mathcal{T},i} = (\mathcal{T}.n).\mathsf{f} \quad (0 \le n < |\mathcal{T}|) \qquad [\![\mathsf{txid}(n)]\!]^{\mathcal{W}}_{\mathcal{T},i} = \mathcal{T}.n \quad (0 \le n < |\mathcal{T}|)$$
$$[\![\mathsf{txlen}]\!]^{\mathcal{W}}_{\mathcal{T},i} = |\mathcal{T}| \qquad [\![\mathsf{txpos}]\!]^{\mathcal{W}}_{\mathcal{T},i} = i \qquad [\![\mathsf{arg}(n)]\!]^{\mathcal{W}}_{\mathcal{T},i} = \mathcal{W}.n \quad (0 \le n < |\mathcal{W}|)$$
$$[\![\mathsf{H}(e)]\!]^{\mathcal{W}}_{\mathcal{T},i} = H([\![e]\!]^{\mathcal{W}}_{\mathcal{T},i}) \qquad [\![\mathsf{versig}(e_1,e_2,e_3)]\!]^{\mathcal{W}}_{\mathcal{T},i} = ver_k(m,s) \begin{pmatrix} m = (\mathcal{T}.i.\mathsf{snd}, [\![e_1]\!]^{\mathcal{W}}_{\mathcal{T},i}) \\ s = [\![e_2]\!]^{\mathcal{W}}_{\mathcal{T},i} \quad k = [\![e_3]\!]^{\mathcal{W}}_{\mathcal{T},i} \end{pmatrix}$$

Fig. 3: Evaluation of scripts in Fig. 2.

a predicate is uniquely determined by the ***authorizer address*** of t, written $auth(t, f_{asst})$. For transaction types *close*, *pay*, *gen*, *optin* the authorizer address is $t.\mathsf{snd}$; for *burn*, *rvk*, *frz* and *unfrz* on an asset τ it is the asset manager $f_{asst}(\tau)$. Intuitively, if $auth(t, f_{asst}) = \mathbf{x}$, then $\mathcal{W}$ authorizes t iff:

1. if $\mathbf{x}$ is a multisig address $(\mathcal{A}, n)$, then $\mathcal{W}$ contains at least n signatures of t, made by users in $\mathcal{A}$; (if $\mathbf{x}$ is a single-user address a: see footnote 5)
2. if $\mathbf{x}$ is a script e, then e evaluates to *true* under the arguments $\mathcal{W}$.

We now formalize the intuition above. Since the evaluation of scripts depends on a whole group of transactions $\mathcal{T}$, and on the index i of the current transaction within $\mathcal{T}$, we define the authorization predicate as $\mathcal{W} \models \mathcal{T}, i$ (read: "$\mathcal{W}$ authorizes the i^{th} transaction in $\mathcal{T}$"). Let $sig_{\mathsf{A}}(m)$ stand for the set of signatures containing $sig_{\mathsf{a}}(m)$ for all $\mathsf{a} \in \mathsf{A}$; then, $\mathcal{W} \models \mathcal{T}, i$ holds whenever:

1. if $auth(\mathcal{T}.i, f_{asst}) = (\mathcal{A}, n)$, then $|set(\mathcal{W}) \cap sig_{set(\mathcal{A})}(\mathcal{T}, i)| \ge n$
2. if $auth(\mathcal{T}.i, f_{asst}) = e$, then $[\![e]\!]^{\mathcal{W}}_{\mathcal{T},i} = true$

Note that, in general, the sequence of witnesses $\mathcal{W}$ is not unique, i.e., it may happen that $\mathcal{W} \models \mathcal{T}, i$ and $\mathcal{W}' \models \mathcal{T}, i$ for $\mathcal{W} \neq \mathcal{W}'$. For instance, the *Oracle* contract in §3 accepts transactions with witnesses of the form $0\,s$ or $1\,s'$, where the first element of the sequence represents the oracle's choice, and the second element is the oracle's signature.
Given a *sequence of sequences* of witnesses $\mathbf{W} = \mathcal{W}_0 \cdots \mathcal{W}_{n-1}$ with $n = |\mathcal{T}|$, the ***group authorization predicate*** $\mathbf{W} \models \mathcal{T}$ holds iff $\mathcal{W}_i \models \mathcal{T}, i$ for all $i \in 0..n-1$.

User-blockchain interaction. We model the interaction of users with the blockchain as a transition system. Its states are pairs (Γ, K), where Γ is a blockchain state, while K is the set of authorization bitstrings currently known by users. The transition relation $\overset{\ell}{\Rightarrow}$ (with $\ell \in \{w, \checkmark, \mathbf{W}{:}\mathcal{T}\}$) is given by the rules:

$$\frac{}{(\Gamma, K) \overset{w}{\Rightarrow} (\Gamma, K \cup \{w\})} \qquad \frac{\Gamma \xrightarrow{\checkmark} \Gamma'}{(\Gamma, K) \overset{\checkmark}{\Rightarrow} (\Gamma', K)} \qquad \frac{\Gamma \xrightarrow{\mathcal{T}} \Gamma' \quad set(\mathbf{W}) \subseteq K \quad \mathbf{W} \models \mathcal{T}}{(\Gamma, K) \xRightarrow{\mathbf{W}:\mathcal{T}} (\Gamma', K)}$$

With the first two rules, users can broadcast a witness w, or advance to the next round. The last rule gathers from K a sequence of witnesses $\mathbf{W}$, and lets the blockchain perform an atomic group of transactions $\mathcal{T}$ if authorized by $\mathbf{W}$.

2.7 Fundamental properties of ASC1

We now exploit our formal model to establish some fundamental properties of ASC1. Theorem 1 states that the same transaction t cannot be issued more than once, i.e., there is no *double-spending*. In the statement, we use $\to^* \xrightarrow{\mathcal{T}} \to^*$ to denote an arbitrarily long series of steps including a group of transactions $\mathcal{T}$.

Theorem 1 (No double-spending). *Let* $\Gamma_0 \to^* \xrightarrow{\mathcal{T}} \to^* \Gamma \xrightarrow{\mathcal{T}'} \Gamma'$. *Then, no transaction occurs more than once in* $\mathcal{T}\mathcal{T}'$.

Define the *value* of an asset τ in a state $\Gamma = \mathbf{x}_1[\sigma_1] \mid \cdots \mid \mathbf{x}_n[\sigma_n] \mid r \mid \cdots$ as the sum of the balances of τ in all accounts in Γ:

$$val_\tau(\Gamma) = \textstyle\sum_{i=1}^{n} val_\tau(\sigma_i) \qquad \text{where} \quad val_\tau(\sigma) = \begin{cases} \sigma(\tau) & \text{if } \tau \in \mathrm{dom}\,(\sigma) \\ 0 & \text{otherwise} \end{cases}$$

Theorem 2 states that, once an asset is minted, its value remains constant, until the asset is eventually burnt. In particular, since Algos cannot be burnt (nor minted, unlike in Bitcoin and Ethereum), their amount remains constant.

Theorem 2 (Value preservation). *Let* $\Gamma_0 \to^* \Gamma \to^* \Gamma'$. *Then:*

$$val_\tau(\Gamma') = \begin{cases} val_\tau(\Gamma) & \text{if } \tau \text{ occurs in } \Gamma \text{ and it is not burnt in } \Gamma \to^* \Gamma' \\ 0 & \text{otherwise} \end{cases}$$

Theorem 3 establishes that the transition systems $\to$ and $\Rightarrow$ are deterministic: crucially, this allows reconstructing the blockchain state from the transition log. Notably, by item 3 of Theorem 3, witnesses only determine whether a state transition happens or not, but they do not affect the new state. This is unlike Ethereum, where arguments of function calls in transactions may affect the state.

Theorem 3 (Determinism). *For all* $\lambda \in \{\checkmark, \mathcal{T}\}$ *and* $\ell \in \{\checkmark, w\}$:

1. *if* $\Gamma \xrightarrow{\lambda} \Gamma'$ *and* $\Gamma \xrightarrow{\lambda} \Gamma''$, *then* $\Gamma' = \Gamma''$;
2. *if* $(\Gamma, K) \overset{\ell}{\Rightarrow} (\Gamma', K')$ *and* $(\Gamma, K) \overset{\ell}{\Rightarrow} (\Gamma'', K'')$, *then* $(\Gamma', K') = (\Gamma'', K'')$;
3. *if* $(\Gamma, K) \overset{\mathbf{W}:\mathcal{T}}{\Longrightarrow} (\Gamma', K')$ *and* $(\Gamma, K) \overset{\mathbf{W}':\mathcal{T}}{\Longrightarrow} (\Gamma'', K'')$, *then* $\Gamma' = \Gamma''$ *and* $K' = K'' = K$.

3 Designing secure smart contracts in Algorand

We now exploit our formal model to design some archetypal smart contracts, and establish their security (§3.2). First, we introduce an attacker model.

3.1 Attacker model

We assume that cryptographic primitives are secure, i.e., hashes are collision resistant and signatures cannot be forged (except with negligible probability). A ***run*** $\mathcal{R}$ is a (possibly infinite) sequence of labels $\ell_1\ell_2\cdots$ such that $(\Gamma_0, K_0) \overset{\ell_1}{\Rightarrow}$

$(\Gamma_1, K_1) \overset{\ell_2}{\Rightarrow} \cdots$, where Γ_0 is the initial state, and $K_0 = \emptyset$ is the initial (empty) knowledge; hence, as illustrated in §2.6, each label ℓ_i in a run $\mathcal{R}$ can be either w (broadcast of a witness bitstring w), $\mathbf{W}{:}\mathcal{T}$ (atomic group of transactions $\mathcal{T}$ authorized by $\mathbf{W}$), or ✓ (advance to next round). We consider a setting where:

- each user a has a ***strategy*** Σ, i.e. a PPTIME algorithm to select which label to perform among those permitted by the ASC1 transition system. A strategy takes as input a finite run $\mathcal{R}$ (the past history) and outputs a single enabled label ℓ. Strategies are *stateful*: users can read and write a private unbounded tape to maintain their own state throughout the run. The initial state of a's tape contains a's private key, and the public keys of all users;[11]
- an ***adversary*** Adv who controls the scheduling with her stateful ***adversarial strategy*** Σ_{Adv}: a PPTIME algorithm taking as input the current run $\mathcal{R}$ and the labels output by the strategies of users (i.e., the steps that users are trying to make). The output of Σ_{Adv} is a single label ℓ, that is appended to the current run. We assume the adversarial strategy Σ_{Adv} can delay users' transactions by at most δ_{Adv} rounds, where δ_{Adv} is a given natural number.[12]

A set $\boldsymbol{\Sigma}$ of strategies of users and Adv induces a distribution of runs; we say that run $\mathcal{R}$ is ***conformant*** to $\boldsymbol{\Sigma}$ if $\mathcal{R}$ is sampled from such a distribution. We assume that infinite runs contain infinitely many ✓: this *non-Zeno condition* ensures that neither users nor Adv can perform infinitely many transactions in a round.

3.2 Smart contracts

We now exploit our model to specify some archetypal ASC1 contracts, and reason about their security. To simplify the presentation, we assume $\delta_{\mathsf{Adv}} = 0$, i.e., the adversary Adv can start a new round (performing ✓) only if all users agree.[13] The table below summarises our selection of smart contracts, highlighting the design patterns they implement.

Use case / Pattern	Signed witness	Timeouts	Commit/ reveal	State machine	Atomic transfer	Time windows
Oracle	✓	✓				
HTLC		✓	✓			
Mutual HTLC [14, §B.1]		✓	✓		✓	
$O(n^2)$-collateral lottery		✓	✓		✓	
0-collateral lottery [14, §B.2]		✓	✓	✓	✓	
Periodic payment						✓
Escrow [14, §B.3]	✓			✓		
Two-phase authorization		✓		✓		✓
Limit order [14, §B.4]		✓			✓	
Split [14, §B.5]		✓			✓	

[11] Notice that new public/private key pairs can be generated during the run, and their public parts can be communicated as labels w.

[12] Without this assumption, Adv could arbitrarily disrupt deadlines: e.g., Σ_{Adv} could make a *always* lose lottery games (like the ones below) by delaying a's transactions.

[13] All results can be easily adjusted for $\delta_{\mathsf{Adv}} > 0$, but this would require more verbose statements to account for possible delays introduced by Adv.

Oracle. We start by designing a contract which allows either a or b to withdraw all the Algos in the contract, depending on the outcome of a certain boolean event. Let o be an oracle who certifies such an outcome, by signing the value 1 or 0. We model the contract as the following script:

$$\begin{aligned} \mathit{Oracle} \triangleq\ & \mathsf{tx.type} = \mathit{close} \text{ and } \mathsf{tx.asst} = \mathsf{Algo} \text{ and } \big((\mathsf{tx.fv} > r_{max} \text{ and } \mathsf{tx.rcv} = \mathsf{a}) \\ & \text{or } (\mathsf{arg}(0) = 0 \text{ and } \mathsf{versig}(\mathsf{arg}(0), \mathsf{arg}(1), \mathsf{o}) \text{ and } \mathsf{tx.rcv} = \mathsf{a}) \\ & \text{or } (\mathsf{arg}(0) = 1 \text{ and } \mathsf{versig}(\mathsf{arg}(0), \mathsf{arg}(1), \mathsf{o}) \text{ and } \mathsf{tx.rcv} = \mathsf{b})\big) \end{aligned}$$

Once created, the contract accepts only *close* transactions, using two arguments as witnesses. The argument $\mathsf{arg}(0)$ contains the outcome, while $\mathsf{arg}(1)$ is o's signature on $(\mathit{Oracle}, \mathsf{arg}(0))$, i.e., the concatenation between the script and the first argument. The user b can collect the funds in *Oracle* if o certifies the outcome 1, while a can collect the funds if the outcome is 0, or after round r_{max}.

Theorem 4 below proves that *Oracle* works as intended. To state it, we define T_{p} as the set of transactions allowing a user p to withdraw the contract funds:

$$T_{\mathsf{p}} = \{t \mid t.\mathsf{type} = \mathit{close},\ t.\mathsf{snd} = \mathit{Oracle},\ t.\mathsf{rcv} = \mathsf{p},\ t.\mathsf{asst} = \mathsf{Algo}\}$$

The theorem considers the following strategies for a, b, and o:

- Σ_{a}: wait for $s = sig_{\mathsf{o}}(\mathit{Oracle}, 0)$; if s arrives at round $r \le r_{max}$, then immediately send a transaction $t \in T_{\mathsf{a}}$ with $t.\mathsf{fv} = r$ and witness $0\,s$; otherwise, at round $r_{max}+1$, send a transaction $t \in T_{\mathsf{a}}$ with $t.\mathsf{fv} = r_{max}+1$;
- Σ_{b}: wait for $s' = sig_{\mathsf{o}}(\mathit{Oracle}, 1)$; if s' arrives at round r, immediately send a transaction $t \in T_{\mathsf{b}}$ with $t.\mathsf{fv} = r$ and witness $1\,s'$;
- Σ_{o}: do one of the following: *(a)* send o's signature on $(\mathit{Oracle}, 0)$ at any time, or *(b)* send o's signature on $(\mathit{Oracle}, 1)$ at any time, or *(c)* do nothing.

Theorem 4. *Let $\mathcal{R}$ be a run conforming to some set of strategies $\boldsymbol{\Sigma}$, such that: (i) $\Sigma_{\mathsf{o}} \in \boldsymbol{\Sigma}$; (ii) $\mathcal{R}$ reaches, at some round before r_{max}, a state $\mathit{Oracle}[\sigma] \mid \cdots$; (iii) $\mathcal{R}$ reaches the round $r_{max}+2$. Then, with overwhelming probability:*

(1) if $\Sigma_{\mathsf{a}} \in \boldsymbol{\Sigma}$ and o has not sent a signature on $(\mathit{Oracle}, 1)$, then $\mathcal{R}$ contains a transaction in T_{a}, transferring at least $\sigma(\mathsf{Algo})$ to a;
(2) if $\Sigma_{\mathsf{b}} \in \boldsymbol{\Sigma}$ and o has sent a signature on $(\mathit{Oracle}, 1)$ at round $r \le r_{max}$, then $\mathcal{R}$ contains a transaction in T_{b}, transferring at least $\sigma(\mathsf{Algo})$ to b.

Notice that in item (1) we are only assuming that a and o use the strategies Σ_{a} and Σ_{o}, while b and Adv can use *any* strategy (and possibly collude). Similarly, in item (2) we are only assuming b's and o's strategies.

Hash Time Lock Contract (HTLC). A user a promises that she will either reveal a secret s_{a} by round r_{max}, or pay a penalty to b. More sophisticated contracts, e.g. gambling games, use this mechanism to let players generate random numbers in a fair way. We define the HTLC as the following contract, parameterised on the two users a, b and the hash $h_{\mathsf{a}} = H(s_{\mathsf{a}})$ of the secret:

$$\begin{aligned} \mathit{HTLC}(\mathsf{a}, \mathsf{b}, h_{\mathsf{a}}) \triangleq\ & \mathsf{tx.type} = \mathit{close} \text{ and } \mathsf{tx.asst} = \mathsf{Algo} \text{ and} \\ & \big((\mathsf{tx.rcv} = \mathsf{a} \text{ and } \mathsf{H}(\mathsf{arg}(0)) = h_{\mathsf{a}}) \text{ or } (\mathsf{tx.rcv} = \mathsf{b} \text{ and } \mathsf{tx.fv} \ge r_{max})\big) \end{aligned}$$

The contract accepts only *close* transactions with receiver a or b. User a can collect the funds in the contract only by providing the secret s_{a} in $\mathsf{arg}(0)$, effectively making s_{a} public.[14] Instead, if a does not reveal s_{a}, then b can collect the funds after round r_{max}. We state the correctness of $HTLC$ in Theorem 5; first, let T_{p} be the set of transactions allowing user p to withdraw the contract funds:

$$T_{\mathsf{p}} \;=\; \{t \mid t.\mathsf{type} = \mathit{close},\ t.\mathsf{snd} = \mathit{HTLC}(\mathsf{a},\mathsf{b},h_{\mathsf{a}}),\ t.\mathsf{rcv} = \mathsf{p},\ t.\mathsf{asst} = \mathsf{Algo}\}$$

We consider the following strategies for a and b:

- Σ_{a}: at a round $r < r_{max}$, send a $t \in T_{\mathsf{a}}$ with $t.\mathsf{fv} = r$ and witness s_{a};
- Σ_{b}: at round r_{max}, check whether any transaction in T_{a} occurs in $\mathcal{R}$. If not, then immediately send a transaction $t \in T_{\mathsf{b}}$ with $t.\mathsf{fv} = r_{max}$.

Theorem 5. *Let $\mathcal{R}$ be a run conforming to some set of strategies $\boldsymbol{\Sigma}$, such that: (i) $\mathcal{R}$ reaches, at some round before r_{max}, a state $\mathit{HTLC}(\mathsf{a},\mathsf{b},h_{\mathsf{a}})[\sigma] \mid \cdots$; (ii) $\mathcal{R}$ reaches the round $r_{max}+1$. Then, with overwhelming probability:*

(1) if $\Sigma_{\mathsf{a}} \in \boldsymbol{\Sigma}$, then $\mathcal{R}$ contains a transaction in T_{a}, transferring at least $\sigma(\mathsf{Algo})$ to a;
(2) if $\Sigma_{\mathsf{b}} \in \boldsymbol{\Sigma}$ and $\mathcal{R}$ does not contain the secret s_{a} before round $r_{max}+1$, then $\mathcal{R}$ contains a transaction in T_{b}, transferring at least $\sigma(\mathsf{Algo})$ to b.

Lotteries. Consider a gambling game where n players bet 1Algo each, and the winner, chosen uniformly at random among them, can redeem n Algos. A simple implementation, inspired by [9–11] for Bitcoin, requires each player to deposit $n(n-1)$ Algos as collateral in an HTLC contract.[15] For $n = 2$ players a and b, such deposits are transferred by the following transactions:

$$t_{H\mathsf{a}} \;=\; \{\mathsf{type} : \mathit{pay},\ \mathsf{snd} : \mathsf{a},\ \mathsf{rcv} : \mathit{HTLC}(\mathsf{a},\ \mathsf{b},\ h_{\mathsf{a}}),\ \mathsf{val} : 2,\ \mathsf{asst} : \mathsf{Algo},\ \ldots\}$$
$$t_{H\mathsf{b}} \;=\; \{\mathsf{type} : \mathit{pay},\ \mathsf{snd} : \mathsf{b},\ \mathsf{rcv} : \mathit{HTLC}(\mathsf{b},\ \mathsf{a},\ h_{\mathsf{b}}),\ \mathsf{val} : 2,\ \mathsf{asst} : \mathsf{Algo},\ \ldots\}$$

The bets are stored in the following contract, which determines the winner as a function of the secrets, and allows her to withdraw the whole pot:

$$\mathit{Lottery} \triangleq \mathsf{tx.type} = \mathit{close}\ \mathsf{and}\ \mathsf{tx.asst} = \mathsf{Algo}\ \mathsf{and}\ \mathsf{H}(\mathsf{arg}(0)) = h_{\mathsf{a}}\ \mathsf{and}\ \mathsf{H}(\mathsf{arg}(1)) = h_{\mathsf{b}}$$
$$\mathsf{and\ if}\ (\mathsf{arg}(0) + \mathsf{arg}(1))\%2 = 0\ \mathsf{then\ tx.rcv} = \mathsf{a}\ \mathsf{else\ tx.rcv} = \mathsf{b}$$

with $h_{\mathsf{a}} \neq h_{\mathsf{b}}$.[16] Players a and b start the game with the atomic transactions:

$$\mathcal{T}_{\mathsf{a},\mathsf{b}} \;=\; t_{H\mathsf{a}}\, t_{H\mathsf{b}}\, t_{L\mathsf{a}}\, t_{L\mathsf{b}} \quad \text{where:}$$
$$t_{L\mathsf{a}} \;=\; \{\mathsf{type} : \mathit{pay},\ \mathsf{snd} : \mathsf{a},\ \mathsf{rcv} : \mathit{Lottery},\ \mathsf{val} : 1,\ \mathsf{asst} : \mathsf{Algo},\ \ldots\}$$
$$t_{L\mathsf{b}} \;=\; \{\mathsf{type} : \mathit{pay},\ \mathsf{snd} : \mathsf{a},\ \mathsf{rcv} : \mathit{Lottery},\ \mathsf{val} : 1,\ \mathsf{asst} : \mathsf{Algo},\ \ldots\}$$

The transaction $t_{L\mathsf{a}}$ creates the contract with a's bet, and $t_{L\mathsf{b}}$ completes it with b's bet. At this point, there are two possible outcomes:

(a) both players reveal their secret. Then, the winner can withdraw the pot, by performing a *close* action on the *Lottery* contract, providing as arguments the two secrets, and setting her identity in the rcv field;

[14] If s_{a} is a sufficiently long bitstring generated uniformly at random, collision resistance of the hash function ensures that only a (who knows s_{a}) can provide such an $\mathsf{arg}(0)$.
[15] A zero-collateral lottery is presented in [14, §B.2].
[16] This check prevents a replay attack: if a chooses $h_{\mathsf{a}} = h_{\mathsf{b}}$, then b cannot win.

(b) one of the players does not reveal the secret. Then, the other player can withdraw the collateral in the other player's HTLC (and redeem her own).

To formalise the correctness of the lottery, consider the sets of transactions:

$$T^{secr}_{\mathsf{p},\mathsf{q}} = \{t \mid t.\mathsf{type} = \mathit{close},\ t.\mathsf{snd} = \mathit{HTLC}(\mathsf{p},\mathsf{q},h_{\mathsf{p}}),\ t.\mathsf{rcv} = \mathsf{p},\ t.\mathsf{asst} = \mathsf{Algo}\}$$
$$T^{tout}_{\mathsf{p},\mathsf{q}} = \{t \mid t.\mathsf{type} = \mathit{close},\ t.\mathsf{snd} = \mathit{HTLC}(\mathsf{p},\mathsf{q},h_{\mathsf{p}}),\ t.\mathsf{rcv} = \mathsf{q},\ t.\mathsf{asst} = \mathsf{Algo}\}$$
$$T^{lott}_{\mathsf{p}} = \{t \mid t.\mathsf{type} = \mathit{close},\ t.\mathsf{snd} = \mathit{Lottery},\ t.\mathsf{rcv} = \mathsf{p},\ t.\mathsf{asst} = \mathsf{Algo}\}$$

and consider the following strategy Σ_{a} for a (the one for b is analogous):

1. at some $r < r_{max}$, send a transaction $t \in T^{secr}_{\mathsf{a},\mathsf{b}}$ with $t.\mathsf{fv} = r$ and witness s_{a};
2. if some transaction in $T^{secr}_{\mathsf{b},\mathsf{a}}$ occurs in $\mathcal{R}$ at round $r' < r_{max}$, then extract its witness s_{b} and compute the winner; if a is the winner, immediately send a transaction $t \in T^{lott}_{\mathsf{a}}$ with $t.\mathsf{fv} = r'$ and witness $s_{\mathsf{a}}\, s_{\mathsf{b}}$;
3. if at round r_{max} no transaction in $T_{\mathsf{b},\mathsf{a}}$ occurs in $\mathcal{R}$, immediately send a transaction $t \in T^{tout}_{\mathsf{b},\mathsf{a}}$ with $t.\mathsf{fv} = r_{max}$.

Theorem 6 below establishes that the lottery is fair, implying that the expected payoff of player a following strategy Σ_{a} is at least negligible; instead, if a does not follow Σ_{a} (e.g., by not revealing her secret), the expected payoff may be negative; analogous results hold for player b. This result can be generalised for $n > 2$ players, with a collateral of $n(n-1)$ Algos. As in the HTLC, we assume that s_{a} and s_{b} are sufficiently long bitstrings generated uniformly at random.

Theorem 6. *Let $\mathcal{R}$ be a run conforming to a set of strategies $\boldsymbol{\Sigma}$, such that: (i) $\mathcal{R}$ contains, before r_{max}, the label $\mathcal{T}_{\mathsf{a},\mathsf{b}}$; (ii) $\mathcal{R}$ reaches round $r_{max}+1$. For $\mathsf{p} \neq \mathsf{q} \in \{\mathsf{a},\mathsf{b}\}$, if $\Sigma_{\mathsf{p}} \in \boldsymbol{\Sigma}$, then: (1) $\mathcal{R}$ contains a transaction in $T^{secr}_{\mathsf{p},\mathsf{q}}$, transferring at least 2 Algo to p; (2) the probability that $\mathcal{R}$ contains $T^{tout}_{\mathsf{q},\mathsf{p}}$ or T^{lott}_{p}, which transfer at least 1 Algo to p, is $\geq \frac{1}{2}$ (up-to a negligible quantity).*

Periodic payment. We want to ensure that a can withdraw a fixed amount of v Algos at fixed time windows of p rounds. We can implement this behaviour through the following contract, which can be refilled when needed:

$$PP(p,d,n) \triangleq \mathsf{tx.type} = \mathit{pay} \text{ and } \mathsf{tx.val} = v \text{ and } \mathsf{tx.asst} = \mathsf{Algo} \text{ and } \mathsf{tx.rcv} = \mathsf{a} \text{ and } \mathsf{tx.fv} \,\%\, p = 0 \text{ and } \mathsf{tx.lv} = \mathsf{tx.fv} + d \text{ and } \mathsf{tx.lx} = n$$

The contract accepts only *pay* transactions of v Algos to receiver a. The conditions $\mathsf{tx.fv} \,\%\, p = 0$ and $\mathsf{tx.lv} = \mathsf{tx.fv} + d$ ensure that the contract only accepts transactions with validity interval $[k\,p, k\,p+d]$, for $k \in \mathbb{N}$. The condition $\mathsf{tx.lx} = n$ ensures that *at most* one such transactions is accepted for each time window.

Finite-state machines. Consider a set of users A who want to stipulate a contract whose behaviour is given by a finite-state machine with states $q_0, \ldots, q_n$. We can implement such a contract by representing each state q_i as a script e_i; the current state/script holds the assets, and each state transition $q_i \to q_j$ is a clause in e_i which enables a *close* transaction to transfer the assets to e_j. This clause requires $\mathsf{tx.rcv} = e_j$ — except in case of loops, which cannot be

encoded directly:[17] in this case, we identify the next state as $\mathsf{tx.rcv} = \mathsf{arg}(0)$, also requiring all users in A to sign $\mathsf{arg}(0)$ to confirm its correctness. To ensure that any user in A can trigger a state transition (by firing the corresponding transaction), their signatures must be exchanged before the contract starts. We show an instance of this pattern as the two-phase authorization contract below.

An alternative technique is based on quines. As before, a state transition $q_i \to q_j$ is rendered as a transaction which closes e_i and transfers the balance to e_j. Here, all such scripts e_k have the same code, except for a single state constant k which occurs at a specific offset, and which represents the current state. To verify that $\mathsf{tx.rcv}$ represents a legit next state, e_i requires a witness w such that: (i) $\mathsf{tx.rcv}$ is equal to the hash of w, and the state constant j within w is indeed a next state for i; (ii) $\mathsf{tx.snd}$ is equal to the hash of w', where w' is obtained from w by replacing the state constant j with the current state i. Performing these checks could be possible by using concatenation and substring operators.[18]

Two-phase authorization. We want a contract to allow user c to withdraw some funds, but only if authorized by a and b. We want a to give her authorization first; if b's authorization is not given within $p \geq \Delta_{max}$ rounds, then anyone can fire a transaction to reset the contract to its initial state. We model this contract with two scripts: *P1* represents the state where no authorization has been given yet, while *P2* represents the state where a's authorization has been given. Conceptually, the contract implements a finite-state machine, looping between two states until the contract funds are withdrawn by c.

$$
\begin{aligned}
\mathit{P1} \;\triangleq\;& \mathsf{tx.type} = \mathit{close} \;\text{ and }\; \mathsf{tx.asst} = \mathsf{Algo} \;\text{ and }\; \mathsf{versig}(\mathsf{txid}, \mathsf{arg}(0), \mathsf{a}) \;\text{ and } \\
& \mathsf{tx.rcv} = \mathit{P2} \;\text{ and }\; \mathsf{tx.fv} \,\%\, (4 * p) = 0 \;\text{ and }\; \mathsf{tx.lv} = \mathsf{tx.fv} + \Delta_{max} \\
\mathit{P2} \;\triangleq\;& \mathsf{tx.type} = \mathit{close} \;\text{ and }\; \mathsf{tx.asst} = \mathsf{Algo} \;\text{ and } \\
& \big((\mathsf{versig}(\mathsf{txid}, \mathsf{arg}(0), \mathsf{b}) \;\text{ and }\; \mathsf{tx.rcv} = \mathsf{c}) \;\text{ or } \\
& \;(\mathsf{versig}(\mathsf{arg}(0), \mathsf{arg}(1), \mathsf{a}_1) \;\text{ and }\; \mathsf{versig}(\mathsf{arg}(0), \mathsf{arg}(2), \mathsf{b}_1) \;\text{ and } \\
& \;\mathsf{tx.rcv} = \mathsf{arg}(0) \;\text{ and }\; \mathsf{tx.fv} \,\%\, (4 * p) = 2 * p \;\text{ and }\; \mathsf{tx.lv} = \mathsf{tx.fv} + \Delta_{max})\big)
\end{aligned}
$$

The scripts *P1* and *P2* use a time window with 4 frames, each lasting p rounds. Script *P1* only accepts *close* transactions which transfer the balance to *P2*; the time constraint ensures that such transactions are sent in the first time frame. The script *P2* accepts two kinds of transactions: *(a)* transfer the balance to c, using an authorization by b; *(b)* transfer the balance to *P1*, in the 4^{th} time frame. Note that in *P2* we cannot use the (intuitively correct) condition $\mathsf{tx.rcv} = \mathit{P1}$, as it would introduce a circularity. Instead, we apply the state machines technique described above: we require $\mathsf{tx.rcv} = \mathsf{arg}(0)$, with $\mathsf{arg}(0)$ signed by both a and b,[19] and assume that these signatures are exchanged before the contract starts.

[17] This is because Algorand contracts cannot have circular references: contract accounts are referenced by script hashes, and no script can depend on its own hash.

[18] In Algorand, these operators are available only for `LogicSigVersion` ≥ 2.

[19] We use other key pairs a_1 and b_1 to avoid confusion with the signatures on txid.

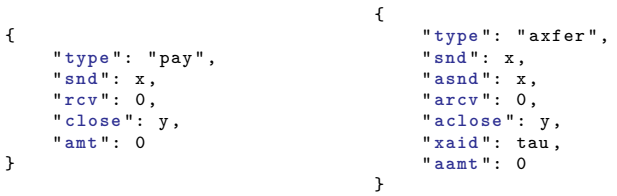

Fig. 4: Translation of a *close* transaction (left: $\tau = \mathsf{Algo}$, right: $\tau \neq \mathsf{Algo}$).

4 From the formal model to concrete Algorand

We now discuss how to translate transactions and scripts in our model to concrete Algorand. We first sketch how to compile our scripts into TEAL. The compilation of most constructs is straightforward. For instance, a script $e + e'$ is compiled by using the opcode `+`, and similarly for the other arithmetic and comparison operators, and for the cryptographic primitives. The logic operators `and`, `or` are compiled via the opcode `bnz`, to obtain the short-circuit semantics. The `not` operator is compiled via the opcode `!`. The operator $\mathsf{txid}(n)$ is compiled as `gtxn n TxID`, txlen is compiled as `global GroupSize`, txpos is compiled as `txn GroupIndex`, and $\mathsf{arg}(n)$ as `arg n`.

Finally, compiling the script $\mathsf{tx}(n).\mathsf{f}$ depends on the field f. If f is fv, lv, or lx, then the compilation is `gtxn n i`, where `i` is, respectively, `FirstValid`, `LastValid`, or `Lease`. For the other cases of f, the compilation of $\mathsf{tx}(n).\mathsf{f}$ generates a TEAL script which computes f by decoding the concrete Algorand transaction fields, and making them available in the scratch space. This decoding is detailed in Table 2 in [14]. From the same table we can also infer how to translate transactions in the model to concrete Algorand transactions. For instance, translating a transaction of the form:

$$\{\mathsf{type} : \mathit{close},\ \mathsf{snd} : \mathbf{x},\ \mathsf{rcv} : \mathbf{y},\ \mathsf{asst} : \tau\}$$

results in the concrete transaction in Fig. 4 (where we omit the irrelevant fields).

Our modelling approach is supported by a prototype tool, called `secteal` (*sec*ure *TEAL*), and accessible via a web interface at:

http://secteal.cs.stir.ac.uk/

The core of the tool is a compiler that translates smart contracts written as expressions, based on the script language (§2.5), into executable TEAL bytecode. In its current form, `secteal` supports experimentation with our model, and is provided with a series of examples from §3.2. Users can also compile their own `secteal` contracts, paving the way to a declarative approach to contract design and development. `secteal` is a first building block toward a comprehensive IDE for the design, verification, and deployment of contracts on Algorand.

5 Conclusions

This work is part of a wider research line on formal modelling of blockchain-based contracts, ranging from Bitcoin [12, 28, 32] to Ethereum [18, 23–26, 31], Cardano [19], Tezos [17], and Zilliqa [33]. These formal models are a necessary prerequisite to rigorously reason on the security of smart contracts, and they are the basis for automatic verification. Besides modelling the behaviour of transactions, in §3.1 we have proposed a model of attackers: this enables us to prove properties of smart contracts in the presence of adversaries, in the spirit of long-standing research in the cryptography area [8, 9, 13, 16, 22, 29, 30].

Differences between our model and Algorand Besides not modelling the consensus protocol, to keep the formalization simple, we chose to abstract from some aspects of ASC1, which do not appear to be relevant to the development of (the majority of) smart contracts. First, we are not modelling some transaction fields: among them, we have omitted the fee field, used to specify an amount of Algos to be paid to nodes, and the note field, used to embed arbitrary bitstrings into transactions. We associate a single manager to assets, while Algorand uses different managers for different operations (e.g., the freeze manager for *frz*/*unfrz* and the clawback manager for *rvk*). We use two different transactions types, *pay* and *close*, to perform asset transfers and account closures: in Algorand, a single *pay* transaction can perform both. Note that we can achieve the same effect by performing the *pay* and *close* transactions within the same atomic group. Although Algorand relies on 7 transaction types, the behaviour of some transactions needs to be further qualified by the combination of other fields (e.g., freeze and unfreeze are obtained by transactions with the same type `afrz`, but with a different value of the `AssetFrozen` field). While this is useful as an implementation detail, our model simplifies reasoning about different behaviours by explicitly exposing them in the transaction type. In the same spirit, while Algorand uses different transaction types to represent actions with similar functionality (e.g., transferring Algos and user-defined assets are rendered with different transaction types, `pay` and `axfer`), we use the same transaction type (e.g., *pay*) for such actions. Our model does not encompass some advanced features of Algorand, e.g.: rekeying accounts, key registration transactions (`keyreg`), some kinds of asset configuration transaction (e.g., decimals, default frozen, different managers), and application call transactions.[20] Our script language substantially covers TEAL opcodes with `LogicSigVersion=1`, but for a few exceptions, e.g. bitwise operations, different hash functions, jumps.

Related work Besides featuring an original consensus protocol based on proof-of-stake [20], Algorand has also introduced a novel paradigm of (stateless) smart contracts, which differs from the paradigms of other mainstream blockchains. On the one hand, Algorand follows the *account-based* model, similarly to Ethereum (and differently from Bitcoin and Cardano, which follow the UTXO model).

[20] Application call transactions are used to implement *stateful* contracts, and therefore are outside the scope of this paper.

On the other hand, Algorand's paradigm of stateless contracts diverges from Ethereum's stateful contracts: rather, it resembles Bitcoin's, where contracts are based upon custom transaction redeem conditions. Besides these differences, Algorand natively features user-defined assets, while other platforms render them as smart contracts (e.g., by implementing ERC20 and ERC721 interfaces in Ethereum). Overall, these differences demand for a formal model that is substantially different from the models devised for the other blockchain platforms.

Our formalization of the Algorand's script language is close, with respect to the level of abstraction, to the model of Bitcoin script developed in [12]. Indeed, both works formalise scripts in an expression language, abstracting from the bytecode. A main difference between Algorand and Ethereum is that Ethereum contracts are stateful: their state can be updated by specific bytecode instructions; instead, (stateless) TEAL scripts merely authorize transactions. Consequently, a difference between our model and formal models of Ethereum contracts is that the semantics of our scripts has no side effects. In this way, our work departs from most literature on the formalization of Ethereum contracts, where the target of the formalization is either the bytecode language EVM [24,25,31], or the high-level language Solidity [15,21,27].

Future work Our formal model of Algorand smart contracts can be expanded depending on the evolution of the Algorand framework. In mid August 2020, Algorand has introduced *stateful* ASC1 contracts [2], enriching contract accounts with a persistent key-value store, accessible and modifiable through a new kind of transaction (which can use an extended set of TEAL opcodes). To accommodate stateful contracts in our model, we would need to embed the key-value store in contract accounts, and extend the script language with key-value store updates. The rest of our model (in particular, the semantics of transactions and the attacker model) is mostly unaffected by this extension. Future work could also investigate declarative languages for stateful Algorand smart contracts, and associated verification techniques. Another research direction is the mechanization of our formal model, using a proof assistant: this would allow machine-checking the proofs developed by pencil-and-paper in [14, §D]. Similar work has been done e.g. for Bitcoin [32] and for Tezos [17].

Acknowledgements The authors are partially supported by: Conv. Fondazione di Sardegna & Atenei Sardi project F74I19000900007 *ADAM* (Massimo Bartoletti); The Data Lab, Innovation Center (Cristian Lepore); EU Horizon 2020 project 830929 *CyberSec4Europe*, and Industriens Fonds Cyberprogram 2020-0489 *Security-by-Design in Digital Denmark (Sb3D)* (Alceste Scalas), and MIUR PON *Distributed Ledgers for Secure Open Communities* (Roberto Zunino).

References

1. Algorand developer docs (2020), https://developer.algorand.org/docs/
2. Algorand developer docs: stateful smart contracts (2020), https://developer.algorand.org/docs/features/asc1/stateful/

3. Algorand developer docs: Transaction Execution Approval Language (TEAL) (2020), https://developer.algorand.org/docs/reference/teal
4. ASC1 coherence-checking experiments (2020), https://github.com/blockchain-unica/asc1-experiments
5. PyTeal: Algorand smart contracts in Python (2020), https://github.com/algorand/pyteal
6. Transaction execution approval language (TEAL) specification (2020), https://developer.algorand.org/docs/reference/teal/specification/
7. Alturki, M.A., Chen, J., Luchangco, V., Moore, B.M., Palmskog, K., Peña, L., Rosu, G.: Towards a verified model of the Algorand consensus protocol in Coq. In: Formal Methods Workshops. Lecture Notes in Computer Science, vol. 12232, pp. 362–367. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_27
8. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via Bitcoin deposits. In: Financial Cryptography Workshops. LNCS, vol. 8438, pp. 105–121. Springer (2014). https://doi.org/10.1007/978-3-662-44774-1_8
9. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P. pp. 443–458 (2014). https://doi.org/10.1109/SP.2014.35, first appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784
10. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM **59**(4), 76–84 (2016). https://doi.org/10.1145/2896386
11. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: SoK: unraveling Bitcoin smart contracts. In: POST. LNCS, vol. 10804, pp. 217–242. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6
12. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Financial Cryptography and Data Security. LNCS, vol. 10957. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6
13. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS. LNCS, vol. 9879, pp. 261–280. Springer (2016). https://doi.org/10.1007/978-3-319-45741-3_14
14. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of Algorand smart contracts. CoRR **abs/2009.12140v3** (2020), https://arxiv.org/abs/2009.12140v3
15. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: Cryptocurrencies and Blockchain Technology. LNCS, vol. 11737, pp. 233–243. Springer (2019). https://doi.org/10.1007/978-3-030-31500-9_15
16. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. LNCS, vol. 8617, pp. 421–439. Springer (2014). https://doi.org/10.1007/978-3-662-44381-1_24
17. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Workshop on Formal Methods for Blockchains. LNCS, vol. 12232, pp. 368–379. Springer (2019). https://doi.org/10.1007/978-3-030-54994-7_28
18. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-

Beguelin, S.: Formal verification of smart contracts. In: PLAS (2016). https://doi.org/10.1145/2993600.2993611
19. Chakravarty, M.M., Chapman, J., MacKenzie, K., Melkonian, O., Jones, M.P., Wadler, P.: The extended UTXO model. In: Financial Cryptography Workshops. LNCS, vol. 12063, pp. 525–539. Springer (2020). https://doi.org/10.1007/978-3-030-54455-3_37
20. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theoretical Computer Science **777**, 155–183 (2019). https://doi.org/10.1016/j.tcs.2019.02.001
21. Crafa, S., Pirro, M.D., Zucca, E.: Is Solidity solid enough? In: Financial Cryptography Workshops. LNCS, vol. 11599, pp. 138–153. Springer (2019). https://doi.org/10.1007/978-3-030-43725-1_11
22. Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: A fair protocol for data trading based on bitcoin transactions. Future Generation Computer Systems (2017). https://doi.org/10.1016/j.future.2017.08.021
23. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of Ethereum smart contracts. In: CAV. LNCS, vol. 10981, pp. 51–78. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_4
24. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Principles of Security and Trust (POST). LNCS, vol. 10804, pp. 243–269. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6_10
25. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.M., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the Ethereum Virtual Machine. In: IEEE Computer Security Foundations Symposium (CSF). pp. 204–217. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00022
26. Hirai, Y.: Defining the Ethereum Virtual Machine for interactive theorem provers. In: Financial Cryptography Workshops. LNCS, vol. 10323, pp. 520–535. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0_33
27. Jiao, J., Kan, S., Lin, S., Sanán, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: Executable operational semantics of Solidity. In: IEEE Symposium on Security and Privacy. pp. 1695–1712. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00066
28. Klomp, R., Bracciali, A.: On symbolic verification of Bitcoin's script language. In: Workshop on Cryptocurrencies and Blockchain Technology (CBT). LNCS, vol. 11025, pp. 38–56. Springer (2018). https://doi.org/10.1007/978-3-030-00305-0_3
29. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS. pp. 30–41 (2014). https://doi.org/10.1145/2660267.2660380
30. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015). https://doi.org/10.1145/2810103.2813712
31. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS. pp. 254–269 (2016). https://doi.org/10.1145/2976749.2978309
32. Rupić, K., Rozic, L., Derek, A.: Mechanized formal model of Bitcoin's blockchain validation procedures. In: Workshop on Formal Methods for Blockchains (FMBC@CAV). OASIcs, vol. 84, pp. 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASIcs.FMBC.2020.7
33. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. Proc. ACM Program. Lang. **3**(OOPSLA), 185:1–185:30 (2019). https://doi.org/10.1145/3360611