Algorithm 1 Block creation

1: **function** $BlockCreation()$
2: $TODO$

The algorand protocol specs don't ensure a specific way or internal order in which transactions are added to the block. In fact, they don't force transactions to be added at all, being entirely possible for a node to propose an empty block. More research is needed on how block data is put together. Block metadata is well documented, but is added into it later in the block proposal stage.

Algorithm 2 Block proposal

1: **function** $BlockProposal(Block\ B,\ Accounts\ A)$
2: $, lowestPriority$
3: **for** $a \in A$ **do**
4: $< sorthash, \pi, j > \leftarrow Sortition(a.sk, seed, t, role, a.w[], W)$
5: **if** $j > 0$ **then**
6: $priority \leftarrow Min_{n \in [1,j]} Hash(sorthash||n)$
7: $lowestPriorityUser \leftarrow a$
8: $GOSSIP\_MESSAGE(priority, SIGNED_{a.sk}(< sorthash, \pi >))$
9: $msgs \leftarrow incomingMsgs[round, step = 0]$
10: **while** $elapsed\ time < proposal\ time$ **do**
11: $incomingMsgs.push(listen())$
12: $lowestHashMsg \leftarrow m \mid m.priority = Min_{m.priority\ \in\ incomingMsgs}\{verifySortition(m.sorthash, m.pi, m.pk)$
13: $seed \leftarrow ComputeSeed(r, lowestHashMsg)$
14: $FullBlockToPropose \leftarrow empty_block(round)$
15: $FullBlockToPropose.seed \leftarrow seed$
16: **while** $elapsedtime < fullblocktime$ **do**
17: $m \leftarrow msgs.next()$
18: $< priority, sorthash, \pi > \leftarrow m$

Algorithm 3 Soft Vote

1: **function** $SoftVote$
2: $hblock \leftarrow CountVotes(ctx, round, REDUCTION\_ONE, Tstep, tstep, lblock + lstep)$
3: $empty\_hash \leftarrow H(Empty(round, H(ctx.last\_block)))$
4: **if** $hblock = TIMEOUT$ **then**
5: $CommitteeVote(ctx, round, REDUCTION\_TWO, tstep, empty\_hash)$ **else**
6: $CommitteeVote(ctx, round, REDUCTION\_TWO, tstep, hblock1)$
7: $hblock \leftarrow CountVotes(ctx, round, REDUCTION\_TWO, Tstep, tstep, lblock + lstep)$
8: **if** $hblock = TIMEOUT$ **then return** $empty\_hash$ **else return** $hblock$

The soft vote stage (step 1, or labeled as reduction in the ASBAC paper) aims to reduce any ammount of potentially conflicting proposed blocks in the stage prior into a binary choice, either a block proposed by a user or an empty one. In the first sub-step, each commitee member gossips the proposed block, and then does a preliminary vote count while it waits for other nodes to catch up. In the second sub-step, committee members vote for the hash that received at least the ammount of votes set by the threshold for this step, or the hash of the default empty block if no hash was observed to receive enough votes. It relies heavily on the assumption that, if the block proposer was honest, most users will get to the soft vote stage with the same hblock parameter. However if the proposer was dishonest, then no single hblock may be popular enough to cross the threshold (and therefore this stage will return an empty_hash). Therefore, there will at most be one non-empty block that can be returned by soft vote for all honest users.

Algorithm 4 CertifyVote

```
1: function CertifyVote(hblock)
2:     step ← 2
3:     while step < 256 do
4:
5:         < hblock, bConfirmed >← BLOCK_STEP(hblock, step)
6:         if bConfirmed then return hblock
7:         step++
8:
9:         < hblock, bConfirmed >← EMPTY_STEP(hblock, step)
10:        if bConfirmed then return hblock
11:        step++
12:
13:        CommonCoinFlipVote(hblock, step)
14:        step++
15:
16:        HangForever()
17:        =0
```

The certify vote stage (which potentially encompasses steps 2 through 255) is the core of the BA* algorithm. The main loop executes a maximum of 254 times (from step 2 to step 255, labeled c̈ertänd d̈own̈respectively on the specs). This is to avoid an attacker indefinitely postponing consensus in a compromised network, giving them a small chance to force consensus on their desired block with each iteration. The main algorithm is comprised of three important procedures: the block step, where, if selected as a committee member, votes for the block selected in the soft voting stage and then attempts to confirm it by listening to other cast votes until a threshold is met. If this step times out and no consensus is reached, it moves on to the empty step. Very similar in nature to the previous one, in this procedure a vote is cast for the value in hblock and then it attempts to confirm an empty block. Finally, if no consensus is reached on the empty block, it arrives to the common coin procedure, where it attempts another vote and vote count for the empty block. If here, no consensus is reached, it uses the pseudo-randomness of the VRF hash output to "flip a coin"by looking at the last bit of the lowest sortition hash owned by observed committee members. This makes it so even tho the coin flip is random, most users will observe the same outcome and will in turn set their hblock values according to the observed bit. In turn, this prevents an adversary from being able to easily predict beforehand what the next vote of a user would be if they were a committee user, and therefore the attacker can't force consensus sending their vote messages to targeted users in a scenario where the voting is split pretty evenly but no block has quite enough votes to push the threshold.

Algorithm 5 BlockConfirmation

```
1: function BlockConfirmation(hblock)
2:     r ← CountVotes(ctx, round, FINAL, Tfinal, tfinal, lstep)
3:     if r = hblock then
4:     return < FINAL, BlockOfHash(hblock) > else
5:     return < TENTATIVE, BlockOfHash(hblock) >
   =0
```

Finally, the proposed block (be it an empty block or a specific one proposed by a designated user) is confirmed and added to the ledger. We introduce the notions of FINAL and TENTATIVE consensus. Specifically, if a block is confirmed in the very first step of the certify vote stage, its vote is cast as FINAL. If enough users reproduce this behavior, the vote count on line 2 for votes cast in this fashion is over the required threshold, and the consensus is labeled as FINAL. This

means that the block is simultaneously added to the ledger and confirmed, as well as immediately confirming any prior blocks that could have been labeled TENTATIVE. If this node is not able to ensure that enough users considered the block represented by hblock as FINAL, the consensus is labeled as TENTATIVE, and it may be confirmed by a later FINAL block. The BlockOfHash() function outputs the full block for the provided hash, either by returning a locally available copy or by requesting it from other users.