

Algorand Protocol Description

Argimiro, CoinFabrik

March 8, 2023

Contents

1	Introduction	1
2	Player accounts	2
3	Ledger definition	2
4	Node as a State Machine	3
5	Main algorithm	4
5.1	Block Proposal	4
5.2	Soft Vote	5
5.3	HandleProposal	6
5.4	HandleVote	6
5.5	HandleBundle	6
5.6	Recovery Attempt	7
5.7	Fast Recovery Attempt	7
6	Subroutines	8

1 Introduction

We aim to describe how the Algorand protocol works. This document is meant to reflect how the Algorand mainnet is behaving today. We relied on Algorand's documents ([GHM⁺17], [Mic16] and Algorand's official specifications) but consulted the node's code and probed the network when information was unclear or unavailable.

Algorand is a proof-of-stake blockchain cryptocurrency protocol using a decentralized Byzantine Agreement protocol that leverages pure proof of stake for consensus. The protocol maintains a ledger that is modified via consensus. In particular, this ledger encodes how many ALGOs (the native token) holds each account. Blocks encode the status of the ledger. Starting on the genesis block (round 0) that encodes the first state of the network, on each round the participation nodes vote on what will the next block be. Their voting power is proportional to the stake (in ALGOs) held by the accounts associated to the node.

At the genesis of the Algorand blockchain, 10Bn ALGO was minted. As of September 2022, circulating supply is approximately 6.9b ALGO, distributed through different forms of ecosystem support and community incentives. The remaining Algos are held by the Foundation in secure wallets assigned to Community and Governance Rewards (53%), Ecosystem Support (36%), and Foundation endowment (11%).

A network is formed with two kind of nodes: relay and participation nodes. Participation nodes can connect only to relay nodes. They listen to one or more participation nodes and they may send a message to a relay node. Relay nodes simply listen to participation nodes connected to them and relay the messages they receive.

Explain how
the network
is formed

In the following document we present a detailed pseudocode interpretation of the program run by an Algorand node managing at least one online (i.e., participating) account. We attempt to explain the different states the node can be in, as well as all kinds of events that cause transitions and network output. This work aims to provide a basis for an Algorand blockchain simulator currently in development.

2 Player accounts

A player is a unique participant of the protocol. We will use the terms player and account indistinctly throughout this document. A player is uniquely identified by a 256-bit string I called an address. Furthermore, they possess a certain balance (an unsigned 64 bit integer), and a pair of participation keys set up according to the key scheme. An account may be online, offline or nonparticipating. An account is offline when it has not sent an online transaction, that creates a participation key and a set of ephemeral keys to be used at each round and destroyed at the end. Also, an account has a set of VRF keys, computed at each round. TODO: keep describing player. Describe register process, online / offline / nonpart attributions)

3 Ledger definition

A ledger is a sequence of states which comprise the common information established by some instantiation of the Algorand protocol. Given a Ledger instance, its immutable characteristics are (...) A ledger state then is defined by the following parameters:

A Ledger entry e (also known as block) is a data structure that models a ledger's state transition. PARAPHRASE DAPA The data in a block is divided between the block header and its block body. The block header contains the following components: • The block's round, which matches the round of the state it is transitioning into. (The block with round 0 is special in that this block specifies not a transition but rather the entire initial state, which is called the genesis state. This block is correspondingly called the genesis block.) The round is stored under msgpack key `rnd`. • The block's genesis identifier and genesis hash, which match the genesis identifier and hash of the states it transitions between. The genesis identifier is stored under msgpack key `gen`, and the genesis hash is stored under msgpack key `gh`. • The block's upgrade vote, which results in the new upgrade state. The block also duplicates the upgrade state of the state it transitions into. The msgpack representation of the components of the upgrade vote are described in detail below. • The block's timestamp, which matches the timestamp of the state it transitions into. The timestamp is stored under msgpack key `ts`. • The block's seed, which matches the seed of the state it transitions into. The seed is stored under msgpack key `seed`. • The block's reward updates, which results in the new reward state. The block also duplicates the reward state of the state it transitions into. The msgpack representation of the components of the reward updates are described in detail below. • A cryptographic commitment to the block's transaction sequence, described below, stored under msgpack key `txn`. 4 • A cryptographic commitment, using SHA256 hash function, to the block's transaction sequence, described below, stored under msgpack key `txn256`. • The block's previous hash, which is the cryptographic hash of the previous block in the sequence. (The previous hash of the genesis block is 0.) The previous hash is stored under msgpack key `prev`. • The block's transaction counter, which is the total number of transactions issued prior to this block. This count starts from the first block with a protocol version that supported the transaction counter. The counter is stored in msgpack field `tc`. • The block's expired participation accounts, which contains an optional slice of public keys of accounts. These accounts are expected to have their participation key expire by the end of the round (or was expired before the current round). The msgpack representation of the components are described in detail below. The slice is stored in msgpack key `partupdrmv`. The block's expired participation accounts slice is valid as long as the participation keys of all the accounts in the slice are expired by the end of the round or were expired before, the accounts themselves would have been online at the end of the round

if they were not included in the slice, and the number of elements in the slice is less or equal to 32. A block proposer may not include all such accounts in the slice and may even omit the slice completely. The block body is the block's transaction sequence, which describes the sequence of updates (transactions) to the account state and box state. A block is valid if each component is also valid. (The genesis block is always valid). Applying a valid block to a state produces a new state by updating each of its components.

4 Node as a State Machine

We model each player as a finite state machine. In order to do so, we will first define and explain all variables involved in a given state a player is able to be in or transition to. A node, the network entity running the protocol, will then be "playing for" one or more online accounts that it manages.

Let round and period number, r and p , be unsigned 64 bit integers. Let the step number, s , be an unsigned 8 bit integer. For convenience and readability, the following aliases for step number are defined:

- $s = 0 \equiv \textit{proposal}$
- $s = 1 \equiv \textit{soft}$
- $s = 2 \equiv \textit{cert}$
- $s \in [3, 252] \equiv \textit{next}_{s-3}$
- $s = 253 \equiv \textit{late}$
- $s = 254 \equiv \textit{redo}$
- $s = 255 \equiv \textit{down}$

A proposal-value is a tuple $v = (I, p, \text{Hash}(e), \text{Hash}(\text{Encoding}(e)))$ where I is an address (the "original proposer"), p is a period (the "original period"), Hash is some cryptographic hash function (usually SHA512/256), Encoding is performed in msgpack, and e is a given ledger entry (as described in the previous section). The special proposal where all fields are the zero-string is called the bottom proposal \perp .

Consider now tuples of the form (I, r, p, s, v) , where v is a value. We will call this kind of construct a vote, noting it $\text{Vote}(r, p, s, v)$, read as a vote for value ' v ' at round r , period p and step s .

We now consider ledger entries of the form $(...)$. A tuple of the form $()$ is called a proposal or proposal payload, noted $\text{Proposal}(v)$.

Given all of the above data types and constructs, a node can be modelled as a finite state machine. A state is given by a 3-tuple of integers, (r, p, s) , two sets P and V of objects which we will call observed *proposal* and *vote* values, a special value v_{prime} which we call the pinned value (see recovery algorithms), a Ledger L and a balance table BT , a table of public addresses for all accounts holding any stake in the network, and their balances. A Ledger is an ordered chain of entries, and fundamentally it represents state changes in the balance table. The round number can be thought of as an index into the ledger, where the event of observing a new round cannot be decoupled from the event of adding an entry (block) to the ledger.

5 Main algorithm

Algorithm 1 Main node algorithm

```

1: function EventHandler(Event ev)
2:   if ev is TimeoutEvent then
3:     time  $\leftarrow$  ev.time
4:     if time = 0 then
5:       BlockProposal()
6:     else if time = FilterTimeout(p) then
7:       SoftVote()
8:     else if time =  $\max\{4\lambda, A\} \vee \text{time} = \max\{4\lambda, A\} + 2^{st-3}\lambda + r, st \in [4, 252], r \in$ 
9:        $[0, 2^{st-3}\lambda]$  then
10:      Recovery()
11:    else if  $\exists k \in \mathbb{Z}, r \sim U(0, \lambda_f), \text{time} = k\lambda_f + r$  then
12:      FastRecovery()
13:    else
14:      msg  $\leftarrow$  ev.msg
15:      if msg is Proposal p then
16:        HandleProposal(p)
17:      else if msg is Vote v then
18:        HandleVote(v)
19:      else if msg is Bundle b then
20:        HandleBundle(b)

```

Algorand nodes operate based on events, which are the only means through which the node state machine can transition and produce outputs. Whenever an event is received, the node will produce a state change and will typically generate an output that will be broadcasted or relayed to other nodes on the network.

There are two types of events: Timeout events and Message events. Timeout events are produced when the node's internal clock reaches a certain time since the start of the current period, while Message events are generated in response to messages sent by other network actors.

The main algorithm describes how the event handling mechanism works. If the event is a timeout and a new period has just started, the node will run the block proposal step. After a certain amount of time, the node will attempt to soft vote, selecting the best received proposal value based on the lowest proposer hash criteria and broadcasting its votes to the network if selected as part of the soft voting committee.

If the event is a message reception, the node will handle it according to its context. For example, a soft vote outside of the soft voting step could produce a next vote, but it could not produce a certification vote without a period change and another round of voting.

After a certain amount of time, the node will start the recovery steps and next vote a value or an empty block if it observes the required threshold votes. Finally, after another specified amount of time, the block will attempt a FastRecovery.

More details on the specifics of each handler can be found in their respective subsections.

5.1 Block Proposal

Algorithm 2 Block proposal

```
1: function BlockProposal
2:   for  $a \in A$  do
3:      $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, \text{proposal},$ 
                                      $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
4:     if  $j > 0$  then
5:       if  $p = 0 \vee \exists s' > 2 / \text{Bundle}(r, p - 1, s', \perp) \subset V$  then
6:          $e \leftarrow \text{AssembleBlock}()$ 
7:          $v \leftarrow \text{Proposal}_{\text{value}}(e)$ 
8:          $\text{Broadcast}(\text{Vote}(I, r, p, \text{proposal}, v))$ 
9:          $\text{Broadcast}(e)$ 
10:      else
11:         $v \leftarrow v_0 | \exists s, \text{Bundle}(r, p - 1, s, v_0) \subset V$ 
12:         $\text{Broadcast}(\text{Vote}(I, r, p, \text{proposal}, v))$ 
13:        if  $\text{Proposal}(v) \in P$  then
14:           $\text{Broadcast}(\text{Proposal}(v))$ 
```

Description:

The node manages multiple accounts and executes a "play" for each account, as each account represents a distinct player in the system. The node runs a sortition process for the proposal step of each account. If the account is selected to propose (i.e., if the account has more than 0 votes for the specific context), the node proceeds with the proposal step and period.

If the check on line 5 passes (i.e., if it is the first period or an empty block has been observed during the recovery step of the previous period), the node assembles a new block (e), calculates the proposal value (v), and broadcasts a vote for v on step 0 along with the full block (e).

If the check on line 5 fails (i.e., it is not the first period and no empty block was observed during the recovery step of the previous period), the period change must have been triggered by observing a bundle for a specific value v_0 . The node broadcasts a vote for this value.

If the actual proposal (i.e., the full ledger entry) is available, the node then broadcasts it in a separate message.

5.2 Soft Vote

Algorithm 3 Soft Vote

```
1: function SoftVote
2:    $\text{lowestObservedHash} \leftarrow \infty$ 
3:    $v \leftarrow \perp$ 
4:   for  $\text{vote} \in V, \text{vote.step} = \text{proposal}$  do
5:      $\text{priorityHash} \leftarrow \min_{i \in [0, j)} H(\text{vote.priority} || i)$ 
6:     if  $\text{priorityHash} < \text{lowestObservedHash}$  then
7:        $\text{lowestObservedHash} \leftarrow \text{priorityHash}$ 
8:        $v \leftarrow \text{vote.v}$ 
9:   for  $a \in \text{Accounts}$  do
10:     $\langle j, \text{hash}, \pi \rangle \leftarrow \text{sortition}(a, \text{soft})$ 
11:    if  $j > 0 \wedge \text{lowestObservedHash} < \infty$  then
12:       $\text{GOSSIP}(\text{Vote}(r, p, \text{soft}, v))$ 
13:      if  $\text{Proposal}(v) \in P$  then
14:         $\text{GOSSIP}(\text{Proposal}(v))$ 
```

Description:

The soft vote stage (also known as "filtering")...

5.3 HandleProposal

Algorithm 4 HandleProposal

```

1: function HandleProposal(Proposale)
2:    $v \leftarrow \text{Proposal}_{\text{value}}(e)$ 
3:   if IsCommittable( $v$ )  $\wedge s \leq \text{cert}$  then
4:     for  $a \in A$  do
5:        $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(\text{ctx}, r), t, \text{cert},$ 
                                      $\text{getSortition}_w(\text{ctx}, r, a_{pk}), \text{getSortition}_W(\text{ctx}, r))$ 
6:       if  $j > 0$  then
7:         Broadcast(Vote( $a.I, r, p, \text{cert}, v$ ))

```

Description:

5.4 HandleVote

Algorithm 5 HandleVote

```

1: function HandleVote(Votev)
2:   if notVerifyVote( $v$ ) then
3:     if then

```

Description:

5.5 HandleBundle

Algorithm 6 HandleBundle

```

1: function HandleBundle(Bundleb)
2:   if VerifyBundle( $b$ )  $\wedge b.r = r \wedge b.p \geq p - 1$  then
3:     for  $\text{vote} \in b$  do
4:       HandleVote( $\text{vote}$ )

```

Description:

5.6 Recovery Attempt

Algorithm 7 Recovery

```

1: function Recovery
2:    $s \leftarrow next_s$ 
3:   ResynchronizationAttempt()
4:   for  $Account a \in A$  do
5:      $\langle sorthash, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, next_s,$ 
                                    $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
6:     if  $j > 0$  then
7:       if  $\exists v | IsCommitable(v)$  then
8:         GOSSIP(Vote( $I, r, p, next_s, v$ ))
9:       else if  $\nexists s_0 > cert | Bundle(r, p-1, s_0, \perp) \subseteq V \wedge$ 
                $\exists s_1 > cert | Bundle(r, p-1, s_1, \bar{v}) \subseteq V$  then
10:        GOSSIP(Vote( $I, r, p, next_s, \bar{v}$ ))
11:       else
12:        GOSSIP(Vote( $I, r, p, next_s, \perp$ ))

```

Description:

5.7 Fast Recovery Attempt

Algorithm 8 FastRecovery

```

1: function FastRecovery
2:   ResynchronizationAttempt()
3:   for  $Account a \in A$  do
4:     if  $IsCommitable(v)$  then
5:        $\langle sorthash, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, 'late',$ 
                                    $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
6:       if  $j > 0$  then
7:         GOSSIP(Vote( $I, r, p, late, v$ ))
8:       else if  $\nexists s_0 > cert | Bundle(r, p-1, s_0, \perp) \subseteq V \wedge$ 
                $\exists s_1 > cert | Bundle(r, p-1, s_1, \bar{v}) \subseteq V$  then
9:          $\langle sorthash, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, 'redo',$ 
                                    $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
10:        if  $j > 0$  then
11:          GOSSIP()
12:        else

```

Description:

The fast recovery algorithm is executed periodically every integer multiple of λ_F seconds. In it, nodes of the network make a synchronization attempt. In the first section, if there is a value v

6 Subroutines

Algorithm 9 IsCommitable

```

1: function IsCommitable( $Proposal_{value} v$ )
2:   return  $Proposal(v) \in P \wedge Bundle(r, p, soft, v) \subset V$ 

```

Algorithm 10 ResynchronizationAttempt

```

1: function ResynchronizationAttempt
2:    $Val = \perp$ 
3:   if  $\exists v | Bundle(r, p, soft, v) \subset V$  then
4:      $GOSSIP(Bundle(r, p, soft, v))$ 
5:      $val = v$ 
6:   else if  $\exists s_0 > cert | Bundle(r, p - 1, s_0, \perp) \subset V$  then
7:      $GOSSIP(Bundle(r, p, s_0, \perp))$ 
8:   else if  $\exists s_0 > cert, v \neq \perp | Bundle(r, p - 1, s_0, v) \subset V$  then
9:      $GOSSIP(Bundle(r, p, s_0, v))$ 
10:     $val = v$ 
11:   if  $val \neq \perp$  and  $Proposal(v) \in P$  then
12:      $GOSSIP(Proposal(v))$ 

```

Description:

Algorithm 11 ComputeSeed

```

1: function ComputeSeed( $ctx, r, B$ )
2:   if  $B \neq empty\_block$  then
3:     return  $VR_{get_{SK_a}(ctx, r)}(ctx.LastBlock.seed || r)$ 
4:   else
5:     return  $Hash(ctx.LastBlock.seed || r)$ 

```

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number
- B = the block whose seed is being computed

Description:

This subroutine computes the required sortition seed for the given round number, which goes in the proposed block's metadata. If the block is empty, the seed is a hash of the previous block's seed. The $get_{SK_a}(ctx, r)$ helper function gets the relevant account's secret ephemeral keys (according to the signing scheme described in specs, the keys 160 rounds prior to r). This roughly corresponds to the secret key from a round b time before block $r - 1 - (r \bmod R)$, where R is the sortition seed's renewal rate, r is the current round's number, and b is the upper bound for the maximum ammount of time that the network might be compromised.

Returns:

- the computed seed for the given block, ledger context and round

Algorithm 12 getSortitionSeed

```
1: function getSortitionSeed(ctx, r, apk)  
    return ctx.block[r - 1 - (r mod R)].seed
```

Arguments:

- *ctx* = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- *round* = current round number
- *a_{pk}* = the account's public key for the look up table

Description:

This helper function gets the relevant sortition seed for the current round *r*, according to the seed lookback parameter *R*. Conceptually, it corresponds with the seed computed *R* rounds prior to *r*, refreshed every *R* rounds.

Returns:

- a sortition seed to be used in the round *r*

Algorithm 13 getSortitionWeight

```
1: function getSortitionw(ctx, round, apk)  
    return ctx.balanceTable[r - (R + SL)] [apk]
```

Arguments:

- *ctx* = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- *r* = current round number
- *a_{pk}* = the account's public key for the look up table

Description:

This helper function retrieves the stake *R* + *SL* rounds prior to *r*, for an account with public key *a_{pk}*

Returns:

- the relevant account's stake

Algorithm 14 getSortitionTotalStake

```
1: function getSortitionW(ctx, r)  
    return  $\sum_{a_{pk} \in ctx.balanceTable[r-(R+SL)]} balanceTable[r-(R+SL)][a_{pk}]$ 
```

Arguments:

- *ctx* = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- *r* = current round number

Description:

This helper function returns the sum of all stake for *R* + *SL* rounds prior to *r*.

Returns:

- the total stake at play in the relevant round (according to lookback parameters)

Algorithm 15 Sortition

```
1: function Sortition( $sk, seed, \tau, role, w, W$ )
2:    $\langle hash, \pi \rangle \leftarrow \text{VRF}(seed || role)$ 
3:    $p \leftarrow \frac{\tau}{W}$ 
4:    $j \leftarrow 0$ 
5:   while  $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^j B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p))$  do
6:      $j++$ 
   return  $\langle hash, \pi, j \rangle$ 
```

Arguments:

- sk = a user's secret key (an ephemeral key for the given round, according to key specs)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)
- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:

The Sortition procedure is one of the most important subroutines in the main algorithm, as it is used in multiple stages and contexts. Generally, it manages to pseudo-randomly but verifiably (through the use of a Verifiable Random Function) select a user with probability proportional to their weight (i.e., stake) by returning a j parameter, which indicates the number of times that specific user was chosen. Algorithmically, every monetary unit the user has is considered a "sub-user", and then each one of them is selected with probability $p = \frac{\tau}{W}$, where τ is the expected amount of users to be selected for the given role. The semi-open interval $[0, 1)$ is then split into consecutive intervals using an accumulated binomial distribution, and wherever the fraction $\frac{hash}{2^{hashlen}}$ belongs to the interval, that's the amount of chosen sub-users for the subroutine caller.

Returns:

- an integer j that will be positive (larger than 0) if the user has been selected, and its size corresponds to the amount of sub-users for a given committee member

Algorithm 16 VerifySortition

```
1: function VerifySortition( $pk, seed, \tau, role, w, W$ )
2:   if  $\neg \text{VerifyVRF}_{pk}(hash, \pi, seed || role)$  then return 0
3:    $p \leftarrow \frac{\tau}{W}$ 
4:    $j \leftarrow 0$ 
5:   while  $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^j B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p))$  do
6:      $j++$ 
   return  $j$ 
```

Arguments:

- pk = a user's public key (their address)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)

- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:

The sortition verification procedure takes Sortition's output and utilizes VRF properties to verify the validity of said output. Once the check is passed, it repeats Sortition's sub-user selection procedure, and outputs the amount of times the user was chosen according to their respective observed stake.

Returns:

- an integer j that will be positive (larger than 0) if the user has been selected, and it's size corresponds to the amount of sub-users for a given committee member

Appendix: Notation

- R = sortition seed renewal rate (in number of rounds). Set to 2 in the specs. as of December 2022
- MAX_STEPS = maximum number of allowed steps in the main algorithm. Defined as 255 in the specs.
- τ_{step} = expected number of members on a regular step committee
- τ_{final} = expected number of members on a final consensus achieving committee
- T_{step} = fraction of expected members for a voting committee on a given step
- T_{final} = fraction of expected members for a voting committee on the final step (step = 2). If observed, achieves final consensus on the given block hash
- $\lambda_{proposal}$ = time interval for the node to accept block proposals, after which it chooses the observed block with the highest priority (lowest hash)
- λ_{block} = waiting time for the full block to be received once decided by vote. If no full block is received, the node falls back to the empty block.
- SL = the account balances lookback interval, in number of rounds (integer). Set to 320 in specs.
- Public keys a_{pk} for every account linked to the node (doubles as their respective addresses)
- Public participation keys $a_{p-partkey}$ for all accounts registered as online (private participation keys are kept securely by respective users and should not be directly accesible by the node)
- Two levels of ephemeral keys, signed according to the signing scheme described in specs. (where first level keys are signed with the participation keys, and second level (aka. leaf) ephemeral keys are signed with first level ephemeral keys). These keys are used for actual participation in the consensus (abstracted as a_{sk} in the following code) They are ephemeral because they live for a single round, after which they are deleted
- A balance table *BalanceTable* for all accounts linked to the node, for the round $SL + R$ rounds before the one currently running

References

- [GHM⁺17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.
- [Mic16] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.