Algorand Protocol Description

Argimiro, CoinFabrik

June 12, 2023

Contents

1 Introduction **1**

2 Node as a State Machine **2**

3 Main algorithm **2**
3.1 Block Proposal 3
3.2 Soft Vote 4
3.3 HandleProposal 5
3.4 HandleVote 6
3.5 HandleBundle 7
3.6 Recovery Attempt 7
3.7 Fast Recovery Attempt 8

4 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 assoicated 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 Node as a State Machine

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, we define the following aliases for step number:

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

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 of objects P and V which we will call observed *proposal* and *vote* values, 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 blocks.

3 Main algorithm

Algorithm 1 Main node algorithm

```
1: function EventHandler(Event ev)
2:     if ev is TimeoutEvent then
3:         time ← ev.time
4:         if time = 0 then
5:             BlockProposal()
6:         else if time = FilterTimeout(p) then
7:             SoftVote()
8:         else if time = max{4λ, A}∨time = max{4λ, A}+2^(st−3)λ+r, st ∈ [4, 252], r ∈ [0, 2^(st−3)λ]
   then
9:             Recovery()
10:        else if time = kλ_f + r, k ∈ ℤ, r ∈ [0, λ_f] then
11:            FastRecovery()
12:    else //ev is MessageEvent
13:        msg ← ev.msg
14:        if msg is Proposal p then
15:            HandleProposal(p)
16:        else if msg is Vote v then
17:            HandleVote(v)
18:        else if msg is Bundle b then
19:            HandleBundle(b)
```

Events are the only way in which the node state machine is able to both internally transition and produce output. In case an event is not identified as misconstrued or malicious in nature, it will certainly produce a state change, and it will almost certainly cause a receiving node to produce and then broadcast or relay an output, to be consumed by its peers in the network. There are two kinds of events: Timeout events, which are produced once the internal clock of a node reaches a certain time since the start of the current period; and Message events, which is output produced by nodes in response to some stimulus (including the receiving node itself). Internally, we consider the relevant data of an event to be a trigger time, a type and associated data (that is cast and interpreted according to event type).

Furthermore, given a proposal value, we denote Proposal(v) = e such that if v is a valid proposal value (validated according to context).

3.1 Block Proposal

Algorithm 2 Block proposal

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

Description:
Block proposal is the first step, and the starting stage of the algorithm at every cycle. First, on line 2, the node attempts a resynchronization. This only has any effect on periods p ¿ 0. Afterwards, the node loops through all of its managed online accounts, Functionally "playing" for each of the accounts. Any account that is selected as a possible proposer Should broadcast a vote and its corresponding proposal, where the proposalvalue v inside the vote matches the proposal e, which is the bloc assembled by the node.

3.2 Soft Vote

Algorithm 3 Soft Vote

```
1:  function SoftVote
2:      lowestObservedHash ← ∞
3:      v ← ⊥
4:      for vote ∈ V, vote.step = proposal do
5:          priorityHash ← min_{i∈[0,j)} H(vote.priority||i)
6:          if priorityHash < lowestObservedHash then
7:              lowestObservedHash ← priorityHash
8:              v ← vote.v
9:      for a ∈ Accounts do
10:         < j, hash, π > ← sortition(a, soft)
11:         if j > 0 ∧ lowestObservedHash < ∞ then
12:             Broadcast(Vote(r, p, soft, v))
13:             if Proposal(v) ∈ P then
14:                 Broadcast(Proposal(v))
```

Description:
The soft vote stage (also known as "filtering") is run after a timeout of X is observed by the node. Let V* be all proposal votes received. By a priority hash function, this stage performas a filtering action, keeping the lowest hashed value observed. Then, it proceeds to broadcast this value. If the associated proposal, p, is in P, then it is subsequently broadcast too.

3.3 HandleProposal

Algorithm 4 HandleProposal

```
1: function HandleProposal(Proposal e)
2:     v ← Proposal_value(e)
3:     if σ(r + 1, 0) = v then
4:         Relay(e)
5:         return //do not observe, as it's for a future round (we're behind)
6:     if ¬VerifyProposal(e) ∨ e ∈ P then
7:         return //ignore proposal
8:     if v ∉ {σ(r, p), v̄, μ(r, p)} then
9:         return //ignore proposal
10:    Relay(e)
11:    P ← P ∪ e
12:    if IsCommitable(v) ∧ s ≤ cert then
13:        for a ∈ A do
14:            ⟨sorthash, π, j⟩ ← Sortition(a_sk, getSortitionSeed(ctx, r), t, cert,
                                        getSortition_w(ctx, r, a_pk), getSortition_W(ctx, r))
15:            if j > 0 then
16:                Broadcast(Vote(a.I, r, p, cert, v))
```

Description:
The proposal handler is triggered when a node receives a message containing a full proposal. It starts by performing a series of checks, after which it will either ignore the received proposal, discarding it and emitting no output; or relay, observe and produce an output according to the proposal's characteristics and the current context. The first check (lines 3:5) is a special case, where if the proposal is from the next round's first period, the node relays it and then ignores it for the purpose of the current round. Whenever the node catches up (observes a round change), and only if necessary, it will request this proposal back from the network. Lines 6:7 check if the proposal is invalid, or if it has been observed already. Any one of those conditions are sufficient to discard and ignore the proposal. Finally, on lines 8:9 it checks if the associated proposal value is either of the special values for the current round and period (σ, μ, or the pinned proposal value $\bar{v}$). Any proposal whose proposal value does not match one of these is ignored. Once the checks have been surpassed, the node is ready to relay the proposal and observe it by adding it to the observed proposals set, P (lines 10:11). Once relayed and observed, the proposal is then processed for further output. Here, and only if the proposal value has become commitable and the current executing node's step is lower or equal to a certification step (it's not yet in a recovery step), the node plays for each account performing sortition to select committee members for the certification step. For each selected member, a corresponding certify vote for the current proposal value is cast.

3.4 HandleVote

Algorithm 5 HandleVote

```
1: function HandleVote(Vote vt)
2:     if ¬VerifyVote(vt) then
3:         PenalizeVoter(vt)  //optional
4:         return  //ignore invalid vote
5:     if vt.step = 0 ∧ (vt ∈ V ∨ IsEquivocation(vt)) then
6:         return  //ignore vote, equivocation not allowed in proposal votes
7:     if vt.step > 0 ∧ IsSecondEquivocation(vt) then
8:         return  //ignore vote if it's a second equivocation
9:     if vt.round < r then
10:        return  //ignore vote of past round
11:    if vt.round = r + 1 ∧ (vt.period > 0 ∨ vt.step ∈ (cert, late)) then
12:        return  //ignore vote of next round if non-zero period or next_k step
13:    if vt.round = r ∧ (vt.period ∉ [p − 1, p + 1]∨
14: (vt.period = p + 1 ∧ vt.step ∈ (next_0, late))∨
15: (vt.period = p ∧ vt.step ∈ (next_0, late) ∧ vt.step ∉ [s − 1, s + 1])∨
16: (vt.period = p − 1 ∧ vt.step ∈ (next_0, late) ∧ vt.step ∉ [s̄ − 1, s̄ + 1])) then
17:        return  //ignore vote
18:    V ← V ∪ vt  //observe vote
19:    if vt.step = soft then
20:        if ∃v|Bundle(vt.round, vt.period, soft, v) ⊂ V then
21:            for a ∈ A do
22:                ⟨sorthash, π, j⟩ ← Sortition(a_sk, getSortitionSeed(ctx, r), t, cert,
                                        getSortition_w(ctx, r, a_pk), getSortition_W(ctx, r))
23:                if j > 0 then
24:                    Broadcast(Vote(a.I, r, p, cert, v))
25:    else if vt.step = cert then
26:        if ∃v|Bundle(vt.round, vt.period, cert, v) ⊂ V then
27:            if Proposal(v) ∉ P then
28:                RequestProposal(v) //waits until proposal is obtained
29:            Commit(v)
30:            StartNewRound(vt.round)
31:            GarbageCollect()
32:    else
33:        if ∃v, s|Bundle(vt.round, vt.period, s, v) ⊂ V then
34:            StartNewPeriod(vt.period)
35:            GarbageCollect()
```

Description:
The vote handler is triggered when a node receives a message containing a vote for a given proposal value, round, period and step.

3.5 HandleBundle

Algorithm 6 HandleBundle

```
1: function HandleBundle(Bundle b)
2:     if VerifyBundle(b) ∧ b.r = r ∧ b.p + 1 ≥ p then
3:         for vote ∈ b do
4:             HandleVote(vote)
```

Description:
The bundle handler is invoked whenever a bundle message is received. If the received bundle is valid, its round is the current node's round and is at most one period behind of the node's current period, then the bundle is processed, which is simply calling the vote handler for each vote constituting the bundle. If any of the conditions outlined previously are not met, then the whole bundle is ignored and discarded.

3.6 Recovery Attempt

Algorithm 7 Recovery

```
1: function Recovery
2:     s ← next_s
3:     ResynchronizationAttempt()
4:     for Account a ∈ A do
5:         ⟨sorthash, π, j⟩ ← Sortition(a_sk, getSortitionSeed(ctx, r), t, next_s,
                                        getSortition_w(ctx, r, a_pk), getSortition_W(ctx, r))
6:         if j > 0 then
7:             if ∃v|IsCommitable(v) then
8:                 Broadcast(Vote(I, r, p, next_s, v))
9:             else if ∄s_0 > cert|Bundle(r, p − 1, s_0, ⊥) ⊆ V ∧
                       ∃s_1 > cert|Bundle(r, p − 1, s_1, v̄) ⊆ V then
10:                Broadcast(Vote(I, r, p, next_s, v̄))
11:            else
12:                Broadcast(Vote(I, r, p, next_s, ⊥))
```

Description:

3.7 Fast Recovery Attempt

Algorithm 8 FastRecovery

1: **function** $FastRecovery$
2: $ResynchronizationAttempt()$
3: **for** *Account* $a \in A$ **do**
4: **if** $IsCommitable(v)$ **then**
5: $\langle sorthash, \pi, j \rangle \leftarrow \mathsf{Sortition}(a_{sk}, \mathsf{getSortitionSeed}(ctx, r), t, late, \mathsf{getSortition}_w(ctx, r, a_{pk}), \mathsf{getSortition}_\mathsf{W}(ctx, r))$
6: **if** $j > 0$ **then**
7: $Broadcast(Vote(I, r, p, late, v))$
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: $\langle sorthash, \pi, j \rangle \leftarrow \mathsf{Sortition}(a_{sk}, \mathsf{getSortitionSeed}(ctx, r), t,' redo', \mathsf{getSortition}_w(ctx, r, a_{pk}), \mathsf{getSortition}_\mathsf{W}(ctx, r))$
10: **if** $j > 0$ **then**
11: $Broadcast(Vote(r, p, redo, \bar{v}))$
12: **else**
13: $\langle sorthash, \pi, j \rangle \leftarrow \mathsf{Sortition}(a_{sk}, \mathsf{getSortitionSeed}(ctx, r), t, down, \mathsf{getSortition}_w(ctx, r, a_{pk}), \mathsf{getSortition}_\mathsf{W}(ctx, r))$
14: **if** $j > 0$ **then**
15: $Broadcast(Vote(r, p, down, \perp))$

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

4 Subroutines

Algorithm 9 IsCommitable

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

Arguments:

- $Proposal_{value}$ v = A value to check for commitability

Description:
Checks that the value v is commitable in the current node's context. To be commitable, the two conditions outlined in line 2 have to be met. That is, the corresponding proposal that the value refers to has to be available (have been observed in the current round), and there must be a valid bundle of soft votes for v observed during the current round and period.

Returns:

- A boolean value indicating commitability of the argument v

Algorithm 10 VerifyVote

```
1: function VerifyVote(Vote vt)
2:     valid ← vt.round <= round + 2
3:     if vt.step = 0 then
4:         valid ← valid ∧ vt.v.p_orig <= vt.period
5:         if vt.period = vt.v.p_orig then
6:             valid ← valid ∧ vt.v.I_orig = vt.I
7:     if vt.step ∈ {propose, soft, cert, late, redo} then
8:         valid ← valid ∧ vt.v ≠ ⊥
9:     else if vt.step = down then
10:        valid ← valid ∧ vt.v = ⊥
       //TODO: verificacion de firmas, VRFs y rounds de validez
11:    return valid
```

Description:

Algorithm 11 VerifyProposal

```
1: function VerifyProposal(Proposal p)
```

Description:

Algorithm 12 VerifyBundle

```
1: function VerifyBundle(Bundle b)
```

Description:

Algorithm 13 StartNewRound

```
1: function StartNewRound(uint64 finishedRound)
```

Description:

Algorithm 14 StartNewPeriod

```
1: function StartNewPeriod(uint64 finishedPeriod)
```

Description:

Algorithm 15 GarbageCollect

```
1: function GarbageCollect
```

Description:

Algorithm 16 RequestProposal

```
1: function RequestProposal(Proposal_value v) //ver como hacer el request...como lo resuelve el nodo?
```

Description:

Algorithm 17 PenalizeVoter

1: **function** PenalizeVoter(*Account a*) //ver como resuelve el nodo la desconexion o blacklist de un peer

Description:

Algorithm 18 AssembleBlock

1: **function** AssembleBlock

Description:

Algorithm 19 ResynchronizationAttempt

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

Description:

Algorithm 20 ComputeSeed

1: **function** $ComputeSeed(ctx, r, B)$
2: **if** $B \neq empty\_block$ **then**
3: return $VRF_{get_{SK_a}(ctx,r)}(ctx.LastBlock.seed||r)$
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 $r - 1 - (r \ mod \ R)$, 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 21 getSortitionSeed

1: **function** $getSortitionSeed(ctx, r, a_{pk})$
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 22 getSortitionWeight

1: **function** $getSortition_w(ctx, round, a_{pk})$
return $ctx.balanceTable[r - (R + SL)][a_{pk}]$

Arguments:

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

1: **function** $getSortition_W(ctx, r)$
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 24 Sortition

function Sortition($sk, seed, \tau, role, w, W$)
  $\langle hash, \pi \rangle \leftarrow \mathsf{VRF}(seed||role)$
  $p \leftarrow \frac{t}{W}$
  $j \leftarrow 0$
  while $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, p), \sum_{k=0}^{j+1} \mathsf{B}(k; w, p))$ **do**
    $j \leftarrow j + 1$
  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 ammount 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 25 VerifySortition

function VerifySortition($pk, seed, \tau, role, w, W$)
  if $\neg \mathsf{VerifyVRF}_{pk}(hash, \pi, seed||role)$ **then return** 0
  $p \leftarrow \frac{t}{W}$
  $j \leftarrow 0$
  while $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, p), \sum_{k=0}^{j+1} \mathsf{B}(k; w, p))$ **do**
    $j++$
  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.