Blockchain consensus protocols based on stake

Master Thesis

Timo Bürk

Faculty of Science
at the University of Bern

April 2022

Prof. Dr. Christian Cachin
Luca Zanolini

Cryptology and Data Security Group
Insitute of Computer Science
University of Bern, Switzerland

Abstract

Over the course of the last few years multiple blockchains emerged that feature stake-based consensus protocols. In comparison to the established proof-of-work-based approaches, these protocols have the advantage of being more energy efficient and more scalable. This makes these protocols a topic of interest to study. In this thesis we describe and compare six of the most prominent stake-based consensus protocols, namely Algorand, Cardano, Ethereum 2.0, EOSIO, Neo and COSMOS. We provide descriptions and pseudocode implementations for each protocol and introduce uniformed terminology and notations across all protocols. With this thesis we aim to provide a basis for further analysis of blockchain consensus protocols based on stake.

Contents

1 Introduction

Over the last few years there are more and more blockchains that feature stake-based consensus protocols. The focus on stake-based approaches is largely motivated due to two key advantages: energy-efficiency and scalability. In contrast, established proof-of-work blockchains such as Bitcoin [21] rely on participants racing to solve a puzzle in order to determine the next block producer. Participants can invest more computational power in order to raise their chances of winning the race. This competition for computational power leads to a raising energy consumption. Stake-based blockchain protocols aim to solve this issue by replacing the proof of work with a more energy-friendly design.

Furthermore the difficulty of the puzzle in proof-of-work protocols limits the rate at which new blocks are discovered. This is because a harder puzzle means that it takes more attempts in order to find a block that solves it. Proof-of-stake protocols are less limited in that regard, since the computation of a new block is cheap compared to proof-of-work designs. This means that stake-based consensus protocols have the potential of scaling better in comparison.

These two advantages make blockchain consensus protocols based on stake a compelling subject to analyze. However these stake-based protocols use a wide variety of ways in how they incorporate stake into their consensus. This variety extends even down to the terminology they use to describe similar concepts. This makes it hard to compare the different protocols to each other. Additionally some protocols lack an appropriate documentation, which further hinders analysis. Therefore the main goal of this thesis is to provide descriptions and pseudocode implementations for the most prominent protocols. We make the protocols easier to compare by introducing common terminology and using a common layout for both the descriptions and pseudocode implementations. This allows us to more easily identify commonalities and differences between the protocols.

The thesis is organized as follows. In Chapter 2 we define the terminology and notations we use across all protocols. In addition we give a brief introduction to cryptographic mechanisms that are used in multiple protocols. In Chapter 3 we give an introduction to stake-based consensus protocols and introduce how we group them together. In Chapters 4, 5, 6 and 7 we describe the six protocols we cover in this thesis divided into family specific chapters. The six protocols are Algorand, Cardano, Ethereum 2.0, EOSIO, Neo and COSMOS. In Chapter 8 we compare the protocols to each other and provide an overview of our work. Finally in Chapter 9 we reflect about the thesis and discuss future work.

2 Preliminaries

In this section we present the terms and notations we use for common concepts in stake-based protocols. In addition we give an introduction for verifiable random functions and practical byzantine fault tolerance, which are both used in multiple stake-based protocols that we discuss in this thesis.

2.1 Terminology and notations

Since we are going to look at multiple protocols, that all feature different terminology for similar concepts, we use this section to define a common terminology. If a protocol uses different terms than the ones introduced here, we briefly mention the original terms in the description of those protocols. We will now introduce the terms and definitions we use for each concept.

2.1.1 Stake

Stake is defined as the wealth someone possesses in a blockchain network. This comes in the form of coins or tokens of the cryptocurrency associated with the blockchain. In stake-based protocols someone's influence in the protocol is usually related to the stake they own in the system.

2.1.2 Parties and leaders

We will use the term party for a participant in the proof-of-stake consensus protocol. In the pseudocode a party will be denoted with p_i, where i is the party's index. A special kind of party is the leader denoted by p_l. A leader is a party that is selected to produce the next block for the blockchain. Note that a party is considered to fill a certain role as soon as it has been selected for that role, regardless of whether or not it actually completes the task. Furthermore there are other roles that are unique to a protocol, in that case we will introduce them, when we discuss those protocols.

2.1.3 Blockchain

Since we are dealing with blockchain protocols in all instances, this term is fairly uniformly used already. Therefore this section serves more to give our terms and notations for all concepts related to the blockchain. A blockchain consists of blocks denoted with B that are linked with a hash chain where each block contains the hash h of the previous one. In addition each block contains some data $\mathcal{D}$ that contains transactions denoted by x and in some cases data relevant to the protocol. The latest block in the blockchain is called the *head* of the chain while the first block in every blockchain is called the *genesis block*. Finally, all parties keep track of the current state of the blockchain themselves and store it locally. This local chain is denoted by $\mathcal{B}$.

2.1.4 Epochs and slots

We divide runs of the protocols into epochs and epochs further into slots. With an epoch e_j we denote a period of time, where relevant parameters for the protocol such as stake distribution and votes are stable in regards to the protocol. For example, even though the stake distribution changes as the result of each new block, those changes do not immediately effect the consensus protocol. Rather the stake distribution is only computed once per epoch and stays stable for the duration of the epoch. An epoch is further divided into slots where one slot is the period during which a leader may potentially produce a new block. A slot is denoted by sl.

2.2 Practical byzantine fault tolerance

Practical byzantine fault tolerance (PBFT) is a consensus protocol developed by Castro and Liskov [15]. The purpose of a BFT protocol is for honest parties to reach consensus on a value v despite the presence of byzantine parties. A byzantine party is a participant in the consensus protocol, which acts either arbitrarily or maliciously. What sets PBFT apart from other BFT protocols however is that it only requires the participant to be partially synchronized instead of fully synchronized. This means that PBFT can be applied in less controlled environments such as the internet or a blockchain network. This is why PBFT is often used in stake-based blockchain protocols, where honest parties try to agree on the next block in the presents of adversarial parties.

The PBFT protocol consists of at least one round, where a round consists of three phases. If the participants successfully finish all three phases during a round, they have reached consensus on a value v. If a phase fails then a new round of consensus is started. The three phases of a round are the *pre-prepare*, the *prepare* and the *commit* phase. In the *pre-prepare* phase the leader of the round proposes a value v to the other parties. The leader is usually selected by a leader-selection function, which assigns a unique leader to each round. In the *prepare* phase the other parties validate the value v and signal their agreement by sending a *prepare* message. If more then $2/3$ of the participants agree with the value v, then they move on to the *commit* phase. In the *commit* phase all parties confirm that they saw enough support for value v during the *prepare* phase. They signal their confirmation with a *commit* message. If more than $2/3$ of the participants commit to value v in this way then, the parties reached consensus on v. In order to PBFT protocols to function properly, it is required that more than $2/3$ of the participants are honest.

2.3 Verifiable random function

A verifiable random function (VRF) is used in some of the stake-based consensus protocols we are discussing. It is an important building block for random-selection mechanisms, as for example seen in lottery based approaches. A VRF $(r, \pi) = \mathit{VRF}(m, sk^{vrf})$ is similar to a pseudo-random function (PRF)

$r = \textit{PRF}(m, k)$ in terms of input and output. A PRF produces a pseudo-random number r based on a message m and a key k. The PRF will always produce the same output r for the same input pair (m, k). The same functionality is also present in the VRF. The output r is also a pseudo-random value based on a message m and a secret key sk^{vrf}. This secret key is part of the key pair (vk^{vrf}, sk^{vrf}), where vk^{vrf} is the public key. The key pair (vk^{vrf}, sk^{vrf}) is generated through the $\textit{KeyGen}(\rho)$ function, where ρ denotes the randomness that is used to generate the keys.

To make the VRF verifiable it features a second output, the proof π. This is a non-interactive cryptographic proof that proves that the random value r is indeed the result of the message m and secret key sk^{vrf}. The proof π can be verified by another party by evaluating the $\textit{Verify}(m, r, \pi, vk^{vrf})$ method. Notice that the key used in the *Verify* method is the public key vk^{vrf}. While the verifying party learns whether the random value r is indeed the result of input (m, sk^{vrf}), the verifier will not learn sk^{vrf} in the process.

3
Stake-based consensus overview

In this section we go over the general concepts and ideas behind stake-based consensus protocols. Then we give a brief introduction to the four families of stake-based protocols we encountered during our research. Finally we describe some attacks that stake-based protocols face.

3.1 Stake-based consensus protocols

In stake-based consensus protocols parties try to find consensus on the next block to add to a blockchain. Here stake is defined as the wealth a party owns in a blockchain network. This comes in the form of coins or tokens of the cryptocurrency associated with the blockchain. The stake, which a party possesses, determines the party's influence in the protocol. This means that parties which have more stake also have more influence. The reasoning here is that the more stake a party has the less likely it is to act in a way that devalues that stake. Therefore a wealthy party is more likely to participate honestly in the protocol in order to not jeopardize the health and reputation of the blockchain.

While some stake-based protocols consider a party's entire stake to determine its influence, others require the stake to be invested. This process is called staking. Here a party locks up a portion of its own stake. While the stake is locked up it counts toward the party's influence, but can not be used for any other purpose. In these types of stake-based consensus protocols only parties which invested stake are able to participate.

There are two aspects that a stake-based consensus protocol needs to define. The first is the chain extension mechanism, where a party is selected as a leader to propose the next block. This leader-selection process is usually dependent on a party's stake, but can take many different forms. The second aspect is block finalization. A finalized block is defined as a block that will always be part of the chain and can no longer be reversed or changed. In stake-based protocols block finalization needs to be defined algorithmically instead of probabilistically. This is due to the fact that the computation of new blocks is cheap. Therefore creating an alternative chain or fork that does not contain a certain block is always possible. That is why a stake-based protocol needs a block-finalization mechanism. We now give a brief definition of the four different approaches for stake-based consensus protocols that we cover in this thesis.

Lottery The lottery-based family of proof-of-stake protocols rely on a weighted random selection to choose the leader. In those approaches any party in the blockchain network has the chance of being selected as a leader. However a party's chance of being chosen depends on the party's relative stake in the blockchain network. Therefore wealthier parties are more likely to be selected.

Slashing In slashing-based approaches parties may only participate in the consensus protocol, if they have invested stake. In slashing protocols this locked-up stake actually serves as a security deposit. If a party is caught after it misbehaved during the consensus protocol, then a portion of its locked-up stake is removed. The removal of invested stake after a party misbehaved is called "slashing". Therefore slashing-based protocols have an additional measure to keep parties honest.

Voting Voting-based approaches limit the number of parties that are allowed to participate in the consensus protocol. All parties vote on which candidates should be selected for those spots. In this election a party's voting power is determined by the stake it possesses. The election is repeated after a set period of time, usually after each epoch. Voters are expected to base their vote on the past performance of the candidates and to vote out adversarial parties.

Hybrid-voting-slashing In this voting-slashing-hybrid approach, the participants of the consensus protocol are voted for in an election. If a party wishes to vote it must first lock up a portion of its stake. Its voting power is then determined by the amount of locked-up stake. This stake simultaneously serves as the collateral for the slashing. In this approach voters are held responsible for the actions of their supported candidate. This puts more pressure on the voter to only support candidates that they trust.

3.2 Attacks on stake-based protocols

When looking for possible attacks against proof-of-stake protocols, we can take the attacks against proof-of-work protocols as a reference. Since many of them, such as double spending or selfish-mining, are relevant to blockchain protocols in general. However, as mentioned before the creation of new blocks takes much less effort in stake-based protocols. This opens up more avenues of attack, since even computing long chains takes little effort. Therefore it is feasible to compute a fork that grows faster than the main chain. In this section we will go over a list of attacks against proof-of-stake consensus protocols based on the work of Kiayias et al. [19]. We will first go over the attacks that are present in proof-of-work-based protocols as well and then transition to the attacks that are more specific to proof-of-stake-based protocols.

Double-spending attack The goal of a double spending attack is to issue two conflicting transactions that are both accepted into the blockchain. For example a malicious party may try to issue two transactions that involve the same coin but two different receivers. If both transactions are accepted in the same chain, then that chains state becomes inconsistent.

Transaction denial attack In this attack the adversary aims to prevent that a certain transaction is included in the blockchain. This attack may be aimed at a single transaction or a class of transactions, such as preventing all transaction of a specific party from being confirmed.

Desychronization attack As the name suggests the goal of this attack is to cause an honest party to become desynchronized from the rest of the network. This causes the desynchronized party to perform actions at the wrong time, even tough the party is still operating honestly from its perspective. This can be achieved by deliberately cutting off the targeted party from its means to synchronize with the network or

through the delay of messages. A desychronization may also occur without the presents of an adversarial party.

Eclipse attack Similar to the desychronization attack, the eclipse attack targets a honest party and prevents it from participating in the protocol. The attack aims to cut off a honest party by blocking the communication to the other participants.

51% attack In this scenario an adversary manages to control the majority of the resources that are used in the leader selection process. In the case of proof-of-work protocols this would be computational power, while for the proof-of-stake protocols this is the majority of stake in the system. Controlling a majority of the resources means that an adversary is able to produce the majority of the blocks. If that is the case, then it can outpace the honest parties in the chain extension and can therefore create forks at will.

Bribery attack The intention here is for the adversary to bribe a leader into producing a block for a fork that serves the adversary. Since we are dealing with proof-of-stake protocols, a party's mining power is proportional to the stake that it possesses. This means that parties with more mining power also have more stake in the system, which makes them less prone to bribery. While this does not completely negate the attack for proof-of-stake systems, they are less prone to it compared to proof-of-work systems.

Selfish-mining In this attack an adversary withholds computed blocks. Once a honest party has generated a new block, the attacker can release the withheld blocks to overtake the chain that the honest party extended. This causes the chain containing the honest block to be abandoned. With this mining strategy the attacker can create a chain where the rate of blocks computed by the attacker is higher. Since the mining of a new block is tied to a reward, a higher rate of mined blocks results in a higher rate of rewards.

Grinding attack Grinding attacks are specific to stake-based protocols and aim to influence the random leader selection process in the favor of the attacker. This is possible if the randomness for the leader selection is based on data that the leader can influence, such as header information or block content. If this is the case then an attacker, that was selected as leader, can compute multiple blocks and pick the one that maximizes their chances to be selected as leader in future selections. Therefore increasing the probability that the attacker is selected above of that of the intended distribution.

Long-range attack This is another attack that is only feasible in proof-of-stake systems. Since the production of blocks takes little effort, an adversary can locally compute a separate chain, perhaps even beginning at the genesis block. In this local run of the protocol the adversary is the only active participant in the protocol. Therefore new blocks are only produced when the adversarial party is selected as the leader. This way an adversary can produce a fork that is longer than the current chain. This enables the adversary to mount a double-spending attack by submitting a transaction to the main chain and waiting for it to be confirmed. After that, the locally computed chain can be presented to invalidate the confirmed transaction.

Nothing-at-stake attack As with the long-range attack the "nothing at stake" problem arises because the production of a block costs only a small amount of effort. Therefore a malicious party can maintain and produce blocks for multiple chains. Furthermore a party that produced a block for all available forks gains the rewards for it, no matter which fork is selected as the new main chain. Which means that they do not have an incentive to resolve a fork. A proof-of-stake system that is prone to the "nothing at stake" problem will have difficulties with resolving forks, which in turn will make it vulnerable to double spending attacks.

Past-majority attack This attack combines the ideas of a 51% attack and a long-range attack. The premise is that an adversary manages to corrupt a party that possessed a lot of stake in the past, but does not so in the present. This can create a situation, where even though the adversary does not possess a majority of the stake in the present, there is a period in the past where the adversary now possesses a majority for. This allows the attacker to recompute the blocks for all selected leaders it now controls. Therefore creating a new fork in the system.

4
Lottery-based proof-of-stake protocols

In this section we have a closer look at lottery-based proof-of-stake protocols. We first give a general definition of lottery-based protocols and then take a closer look at two representatives of this family.

The central idea behind lottery-based proof-of-stake protocols is close to those of proof-of-work designs. Proof-of-work protocols can be thought of as a random selection or lottery. In this case a party's chance of being selected as leader is proportional to its computational power. Lottery-based proof-of-stake protocols use the same concept, but replace the computational power with the party's stake in the system. Note that a major challenge in proof-of-stake protocols is to design the selection process itself, since the selection has to be random and verifiable for all parties.

The two lottery-based protocols we look at are Algorand and Cardano. Both of them use a weighted random selection for key roles in the protocol, where the weight is a party's stake in the system. Therefore they have similarities and shared building blocks in regards to the lottery mechanisms. They differ the most in the way they select and extend the main chain. Algorand avoids forks by using byzantine consensus to agree on and finalize one proposed block at the time. Meanwhile Cardano resolves forks by always selecting the longest valid fork as the current main chain.

4.1 Algorand

The first lottery-based protocol we look at is Algorand. Our analysis of Algorand is based on the works of Chen and Micali [16] and Gilad et al. [18]. We split up the Algorand protocol into three parts. In the order that these sub-protocols are executed, they are the block proposal protocol, the reduction protocol and finally $BA\star$ a binary-byzantine-consensus protocol. For each block all three sub-protocols are run once in succession. In the Algorand protocol one such execution is known as a round. We refer to it as a slot instead, in order to unify the terms across the protocols. However Algorand does not group together several slots into an epoch like other protocols. Therefore the term epoch is not used in the description of Algorand.

Algorand does not feature a global clock but instead all steps in the protocol are bound to a time limit. The time limits for each step are the estimated time it takes for a party to compute and send the required data. Therefore a new step in the protocol begins once the previous step concluded.

Furthermore, during multiple steps of the protocol parties are randomly selected to fill key roles such as the leader of a slot or that of a committee member during the $BA\star$ protocol. In these instances Algorand introduces the $(r, \pi, v) = \textit{Sortition}(sk_i^{vrf}, \tau, m, w_i, W)$ method, for a party p_i to check privately whether or not it was selected for the role. The *Sortition* method serves as a wrapper that combines the result of multiple *VRF* evaluations into a single pair of random value r and proof π. In addition the value v is added to the result to describe how many of the *VRF* evaluations resulted in a win.

A single *VRF* evaluation inside the *Sortition* method is computed as $(r', \pi') = \textit{VRF}(m', sk_i^{vrf})$. While the same private key sk_i^{vrf} is used in all evaluations, the message m is modified to form a unique message m' for each attempt. In order to determine if the *VRF* evaluation is a win, the value r' is compared to the threshold τ. If r' is smaller than τ then the attempt is counted as a win. Note that rather then changing the threshold according to a party's stake in the system, Algorand gives parties one attempt for every token they own. Which means each party has exactly w_i attempts to win, where w_i is the amount of tokens the party p_i possesses. Because of this it is actually possible for a party to win multiple times during one selection, as represented by the output v of the *Sortition* method. Lastly note that the *Sortition* method combines the output pairs (r', π') of all w_i *VRF* evaluations into a single pair (r, π). Therefore only one proof is needed to verify the correctness of the *Sortition* result. This is reflected in the $\textit{SortVerify}(r, \pi, v, m, vk^{vrf})$ method, where only proof π is needed to verify all wins v that the party claims to have.

Finally, note that w_i used for the slot sl_j is actually the stake of party p_i from b blocks prior. So it is the stake party p_i possessed in slot sl_{j-b}. This delay is introduced to prevent an adversary from creating new key pairs that win the selection of the current slot sl_j. Since those keys did not exist in slot sl_{j-b} they are not allowed to participate in the selections of the current slot sl_j.

4.1.1 Chain-extension protocol

In this section we go over the three sub-protocols that together form the chain-extension protocol of Algorand: the block proposal, the reduction and the $BA\star$ sub-protocol. We give descriptions and pseudocode implementations for each of the three sub-protocols in the following sections.

4.1.1.1 Block proposal

The goal of the block-proposal sub-protocol is to select a leader for the current slot and for that leader to then propose a block for the chain extension. We have two subroutines in the block proposal protocol. The first is triggered when a party confirms the block for the previous slot. This routine is responsible to compute the randomness for the current slot and to determine a single leader that gets to extend the chain with a new block. The other subroutine is triggered when delivering a block message. The block messages contain a priority as well as the proposed block. The priority is used to select a single winner out of all possible leaders and therefore also determines which block is the main candidate for the following sub-protocols. We now describe the two subroutines in more detail and give a pseudocode representation of the block-proposal protocol in Algorithm 1.

On confirming the block of the previous slot A party starts with the block-proposal protocol as soon as the block of the previous slot is confirmed by the party itself. The first step is for the party to determine the randomness η_j that is used as the seed in the evaluations of the *Sortition* method during the slot sl_j. Normally the randomness η_j is provided in the block B_{j-1} of the previous slot. The η_j is computed by the previous leader by evaluating $(\eta_j, \pi) = \textit{VRF}(\eta_{j-1}||sl_j, sk^{vrf})$. The message used in the *VRF* is a concatenation of η_{j-1} the randomness for the previous slot and sl_j the slot identifier of the upcoming slot. Since the message does not include any data that can be chosen by the leader, it can not influence the randomness for the next slot. Of course the randomness proposed by the previous leader is only used,

if the verification *Verify*$(\eta_{j-1}\|sl_j, \eta_j, \pi, vk^{vrf})$ succeeds. Alternatively if the verification fails or if the previous block is empty, then the randomness η_j is computed as the hash $H(\eta_{j-1}||sl_j)$ instead.

After the randomness η_{j-1} has been computed a party p_i evaluates *Sortition*$(sk_i^{vrf}, \tau_{leader}, (\eta_j,$ LEADER$, sl_j), w_i, W) = (r, \pi, v)$ to check if it is a possible leader for this slot. As we mentioned previously the *Sortition* method signals with the return value v how many times the party p_i won out of the w_i chances it has. In the case of the leader selection $(\eta_j,$ LEADER$, sl_j)$ is the message for which the *VRF* is evaluated and τ_{leader} is the threshold for winning. If a party has at least one win, then it will propose a block. Even tough multiple parties may win a chance to propose a block only one party's block will be selected in the end.

The selection criteria for this is the priority of the proposed block, where the block with the highest priority wins. If a party wins multiple times it will still only propose one block, since only the block with the highest priority has a chance of being selected in the end. Therefore a party computes the priority for each of their wins as the hash $H(r||k)$ for $1 \leq k \leq v$ and then selecting the highest priority. The priority along with the output of the *Sortition* function and the identity of the party p_i are included in the block message. This way the other parties are able to verify the claimed priority. Please note that in practice the priority is split of from the block into its own message and broadcast ahead of time. This is done for efficiency, since this allows the parties to filter out low priority blocks much earlier. As a result there are fewer block messages broadcast in vain.

After computing the priority the party then computes a block $B_i = (sl_j, h, \mathcal{D}, (\eta_{j+1}, \pi), \sigma)$ to propose. A block contains the identifier for the current slot sl_j, the hash h of the previous block B_{j-1}, the list of transactions $\mathcal{D}$ included in the block, the randomness for the next slot η_{j+1} along with the proof π to verify it and finally the signature σ over $(sl_j, h, \mathcal{D}, (\eta_{j+1}, \pi))$. The party then stores the block along with its priority as the highest priority block it has seen so far. Finally the party then broadcasts the block using *gossip-broadcast*$([$BLOCK$, (p_i, priority, (r, \pi, v), B_i)])$ along with the block's priority and the proof data (r, π, v) to the other parties.

Regardless of whether or not the party did propose a block it then waits for block messages from other parties to arrive. The party waits for block messages for a duration of $\lambda_{step} + \lambda_{block}$. This is the estimated time it takes for a party to finish their block-proposal step and to send the block message to other parties. After the waiting period is over, the party then concludes the block-proposal sub-protocol and will start the reduction protocol with the highest priority block it has witnessed thus far.

On block message delivery When a new block arrives, a party first compares the priority of the block to the highest priority $priority_{max}$ it has witnessed so far. If the priority is smaller, the message is discarded, because the block has no chance to be the highest priority block. Otherwise the party verifies the priority by first evaluating *SortVerify*$(r, \pi, v, ($LEADER$, j), vk_l^{vrf})$ and afterwards recomputes the priority with the help of r and v. If the priority passes the verification, then it is recorded as the new highest priority witnessed so far and the message is broadcast to other parties. This is done even though the block itself might be invalid, since the protocol is strictly searching for the block with the highest priority, regardless of its validity.

That being said, a party then immediately verifies the validity of the block B. First it verifies that all transactions in the block data $\mathcal{D}$ are valid in regards to the local chain $\mathcal{B}$. Then the party checks the block signature σ over $(sl_j, h, \mathcal{D}, (\eta_{j+1}, \pi))$. If both verifications hold then the block B will be recorded as the highest priority block so far B_i. Should any of the verifications fail then B_i is set to an empty block instead. As a special case, should a party ever receive two different valid blocks produced by the same party, then it will reset B_i to the empty block. Since in that case the leader went against the protocol and is most likely adversarial.

Algorithm 1 Pseudocode: Algorand block proposal (party p_i).

state

$\mathcal{B}$: local blockchain
h: latest hash of local chain = $H(head(\mathcal{B}))$
sl_j: current slot of chain-extension protocol
$priority_{max}$: the highest priority of a selected leader witnessed by p_i
p_{max}: party with the highest priority of a selected leader witnessed by p_i
B_i :block that p_i enters into $BA\star$
$unordered$: set of transactions to be included in the block

vk_i, sk_i: signing key pair for the blocks
vk_i^{vrf}, sk_i^{vrf}: key pair for the VRF proofs
w_i: amount of stake of party p_i in regards to the current slot sl_j

$\lambda_{step} + \lambda_{block}$: estimated time for a party to finish the protocol step and their block message to arrive
η_j: the randomness for the slot sl_j
τ_{leader}: the threshold to be selected as leader

upon p_i confirms block B_{j-1} of the previous slot sl_{j-1} **do**
$(\eta_j, \pi) \leftarrow$ select (η_j, π) from the previous block B_{j-1}
if not $Verify(\eta_{j-1}||sl_j, \eta_j, \pi, vk^{vrf})$ **then**
$\eta_j \leftarrow H(\eta_{j-1}||sl_j)$
$(r, \pi, v) \leftarrow Sortition(sk_i^{vrf}, \tau_{leader}, (\eta_j, \text{LEADER}, sl_j), w_i)$

$priority_{max} \leftarrow 0$
$B_i \leftarrow empty\_block$
if $v > 0$ **then**
$priority \leftarrow max(H(r||k))$ for $1 \leq k \leq v$

$D \leftarrow$ select maximal valid set of transactions from $unordered$
$(\eta_{j+1}, \pi) \leftarrow VRF(\eta_j||sl_j, sk_i^{vrf})$
$B_i \leftarrow (sl_j, h, \mathcal{D}, (\eta_{j+1}, \pi), \sigma)$
$priority_{max} \leftarrow priority$
$p_{max} \leftarrow p_i$
$gossip\text{-}broadcast([\text{PRIORITY}, (p_i, priority, (r, \pi, v), B_i)])$

wait$(\lambda_{step} + \lambda_{block})$ //wait while gathering block messages
after timeout begin reduction step with block B_i (Alg. 2)

upon $gossip\text{-}deliver([\text{BLOCK}, (p_l, priority, (r, \pi, v), B)])$**do**
if $priority \geq priority_{max}$ **then**
verify $SortVerify(r, \pi, v, (\text{LEADER}, j), vk_l^{vrf})$
verify $priority = max(H(r||k))$ for $1 \leq k \leq v$
if priority passes all verifications **then**
$priority_{max} \leftarrow priority$
$p_{max} \leftarrow p_l$
$gossip\text{-}broadcast([\text{BLOCK}, (p_l, priority, (r, \pi, v), B)])$

verify the block signature σ over with the public key vk_l of the current leader p_l
verify that all transactions $\mathcal{D}$ in block B are valid in regards to the local chain $\mathcal{B}$
if the block B passes all verifications **then**
$B_i \leftarrow B$
if at any point two different but valid blocks by the same party p_l are delivered **then**
$B_i \leftarrow empty\_block$
else:
$B_i \leftarrow empty\_block$

4.1.1.2 Reduction

The reduction protocol consists of two subroutines. The first is the main routine and begins immediately after the block proposal has concluded. The second is triggered by incoming vote messages. In this section we give a description of the reduction protocol and present a pseudocode representation of the protocol in Algorithm 2.

The purpose of the reduction protocol is to prepare the parties for the $BA\star$ protocol. Since the $BA\star$ is a binary-byzantine-agreement protocol we must reduce the blocks that enter into $BA\star$ to two choices. Those two choices are the block with the highest priority from the block proposal and the empty block. Not all parties may have witnessed the highest priority block during the block proposal. Therefore the goal of the reduction protocol is to convince all parties to change their choice of block to the block that a majority of parties has witnessed. If a party is not convinced to support this block, then it will enter the $BA\star$ protocol with the empty block.

In the reduction protocol as well as in the $BA\star$ protocol, parties vote for the block that they support at the moment. We represent this in the pseudocode with the method *Vote*$((sk_i^{vrf}, \tau, m, w_i), h_i)$. In the *Vote* method a party first evaluates the *Sortition*$(sk_i^{vrf}, \tau_{step}, m, w_i)$ to compute if they are eligible to vote in this round. This is the case if the party had at least one win during the *Sortition*. In other words when $v > 0$. If they are allowed to vote then they will *gossip-broadcast*$([\text{VOTE}, (p_i, h_i, (r, \pi, v))])$, where h_i is the hash of the block the party votes for. Note that the number of wins v acts as the weight of the vote, therefore parties with more wins have more influence in the voting process.

On conclusion of block proposal When block proposal concludes the party p_i has decided on a block B_i. This block is either the highest priority block the party has seen or the empty block. The reduction protocol consists of two rounds of voting. Instead of including the whole blocks in the vote messages, only the hashes of the blocks are used in order to be more efficient. Therefore the first step in the protocol is to compute the hash h_i of the block B_i. Additionally each party prepares the hash of the empty block.

The parties then conduct two rounds of voting. As mentioned above for a vote the parties first determine if they are allowed to vote for this round. If applicable they then broadcast a vote message for the block hash h_i they support. The message m that is used to determine the eligibility of a party is specific to that round of voting. This means that for each round there is a different set of parties that are part of the voting committee. The two rounds of voting are almost identical. First a party votes for their block hash h_i if applicable. Regardless of whether the party did vote, it then waits for vote messages to arrive. The only notable difference between the two rounds is that a party waits longer during the first round. Namely $\lambda_{block} + \lambda_{step}$ instead of just λ_{step} as in the second round. This is because for the first round we want to give the other parties enough time to finish their block proposal.

This waiting period has two possible outcomes: either a block hash gathers enough votes to win or the waiting period ends in a timeout without a winner. A block hash needs more than $T_{step} * \tau_{step}$ votes to be accepted. Where $T_{step} * \tau_{step}$ amounts to $2/3$ of the estimated total number of votes in the voting round. If there is a block hash h_w that passed the threshold then party p_i replaces the block hash they support h_i with h_w. Otherwise if the vote ends in a timeout then p_i changes h_i to the empty block hash h_{empty}.

On voting message delivery This subroutine tallies incoming votes for block hashes. The tallying stops immediately if one block hashes gets enough votes to pass the threshold $T_{vote} * \tau_{vote}$. Incoming votes are first validated by verifying the *VRF* proof. If the vote is valid then the votes v are added to the total votes of the corresponding block hash. Finally we check if the block hash received enough votes to pass the threshold. If that is the case, then the winning block hash is stored in the local variable h_w.

Algorithm 2 Pseudocode: Algorand reduction protocol (party p_i).

state

B_i: the block that p_i is entering into $BA\star$ with
h_i: the hash of the block that p_i is entering into $BA\star$ with

h_w: is used to store the winning block hash of a vote

vk_i^{vrf}, sk_i^{vrf}: key pair for the VRF proofs
w_i: amount of stake of party p_i in regards to the current slot sl_j

$\lambda_{step} + \lambda_{block}$: estimated time for a party to finish the protocol step and their block message to arrive
η_j: the randomness for the slot sl_j
τ_{step}: the threshold to be selected for a step in $BA\star$
T_{step}: the fraction of votes needed in a step of $BA\star$

upon p_i concludes block proposal step **do**:
$h_i \leftarrow H(B_i)$
$h_{empty} \leftarrow H(empty\_block)$

Vote$((sk_i^{vrf}, \tau_{step}, (\eta_j, \text{REDUCTION\_ONE}, sl_j), w_i), h_i)$
$h_w \leftarrow$ NULL
wait$(\lambda_{block} + \lambda_{step})$ // wait while gathering vote messages
if a winning block hash h_w was set **then**
    $h_i \leftarrow h_w$
else: // no winner after $\lambda_{block} + \lambda_{step}$
    $h_i \leftarrow h_{empty}$

Vote$((sk_i^{vrf}, \tau_{step}, (\eta_j, \text{REDUCTION\_TWO}, sl_j), w_i), h_i)$
$h_w \leftarrow$ NULL
wait(λ_{step}) // wait while gathering vote messages
if a winning block hash h_w was set **then**
    $h_i \leftarrow h_w$
else: // no winner after λ_{step}
    $h_i \leftarrow h_{empty}$
begin $BA\star$ step with block hash h_i (Alg. 3)

upon *gossip-deliver*$([\text{VOTE}, (p_c, h_b, r, \pi, v)])$**do**
if *SortVerify*$(r, \pi, v, m, vk_c^{vrf})$ **then**
    increase $votes$ for h_b by v
    if $votes$ for $h_b \geq T_{vote} * \tau_{vote}$ **then**
        $h_w \leftarrow h_b$

4.1.1.3 Binary byzantine agreement $BA\star$

Like other consensus protocols the goal of $BA\star$ is for all honest parties to agree on the block that is added to the chain. It is called a binary protocol because the parties initial values for the block is one of two options. Either the highest priority block from the block proposal or the empty block. In the reduction protocol, that comes before $BA\star$, the parties previously narrowed down their vote to one of those two options. $BA\star$ is a special case of a byzantine agreement, since in each step of the protocol the active participants change. Like in the leader selection, the committee membership for the $BA\star$ protocol is won via the *Sortition* method. But since a consensus protocol usually requires a participant to send multiple messages, the committee members would be known after the first message they send. This makes them a target for corruption by the adversary. Therefore $BA\star$ randomly re-selects its committee members after each step in the protocol to prevent them from becoming targets.

Like the reduction protocol we use the *Vote* method as a short hand for a party to first check whether it is a committee member for the current step and then vote for the block hash they support if they were selected. Additionally these votes are then tallied the same way. Therefore a result of the vote is either a block hash that managed to accumulate enough votes during the waiting period or else a timeout. This is reflected in the subroutine that is triggered when a new vote message is delivered. Note this subroutine is identical to the one in the reduction protocol, therefore we wont go over it again here. Please refer to Algorithm 3 for the pseudocode.

On conclusion of reduction protocol In this subroutine the parties try to find consensus over which block to add to the chain. As a result of the reduction each party has a block hash h_i that they intend to support in the $BA\star$ protocol. As a first step each party assigns this block hash to h_r, which is the block hash they vote for in each step. The value of h_r may change from step to step based on the results, while h_i will remain the same during the whole protocol. The parties then try to reach consensus over a loop of three steps. A party stops the loop when it either reaches consensus or when it reaches a maximum number of steps s_{max}. The three steps all follow a similar structure. They consist of a vote, where all eligible parties vote for the block hash h_r they currently support. Then each party tallies the vote messages they delivered. If one option received enough votes the party will then check if the $BA\star$ protocol can be concluded. If instead there was a timeout then the party changes their voting strategy for the next step.

In the first step a party will only conclude $BA\star$ if the party witnessed enough votes for a non-empty block hash. If this happens in the very first step, then the party votes for this block hash in the finalization round. If the winning block hash also gathers enough votes in the finalization round, then it is considered final. Otherwise it is considered tentative. A tentative block is only considered final once a final block is added to the chain after it. Additionally a party that concluded the $BA\star$ protocol will continue to vote for the winning block for three more steps. This is done to prevent a small group of parties from getting stuck in the $BA\star$ protocol, because they have insufficient votes to get past the threshold by themselves. Finally if no option reached enough votes during the first step, then the party switches its vote h_r to h_i.

In the second step the party concludes $BA\star$ if it witnesses enough votes for the empty block hash. Like in step one it will continue to vote for the empty block hash for three additional steps before moving on to the next slot. If the party did not witness enough votes for either option then it will switch its vote h_r to the empty block hash h_{empty}.

Finally in the third step it is not possible for a party to conclude the $BA\star$ protocol. Here we only care about if there was a timeout or not. If there was a timeout, then the party will switch its vote based on the result of the *CommonCoin* method. The result of this method is a coin flip of a slightly common coin. The party will switch h_r to h_i if the bit is zero or to the h_{empty} if the bit is one. To compute the result of the coin flip a party takes all vote messages that it delivered during this step and computes the lowest priority across all messages. Then it takes the least significant bit of the lowest priority as the result of the coin flip. By basing the result of the coin on the lowest priority that was witnessed by a party, there is a high

chance that a majority of the parties witnessed the same message. As a result they will switch their vote to the same choice. This slightly common coin flip helps the parties to get unstuck if they are split in a way where no option can reach the necessary votes to win. After the third step the party repeats the loop until it finds consensus or the maximum number of steps is reached.

After the $BA\star$ protocol concludes, then all parties will deliver the winning block to their local chain. Furthermore each party will evaluate the finalization vote. As noted above, a party will only vote in the finalization vote if it reached consensus on a non-empty block in the very first step. If there are more than $2/3$ of votes for the finalization then the block that is added to the chain is considered final. Otherwise it is considered tentative. A tentative block becomes final once a final block is added to the chain after it. Note that in addition to the block and its status, the parties will store a kind of certificate for the new block. This certificate consists of the signed collection of votes that caused the party to conclude the $BA\star$ protocol. These certificates are used to convince other parties of the correctness of the chain. Therefore they enable parties that newly join or rejoin the chain to verify the chains that are proposed to them.

4.1.2 Protocol attributes

Algorand achieves its random selection by using VRFs as the basis of its selection mechanism. A important feature of VRFs is that they can be evaluated privately and the result can be verified at a later stage. This has the distinct advantage that a party can keep its role secret until it is ready to share its results. For example a party that was selected as a leader will only share that fact with everyone else once it has proposed its block. This means that an adversary will only learn of a party's role after it has already acted. This means that Algorand is capable of dealing with an adversary that is able to corrupt a party instantly.

Additionally, as we saw above, the Algorand protocol ensures that there is exactly one block added to the chain for each slot. As a result there are no forks in Algorand. This attribute is achieved by including a byzantine-consensus protocol in the chain extension. This ensures that the parties reach consensus on the block that extends the chain. Because a byzantine-consensus protocol is included in the chain extension, the minimum amount of honest stake needed is $2/3$. This means that more than $2/3$ of the stake must be controlled by honest parties.

Finally, by having the parties store certificates for the blocks they found consensus on, new parties are able to verify the chain they received upon joining. This is done by gathering and verifying certificates for each block from other parties in the blockchain network. This way new parties or rejoining parties can be protected from receiving false chains form an adversary, since the adversary is unable to forge the necessary certificates for the false blocks that they inserted.

Algorithm 3 Pseudocode: Algorand $BA\star$ (party p_i).

state

h_i: the hash of the block that p_i is entering into $BA\star$ with
h_r: the hash of the block that p_i is voting for in the next step
h_w: is used to store the winning block hash of a vote
s: current step in $BA\star$ initialized as 1

vk_i^{vrf}, sk_i^{vrf}: key pair for the VRF proofs
w_i: amount of stake of party p_i in regards to the current slot sl_j

λ_{step}: timeout time for a party to finish a protocol step
η_j: the randomness for the slot sl_j
τ_{step}: the threshold to be selected for a step in $BA\star$
τ_{final}: the threshold to be selected for a finalization step in $BA\star$
T_{step}: the fraction of votes needed in a step of $BA\star$
T_{final}: the fraction of votes needed in the finalization step of $BA\star$

upon p_i concludes reduction step **do**:
$h_r \leftarrow h_i$
until $s > s_{max}$ **do**:

Vote$((sk_i^{vrf}, \tau_{step}, (\eta_j, \text{COMMITTEE}, sl_j, s), w_i), h_r)$
$h_w \leftarrow$ NULL
wait(λ_{step}) // wait while gathering vote messages
if a winning block hash h_w was set **then**
if $h_r \neq h_{empty}$ **then**
p_i votes for h_r for the next 3 steps if it is part of the commitee for the step
if $s = 1$ **then**
Vote$((sk_i^{vrf}, \tau_{final}, (\eta_j, \text{FINAL}, sl_j), w_i), h_r)$
conclude $BA\star$ with h_r as the result
else: // no winner after λ_{step}
$h_r \leftarrow h_i$
$s \leftarrow s + 1$

Vote$((sk_i^{vrf}, \tau_{step}, (\eta_j, \text{COMMITTEE}, sl_j, s), w_i), h_r)$
$h_w \leftarrow$ NULL
wait(λ_{step}) // wait while gathering vote messages
if a winning block hash h_w was set **then**
if $h_r = h_{empty}$ **then**
p_i votes for h_r for the next 3 steps if it is part of the committee for the step
conclude $BA\star$ with h_r as the result
else: // no winner after λ_{step}
$h_r \leftarrow h_{empty}$
$s \leftarrow s + 1$

Vote$((sk_i^{vrf}, \tau_{step}, (\eta_j, \text{COMMITTEE}, sl_j, s), w_i), h_r)$
$h_w \leftarrow$ NULL
wait(λ_{step}) // wait while gathering vote messages
if no a winning block hash h_w was found **then**
$b \leftarrow$ *CommonCoin*()
if $b = 0$ **then**
$h_r \leftarrow h_i$
else:
$h_r \leftarrow h_{empty}$
$s \leftarrow s + 1$

upon *gossip-deliver*$([\text{VOTE}, (p_c, h_b, r, \pi, v)])$**do**
if *SortVerify*$(r, \pi, v, (\text{COMMITTEE}, j, s), vk_c^{vrf})$ **then**
increase $votes$ for h_b by v
if $votes$ for $h_b \geq T_{vote} * \tau_{vote}$ **then**
$h_w \leftarrow h_b$
$h_r \leftarrow h_b$

4.2 Cardano

The second lottery-based protocol we are looking at is Cardano. Cardano uses Ouroboros for its proof-of-stake consensus protocol. While there are many extensions to Ouroboros, we will focus on Ouroboros Classic [19] and Praos [17] and will give only a high-level overview for the other extensions. We begin with a detailed look at Ouroboros Classic and then go over the changes that were made for Praos.

4.2.1 Ouroboros Classic

Ouroboros Classic is the first iteration of the Ouroboros protocol. It can be viewed as consisting of three sub-protocols. Namely the protocols for chain extension, generating randomness and preparing the input for the blocks. A protocol run is divided into epochs which are further divided into slots. The participants in the protocol are called stakeholders or players in the work of Kiayias et al. [19], but we will use the term party instead. Furthermore we will use the term hash instead of state when we discuss the hashes that make up the hash chain.

At the beginning of each epoch e_j a leader is randomly selected for each slot sl in the epoch. The leaders are selected from the set of all parties by using the deterministic function $F(\mathbb{S}_j, \rho^j, sl)$ where $\mathbb{S}_j$ and ρ^j are the stake distribution among the parties and the randomness during the epoch e_j. Finally sl is the slot for which the leader is selected. A party's relative stake directly corresponds to the probability that it is selected as leader for a slot. The selections of $F(\mathbb{S}_j, \rho^j, sl)$ for each slot are independent from each other.

A party that was selected as leader has two main tasks during the epoch. The first is to extend the chain with a new block during the slot where it is the leader. The second is to participate in a multi-party coin-toss protocol in order to compute the randomness ρ^{j+1} for the next epoch. Both protocols run parallel to each other. The second role that is selected for each slot is that of the input endorser. As with the leader role one input endorser is chosen per slot out of all parties. Again the relative stake of a party directly corresponds to the probability that it is selected for a slot. Note that there may be overlaps between the group of leaders and the input endorsers. An input endorsers task is to collect, validate and prepare the data to be used in the next block.

4.2.1.1 Chain-extension protocol

The chain-extension sub-protocol consists of two subroutines. The first is responsible for collecting new blockchain candidates broadcast by other parties and is triggered when a new blockchain candidate is delivered. The second is executed by each party for each slot and consist of updating the local blockchain and the randomness if applicable. In addition the leader of the current slot will extend the blockchain by adding a new block and broadcasting the resulting chain. We now go over the two subroutines in more detail and give a pseudocode representation of the chain-extension sub-protocol in Algorithm 4

On blockchain candidate delivery When a party delivers a new blockchain candidate, the party will first validate it. This is done by validating each block in the blockchain candidate. A block $B_i = (h_{i-1}, \mathcal{D}_i, sl_l, \sigma_{sk_l})$ consists of the hash of the previous block $h_i = H(B_{i-1})$, the data $\mathcal{D}_i$ that is contained in the block B_i, the index of the slot during which the block was computed sl and lastly the signature σ_{sk_l} over $(h_{i-1}, \mathcal{D}_i, sl)$ by the leader of the slot sl. A block B_i is valid if signature σ_{sk_l} passes verification and the signature key pair for σ belongs to the selected leader of slot sl. This can be verified by checking if $vk_l = F(\mathbb{S}_j, \rho^j, sl)$, where $\mathbb{S}_j$ and ρ^j are the the stake distribution and the randomness for the epoch that contains slot sl. If every block in the candidate blockchain passes this validation then the blockchain is considered valid and is added to the set $\mathbb{C}$ of valid blockchain candidates.

On the start of a new slot Each party becomes active at the beginning of each slot. The beginning of a slot is signaled by a global clock. There are three tasks in total, two of which are executed by all parties and one only by the leader of the slot. First each party checks if a new epoch e_j for $j \geq 2$ has started. If that is the case then it computes the the stake distribution $\mathbb{S}_j$ for the new epoch. This is done by considering all transactions in the local blockchain $\mathcal{B}$ up until the most recent stable block. A block is considered stable when it is k blocks deep in the blockchain. Since this data is not available for the first epoch the initial stake distribution $\mathbb{S}_1$ is given by the genesis block.

The second task preformed by all parties is to update the local chain $\mathcal{B}$. As mentioned above, all valid candidate chains are collected in the set $\mathbb{C}$. Each party runs the method $maxvalid_k(\mathbb{C}, \mathcal{B})$. This function returns the longest valid chain $\mathcal{B}'$ out of the set $\mathbb{C} \cup \mathcal{B}$. $\mathcal{B}'$ will replace the current local chain $\mathcal{B}$. To be considered valid, a chain can not fork from the local chain $\mathcal{B}$ for more then k blocks. This ensures that there are no radical changes in the local blockchain and blocks that are considered stable are not invalidated. If the method $maxvalid_k(\mathbb{C}, \mathcal{B})$ finds multiple chains with maximal length, then $\mathcal{B}$ has priority if it is an element of that set. Otherwise $\mathcal{B}'$ is selected arbitrarily from the set of longest chains. Note that $maxvalid_k$ does require that a party is never offline for more then k blocks. Because at that point the parties local chain $\mathcal{B}$ will have fallen behind for more than k blocks. This means that all candidate chains are no longer valid, since all of them are now filtered out by $maxvalid_k$. Therefore creating a situation where a long-absent party is no longer able to rejoin the protocol.

Finally each party checks if it was selected as the leader for the slot by evaluating the function $F(\mathbb{S}_j, \rho^j, sl)$ for the current slot sl. The function will return the public signature key of the party that was selected. By comparing the result of $F(\mathbb{S}_j, \rho^j, sl)$ to its own public signature key vk_i a party p_i can determine if it is the leader. If that is the case then the party will compute the new block $B = (h, \mathcal{D}, sl, \sigma)$ where $h = H(head(\mathcal{B}))$ is the hash of the latest block in the local blockchain. The variable $\mathcal{D}$ represents the data that is included in the block, we will discuss the construction of $\mathcal{D}$ in the input endorsement sub-protocol section. The slot that the block belongs to is represented by the variable sl. And σ is the signature over $(st, \mathcal{D}, sl)$ with the secret signature key sk_i of party p_i. The new block B is appended to the local chain $\mathcal{B}$. Finally the new chain is then broadcast to the other parties.

4.2.1.2 Creating randomness

The leader-selection function in the Ouroboros Classic protocol requires randomness in order to work properly. This randomness needs to be renewed for each epoch. This is where the sub-protocol for creating this randomness comes into play. This sub-protocol is run in parallel to the chain extension and involves all the leaders of an epoch e_j. The goal is to construct the randomness ρ^{j+1} for the following epoch e_{j+1} through multi-party computation. An epoch consists of R slots, therefore this protocol involves up to R parties. Note that it is possible for the same party to be leader in multiple slots during one epoch, therefore the R parties are not necessarily distinct. Furthermore as a prerequisite all parties that participate in this sub-protocol need to have published their public encryption key y_i (of encryption key pair(y_i, s_i)) on the blockchain. Otherwise they are not included in the multi-party computation.

The idea is to use a R-party coin-toss protocol to construct the randomness for each epoch. However using the coin-toss protocol by itself can be problematic since an adversarial party can choose to abort the protocol. When this happens the honest parties will never learn the result of the coin toss. This can be done even after every other party revealed its results. Because of this the adversarial party can learn the overall result and decide afterwards to abort the protocol, if the result is not in its favor. This is why the coin-toss protocol needs to be extended with a publicly-verifiable-secret-sharing scheme (PVSS) in order to prevent an adversary from aborting the protocol. We will now describe this extended scheme in more detail in the following paragraphs and give a pseudocode implementation in Algorithm 5.

Algorithm 4 Pseudocode: Ouroboros Classic chain extension (party p_i).

state
- $\mathcal{B}$: local blockchain
- h: latest hash of local chain = $H(head(\mathcal{B}))$
- $\mathbb{C}$: set of valid chain candidates
- vk_i, sk_i: signing key pair for the blocks
- $\mathbb{D}$: list of the last d endorsed inputs to be included in the block
- ρ^j: the randomness for the epoch e_j

upon *deliver*$([\text{CHAIN}, C])$ **do**
- **for** every block $B = (h, d, sl, \sigma) \in C$ **do**
  - verify the signature σ over (h, d, sl) with the public key vk_j of the leader of the slot sl
- **if** C passed all block verifications **then**
  - $\mathbb{C} \leftarrow \mathbb{C} \cup \{C\}$

upon clock indicating the start of a new slot sl **do**
- **if** a new epoch e_j has started for $j \geq 2$ **then**
  - compute the stake distribution $\mathbb{S}_j$ for the next epoch
- $\mathcal{B} \leftarrow maxvalid_k(\mathcal{B}, \mathbb{C})$
- **if** p_i is the leader of slot sl **then**
  - $\mathcal{D} \leftarrow$ select all endorsed inputs from $\mathbb{D}$ that are not included in a block of $\mathcal{B}$
  - $B \leftarrow (h, \mathcal{D}, sl, \sigma)$ where $\sigma = Sign_{sk_i}(h, \mathcal{D}, sl)$
  - $\mathcal{B} \leftarrow \mathcal{B} \| B$
  - $h \leftarrow H(head(\mathcal{B}))$
  - *broadcast*$([\text{CHAIN}, \mathcal{B}])$

Commitment Phase In a first step we adjust R if necessary. Since a party's public encryption key must be published in order to participate in the PVSS, we adjust the number of participants R accordingly. Each party starts by executing the method *Deal*$(R, R/2, y_1, ..., y_R)$. This method will select a secret σ, in this case σ is the contribution of the party for the coin-toss scheme. This secret is then split into R shares, one for each party. The $R/2$ indicates that more then half of the shares are needed to reconstruct the secret σ. The shares which are denoted by $(\sigma_1, ..., \sigma_R)$ are then encrypted with the public key of the receiver. For example $\beta_i = Enc(\sigma_i, y_i)$ denotes the encrypted share for party p_i. Note that in addition β_i contains a non-interactive-zero-knowledge proof that allows the other parties to verify the validity of the encryption. Each party then broadcasts the encrypted shares $(\beta_i, ..., \beta_R)$ so that they are included in the blockchain. The *commitment* phase starts at slot $R - 2k - 4\ell$ of each epoch and lasts for $2k + 3\ell$ slots.

Reveal Phase Immediately after the *commitment* phase begins the *reveal* phase at slot $R - \ell$ and last for ℓ slots until the end of the epoch. In this phase the parties will run the method *Decrypt*(s_i, β_{li}) where β_{li} is the encrypted share for party p_i for the secret of party p_l. However a party will only decrypt a share if two conditions are met. First all R shares $(\beta_{l1}, ..., \beta_{lR})$ must be present in the blockchain. Secondly every share needs to be valid, which can be verified by evaluating the associated proof. Only if both conditions are met is a party allowed to decrypt and broadcast its share of the corresponding secret.

Computing the randomness At the end of an epoch the shared secrets are reconstructed in order to compute the randomness for the next epoch. First each party will collect all shares for each secret from the blockchain. As mentioned, a secret can be reconstructed if more then $R/2$ shares are present. If that is the case, we can compute the randomness that a party p_l contributes with $\rho_l^j = H(Rec(\sigma_{l1}, ..., \sigma_{lR}))$. Here the ρ_l^j denotes the randomness contributed by party p_l for the epoch e_j. If there are less then $R/2$ shares present, then the ρ_l^j is set to zero. Finally the randomness for the next epoch can be computed with $\rho^j = \oplus_{l=1}^{R} \rho_l^j$.

4.2.1.3 Input endorsement

In Ouroboros it is not the leaders that decide what data is included in a block, instead it is prepared by the input endorsers. We will denote those parties as p_e. Exactly like with the leaders a single input endorser is selected randomly for each slot. The probability of a party being selected as an input endorser is again based on its relative stake in the system. An input endorser's purpose is to collect transactions and sub-protocol data to be included in the next block. The input endorser is responsible for verifying the collected data and for resolving possible conflicts between transactions. The data is collected in an endorsed input set D which is then broadcast.

Each party that delivers such an endorsed input set D will store it in a queue $\mathbb{D}$ which contains the last d endorsed inputs, where d is a parameter of the sub-protocol. When a party is selected to be leader, it includes all endorsed input sets $D \in \mathbb{D}$ in its next block . If a input set D is already part of a previous block, it will not be included again. This means that a block may contain 0 to d endorsed input sets D. Note that there may be transactions that are included in the blockchain more then once. This is resolved by a strictly increasing serial number that each party includes in its own transactions. Therefore duplicate transactions can be identified by their serial number. The serial numbers are also used in the case of two conflicting transactions. Here the priority is given to the transaction with the smaller serial number. The pseudocode for the input endorsement is included in Algorithm 6.

4.2.2 Ouroboros Praos

Ouroboros Praos [17] is an evolution of Ouroboros Classic that raises the security by allowing for a more powerful adversary. It does this by changing the leader selection sub-protocol. Since in Ouroboros Classic

Algorithm 5 Pseudocode: Ouroboros Classic creating randomness sub protocol (party p_i).

state
$\mathcal{B}$: local blockchain
ρ^j: the randomness for the epoch e_j
y_i, s_i: encryption key pair for VPSS

upon the start of the *commitment* phase **do** // phase lasts form slot $R - 2k - 4\ell$ to slot $R - \ell - 1$
$(\beta_1, ..., \beta_R) \leftarrow$ *Deal*$(R, R/2, y_1, ..., y_R)$
broadcast$([\text{COMMIT}, (\beta_1, ..., \beta_R)])$

upon the start of the *reveal* phase **do** // phase lasts form slot $R - \ell$ to the end of the epoch
for $1 \leq l \leq R$ **do**
$(\beta_{l1}, ..., \beta_{lR})$ = query the local chain $\mathcal{B}$ for all encrypted shares provided by party p_l
if all shares $(\beta_{l1}, ..., \beta_{lR})$ exist and every share passes verification **then**
$\sigma_{li} \leftarrow$ *Decrypt*(s_i, β_{li})
broadcast$([\text{REVEAL}, \sigma_{li}])$

upon end of epoch e_{j-1} **do**
for $1 \leq l \leq R$ **do**
$(\sigma_{l1}, ..., \sigma_{lR})$ = query the local chain $\mathcal{B}$ for all decrypted shares provided by party p_l
if more then $R/2$ decrypted shares are present **do**
$\rho_l^j \leftarrow H(Rec(\sigma_{l1}, ..., \sigma_{lR}))$
else
$\rho_l^j \leftarrow 0$
$\rho^j \leftarrow \oplus_{l=1}^{R} \rho_l^j$

Algorithm 6 Pseudocode: Ouroboros Classic input endorsement sub protocol (input endorser party p_e).

state
$\mathbb{D}$: list of the last d endorsed inputs to be included in the block
ρ^j: the randomness for the epoch e_j
$unordered$: set of input data to be endorsed

upon *deliver*$([\text{DATA}, D])$ **do**
if $|\mathbb{D}| \geq d$ **then** remove the oldest entry from $\mathbb{D}$
$\mathbb{D} \leftarrow \mathbb{D} \| D$

upon clock indicating the start of a new slot sl **do**
if p_i is the input endorser of slot sl **then**
broadcast$([\text{DATA}, unordered])$

upon *deliver*$([\text{OP}, x])$ **do**
$unordered \leftarrow unordered \cup \{x\}$

upon *deliver*$([\text{COMMIT}, c])$ **do**
$unordered \leftarrow unordered \cup \{c\}$

upon *deliver*$([\text{REVEAL}, r])$ **do**
$unordered \leftarrow unordered \cup \{r\}$

all leaders of an epoch are known at the beginning of the epoch, they are in risk of being targeted by an adversary. That is why the adversary in Ouroboros Classic may not corrupt a party instantly but only after a delay. Otherwise the adversary would be able to systematically corrupt selected leaders before their slot begins to take control of the system. To remove this restriction from the adversary, the leader-selection process was changed, so that each party can evaluate privately whether it is the leader or not. If and only if a party is a leader of the slot, it will be able to successfully prove it is the leader and therefore compute a valid new block. Another change is the introduction of key-evolving signatures that provide forward security in order to protect against adaptive corruption. We will now discuss the changes made to the Ouroboros Classic protocol in more detail.

4.2.2.1 Verifiable random function for leader selection

In Praos the deterministic leader-selection function $F(\mathbb{S}_j, \rho^j, sl)$ form Ouroboros Classic is replaced with a VRF-based selection mechanism. The main difference between the two approaches is that in Classic every party that evaluates F will receive the same result for each slot. This results in every slot having exactly one leader. This is not the case for Praos. Every party will evaluate the VRF privately and will then find out whether or not they are selected as leader for the slot. However since the results of those evaluations are independent form each other, there are slots that have no leader and slots that have multiple leaders. Note that because the parties evaluate the VRF privately, the other parties only find out which parties are leaders after they already broadcast their new blocks. This gives the adversary no time between learning the set of leaders of a slot and them computing their blocks to corrupt them.

In Praos all parties evaluate the function $\mathit{VRF}(m, sk_i^{vrf}) = (r, \pi)$ privately during each leader selection. To determine if a party p_i is a leader for the slot it evaluates $r < T_i^j$. If the result is true then p_i is a leader of the slot. The threshold T_i^j depends on the relative stake of party p_i during epoch e_j. The higher the party's relative stake is, the higher is the threshold T_i^j. Therefore raising the party's chances to become a leader. As for the message m used in the VRF each party uses $m = \eta_j \| sl \| \text{TEST}$ where η_j is the nonce for epoch e_j and sl the slot for which *VRF* is evaluated. Lastly TEST is simply the string TEST. This string is used because the VRF is reused in the computation of the nonce η_j. We will discuss computation of η_j later in this section. In short, a party computes $(r, \pi) = \mathit{VRF}(m, sk_i^{vrf})$ and checks if $r < T_i^j$. If that is the case then the party is a leader of the slot and may compute a new block. The party must include $B_\pi = (p_i, r, \pi)$ in the new block so that the other parties can verify that p_i is in fact a leader of the slot. They learn the party's identity through B_π and can therefore verify the claim by evaluating $\mathit{Verify}(\eta_j \| sl \| \text{TEST}, r, \pi, vk^{vrf})$ and $r < T_i^j$. If both statements are true the party p_i is indeed a leader of the slot.

When a party p_i evaluates the VRF for the leader selection, it will also evaluate $\rho = (\rho_r, \rho_\pi) = \mathit{VRF}(\eta_j \| sl \| \text{NONCE}, sk^{vrf})$. The randomness from this evaluation is used to compute the randomness η_{j+1} for the upcoming epoch. For this purpose ρ is included in the block. Therefore a block in the Praos protocol is defined as $B = (h, \mathcal{D}, sl, B_\pi, \rho, \sigma)$. When a new epoch e_j begins every party can compute the randomness for the epoch as follows: $\eta_j = H(\eta_{j-1} \| j \| v)$. Here η_{j-1} is the randomness of the previous epoch and j the index of the current epoch. The value v is a concatenation of the values ρ that are included in the stable blocks of the previous epoch e_{j-1}.

Since Ouroboros Praos has additional variables included in the blocks compared to Classic, it is important to note that the validation process has some additional steps. As in Classic the signature σ over (h, d, sl, B_π, ρ) needs to be verified. Secondly the proof π in B_π needs to be verified successfully and $r < T_i^j$ must be true. And lastly the proof ρ_π in ρ must pass the verification as well.

4.2.2.2 Key-evolving signatures

The Praos iteration of Ouroboros switches to a key-evolving signature scheme for the block signatures. In this scheme the signature key is updated after each signature, while the old signature key is deleted. While all signatures can still be linked to the same key, it is not feasible to reconstruct previous keys from the current one. In Praos [17] these signatures are used to achieve forward security. This means that when an adversary corrupts a party it can create valid signatures with the current key and all future keys, but is unable to create valid signatures for past keys. In Praos the parties use a new key for every slot. This means that even if the party did not sign a block during the slot, the key for this slot will be discarded.

This means that there is a slight change to the *Sign* method. We add the parameter sl to the method resulting in $\mathit{Sign}_{sk_i}((h, \mathcal{D}, sl, B_\pi, \rho), sl)$. With this added parameter we signal the method which iteration of the signature key to use. If necessary the method will internally update the key until it arrives at the key for the current slot. After the signature is created the key is updated one last time. All keys except the newest one are erased in the process. Similarly the verification function $\mathit{Verify}_{vk_i}((h, \mathcal{D}, sl, B_\pi, \rho), \sigma, sl)$ features the additional parameter of sl to identify which iteration of the signature key was used.

4.2.2.3 Chain-extension protocol

In this section we will give psuedocode for the Praos protocol depicted in Algorithm 7. The code only covers the chain-extension sub-protocol. Because, as we discussed above, in Praos the creation of the randomness for the next epoch is closely tied to the chain extension. Therefore we will combine the two sub-protocols into one pseudocode snippet. Finally we will not present the input endorsement sub-protocol again because it remains largely unchanged.

4.2.3 Further protocol extensions

Ouroboros features many more extensions after Praos. In this section we will quickly go over the features that these extensions provide.

Ouroboros Genesis As we saw before Ouroboros Classic and Praos both require a party to be online at least once every k blocks. This is mainly due to the chain-selection function $maxvalid_k$. The Genesis extension for Ouroboros changes the chain-selection function and enables parties to rejoin the chain even after being offline for more then k blocks. All a party needs is a trusted copy of the genesis block of the chain in order to rejoin. In the same way this extensions also allows new parties to join a long-running chain.

Ouroboros Chronos Ouroboros Chronos aims to eliminate the need for a global clock by introducing a sub-protocol for clock synchronization. This removes the need for a global clock, so that the parties only require local clocks the run at the approximately same speed.

Ouroboros Crypsinous This extension extends Ouroboros into a privacy-preserving protocol. It introduces a new coin evolution technique based on SNARKs and key-private forward-secure encryption.

Ouroboros Hydra The Hydra extension introduces isomorphic multi-party state channels to Ouroboros. The state channels use an adopted version of the smart-contract code base, which allows users to use the same code for both smart contracts and state channels.

Algorithm 7 Pseudocode: Ouroboros Praos chain extension (party p_i).

state

$\mathcal{B}$: local blockchain
h: latest hash of local chain = $H(head(\mathcal{B}))$
$\mathbb{C}$: set of valid chain candidates
vk_i, sk_i: signing key pair for the blocks
vk_i^{vrf}, sk_i^{vrf}: key pair for the VRF proofs
$\mathbb{D}$: list of the last d endorsed inputs to be included in the block
η_j: the randomness for the epoch e_j
T_i^j: the party's threshold of epoch e_j

upon *deliver*$([\text{CHAIN}, C])$ **do**
for every block $B = (h, \mathcal{D}, sl, B_\pi, \rho, \sigma) \in C$ **do**
$Verify_{vk_l}((h, \mathcal{D}, sl, B_\pi, \rho), \sigma, sl)$ // where σ is a key-evolving signature
$Verify(\eta_j \| sl \| \text{NONCE}, \rho_r, \rho_\pi, vk^{vrf})$
$Verify(\eta_j \| sl \| \text{TEST}, r, \pi, vk^{vrf})$
verify $r < T_i^j$
if C passed all block verifications **then**
$\mathbb{C} \leftarrow \mathbb{C} \cup \{C\}$

upon clock indicating the start of a new slot sl **do**
if a new epoch e_j has started for $j \geq 2$ **then**
compute the threshold T_i^j for the leader selection for the next epoch
$\eta_j \leftarrow H(\eta_{j-1} \| j \| v)$

$\mathcal{B} \leftarrow maxvalid_k(\mathcal{B}, \mathbb{C})$

$(\rho_r, \rho_\pi) \leftarrow VRF(\eta_j \| sl \| \text{NONCE}, sk_i^{vrf})$
$(r, \pi) \leftarrow VRF(\eta_j \| sl \| \text{TEST}, sk_i^{vrf})$
if $r < T_i^j$ **then** // tests if p_i is a leader of slot sl
$\mathcal{D} \leftarrow$ select all endorsed inputs from $\mathbb{D}$ that are not included in a block of $\mathcal{B}$
$B_\pi \leftarrow (p_i, r, \pi)$
$\rho \leftarrow (\rho_r, \rho_\pi)$
$B \leftarrow (h, \mathcal{D}, sl, B_\pi, \rho, \sigma)$ where $\sigma = Sign_{sk_i}((h, \mathcal{D}, sl, B_\pi, \rho), sl)$
$\mathcal{B} \leftarrow \mathcal{B} \| B$
$h \leftarrow H(head(\mathcal{B}))$
$broadcast([\text{CHAIN}, \mathcal{B}])$

4.2.4 Protocol attributes

While some aspects are very similar between Ouroboros Classic and Praos the main difference is the selection process. In Classic we have a global function that selects exactly one leader for each slot. This function can be evaluated by each party to learn the leader of all slots for one epoch. This means that an adversary must be limited in its ability to corrupt parties. Otherwise the adversary would be able to corrupt the upcoming leaders in the epoch and therefore securing the majority of blocks that are produced. This would open the opportunity for double-spending attacks. To prevent such a scenario Ouroboros Praos changes the leader selection to a VRF-based mechanism. With this approach the identity of a leader remains hidden until the new block is broadcast. Since the adversary does no longer learn who the leader is ahead of time, the restrictions on the adversary can be lifted. This means that in Ouroboros Praos an adversary is capable to corrupt a party instantly. Note that in order for both protocols to run correctly the adversary can not be allowed to propose more than half the blocks. To prevent this from happening more than $1/2$ of the stake must be controlled by honest parties.

A side effect of the new selection mechanism in Praos is that there may be multiple leaders for one slot. In addition it is also possible that there is no leader at all. Since there can be multiple leaders and therefore multiple blocks per slot, there are going to be forks in Praos. This is the case even if all leaders are honest. In contrast forks in Ouroboros Classic only happen when the leader of a slot is adversarial. Both versions of Ouroboros deal with forks by using the chain selection method $maxvalid_k$. The method simply chooses the longest valid fork as the new local chain. However forks that go back more than k blocks are not considered. This is mainly a defense against long-range attacks and provides a way to enforce that blocks, that are more then k blocks deep in the chain, are final. Otherwise an adversary could construct a fork that goes all the way back to the genesis block. This is feasible because the computation of blocks is cheap.

This comes at a cost however. Since parties are not allowed to consider forks that differ for more than k blocks from their local chain, they can not be absent for more than k slots. Otherwise they would be in a situation where they are unable to select any of the chains suggested to them, because every option is deemed invalid by $maxvalid_k$. Similarly new parties face the same problem, they are unable to obtain the current chain without violating the restrictions that $maxvalid_k$ imposes. It should be noted that the Ouroboros Genesis extension provides a solution for this problem, where parties have a secure way to join or rejoin the protocol.

4.3 Comparison: Algorand and Cardano

In this section we give a short comparison between the Algorand and Cardano. For the sake of this comparison we are mainly focusing on Ouroboros Praos in case of Cardano, since it is the newer version of Ouroboros and is currently being used in practice.

Starting out with the similarities we can see that both protocols use a VRF as the core building block for their role selection, although there are differences in the selection mechanisms. For starters they both have a different solutions of how a party's stake changes their chances of winning the selection. Cardano computes individual thresholds for each party based on their stake, while Algorand lets parties have multiple tries based on the stake they own. Furthermore they pursue different strategies when it comes to how many leaders are likely to be selected for one slot. Algorand chooses its threshold high enough, so that there is always at least one leader being selected. Afterwards multiple selected leaders are narrowed down to one overall winner. Meanwhile even though having a single leader is the most desired outcome in Cardano. Cardano will allow for multiple leaders or even no leaders being selected. In fact such empty slots are indeed even beneficial in Cardano by allowing parties to re-synchronize if needed.

The two protocols further differ in their strategy for the block selection. Cardano allows for all leaders to propose blocks to the other parties. Each party will then decide individually which chain it is going

to adapt based on the options it received. This is done with the chain selection method $maxvalid_k$, that selects the longest valid chain, while considering only chains that do not fork for more than k blocks from the party's local chain. Algorand on the other hand uses a binary-byzantine-agreement protocol, where all honest parties must find consensus on the block that is to be added to the chain. Since the new block is agreed upon by all honest parties it can be considered final as soon as it is added. In Cardano on the other hand a block is only considered final once it is k blocks deep into the chain and is protected by the restrictions of $maxvalid_k$. Therefore Algorand is able to finalize blocks faster than Cardano. However the inclusion of a byzantine-agreement protocol requires Algorand to have more than $2/3$ of the stake to be in the hands of honest parties. In comparison Cardano does only require more than $1/2$ of the stake to be honest.

5
Slashing-based proof-of-stake protocols

In this next section we focus on slashing based proof-of-stake protocols. We first go over the general concepts of slashing and then look more closely at one representative of this family of protocols.

In slashing-based proof-of-stake protocols only a sub-group of parties participates in the chain-extension protocol. To become a member of that group, a party has to lock up some of its stake in a security deposit. The party can then participate in the chain-extension protocol and earn rewards for doing so. However, should a party misbehave and go against the protocol rules, it is penalized. In this case a portion of the party's locked-up stake is slashed (or, in other words, lost). These penalties or slashings are what keeps the parties honest, since in most cases the stake gained from going against the protocol is not worth the penalty. While slashing does incentivize parties to follow a protocol, it is not a chain-extension protocol by itself. Therefore different representatives of slashing-based designs may feature different chain-extension protocols.

We are going to focus on Ethereum 2.0 as the representative for the slashing-based protocols. While slashing is a core concept of Ethereum's design, its chain-extension protocol shares some similarities with lottery-based approaches.

5.1 Ethereum 2.0

In this section we focus on Ethereum 2.0 as a representative of a slashing-based proof-of-stake protocol. Ethereum started out as a proof-of-work-based blockchain protocol and is now in the midst of upgrading to Ethereum 2.0. Our analysis of Ethereum 2.0 is based on the work of Buterin and Griffith [13] and Buterin et al. [14]. We also use additional information from the annotated code specification of Ethereum 2.0 [12].

In Ethereum 2.0 parties can voluntarily lock up some of their stake to participate in the chain-extension protocol. The stake that is required to join the chain extension is the same amount for all parties. A party that takes part in the chain extension this way is called a validator. We will denote such a party as v_i and the set of all validators as $\mathbb{V}$. The set of validators along with their locked-up stake are stored and maintained in the beacon chain. Ethereum 2.0's architecture consists of one beacon chain and several shard chains. The beacon chain keeps track of the validators and manages the chain-extension protocol. It does however not contain transactions and smart contracts. That is where the shard chains come in. Shard chains are responsible for keeping track of stake transactions and smart contracts. The chain extension of

the shard chains is going to be linked to that of the beacon chain. However at this time only the beacon chain has been implemented, therefore we will limit our analysis of Ethereum 2.0 to the chain-extension protocol of the beacon chain.

Ethereum 2.0 uses Gasper as part of its chain-extension protocol. Gasper itself is a combination of two building blocks working together. The first is Casper the Friendly Finality Gadget (Casper FFG). Casper FFG is a protocol that is based on PBFT with the goal of finalizing the blocks in the beacon chain. The second is a chain-selection protocol called LMD GHOST (Latest Message Driven Greediest Heaviest Observed SubTree). We describe the functionality of Casper FFG and LMD GHOST in more detail in later sections.

For its leader selection Ethereum 2.0 randomly selects a leader form the validator set $\mathbb{V}_j$ for each slot sl in epoch e_j. The leaders are selected with the help of a deterministic function $F(\mathbb{S}_j, \eta_j, sl)$. Here $\mathbb{S}_j$ is the stake distribution of the locked-up stake of the validators and η_j the randomness during epoch e_j. The input sl is the identifier of the slot for which the leader is selected. Because all validators locked up the same amount of stake, they have the same probability of being selected. The only exceptions are validators that got their stake slashed due to a violation of the protocol. Those validators have a lesser chance of being selected as the leader compared to everyone else. The leader of the slot computes and broadcast a new block. All other validators will then select their new local chain based on LMD GHOST.

In addition to a leader, each slot also features a committee of attesters. These committees are formed with a deterministic function $C(\eta_j, \mathbb{V}_j)$. This function takes as inputs the randomness η_j and the set of validators $\mathbb{V}_j$ during epoch e_j. The function forms equally sized committees for each slot of epoch e_j. Each validator in $\mathbb{V}_j$ is part of exactly one of these committees per epoch.

Attesters compute and broadcast attestations during the slot where they are part of the committee. Note that because the leader selection and the forming of the committees are independent of each other, there can be the case that a validator is the leader and an attester in the same slot. Since every validator is an attester in exactly one slot, every validator broadcasts exactly one attestation α per epoch. These attestations are used as votes in both Casper FFG for the block finalization and LMD GHOST for the chain selection. In the case of LMD GHOST the attesters vote for the block that they consider the correct head of the chain based on their current view. For Casper FFG the attesters vote on which block should be finalized next. This is done in the form of a checkpoint edge. We now look at Casper FFG and LMD GHOST in more detail.

5.1.1 Casper FFG

As mentioned above Casper FFG is a protocol that is designed to finalize blocks in the beacon chain. Casper FFG is based on PBFT where all validators $\mathbb{V}_j$ of the current epoch e_j participate. The attestations that the validators broadcast during the epoch, serve as votes in the PBFT consensus. Since it takes a whole epoch for all attestations to be computed and broadcast, the validators will only be able to reach consensus on a single block per epoch. These blocks are referred to as checkpoint blocks or epoch boundary blocks. An epoch boundary block is usually the first block in an epoch. If the first block of the epoch should be missing then the last existing block of the previous epoch fills the role. We can think of these checkpoint blocks forming their own checkpoint chain and the validators try to find consensus on which checkpoint block to add to the chain next.

To achieve this the validators vote for the next checkpoint block in their attestations in the form of a checkpoint edge. A checkpoint edge consists of two checkpoint blocks, a source block and a target block. A validator will choose the source block as the current head of the checkpoint chain according to their view $\mathcal{V}$. This block is also referred to the last justified checkpoint block $LJ(\mathcal{V})$. The view $\mathcal{V}$ in this case is a set of all blocks, attestations and supermajority links the validator has witnessed so far. The target block is the epoch boundary block of the current epoch according to the validators local chain $\mathcal{B}$. This is denoted by $LEBB(\mathcal{B})$.

The validators reach consensus on a checkpoint edge if it reaches more than $2/3$ of the votes. Note that the votes of the validators are weighted according to their locked-up stake. So a checkpoint edge needs to gather the support of $2/3$ of the stake. If that is the case the checkpoint edge is considered a supermajority link and is added to the checkpoint chain. Therefore the target block in the supermajority link is now the new head of the checkpoint chain. A block in the checkpoint chain can have two states: justified and finalized. The genesis block is always considered to be both justified and finalized. A block is justified if it is the target of a supermajority link, where the source is a justified block. A block is finalized if it is justified and there is a supermajority link where it is the source.

5.1.2 LMD GHOST

LMD GHOST uses the second vote in the attestations for its chain-selection algorithm. In the second vote a validator votes for the block that they consider the current head of the beacon chain. In short, LMD GHOST selects the chain with the most weight based on the votes. Also here, the weight of a vote is the equal to the locked-up stake of the validator that cast the vote. In more detail LMD GHOST starts its search from the last finalized checkpoint block. It then works its way towards the head of the chain. Each time it reaches a fork it computes the weight of all forks based on the most recent attestations of all validators. It then chooses the fork with the highest weight and continues the search. The search ends when LMD GHOST reaches the head of one of the forks. This block is then selected as the current head of the beacon chain.

5.1.3 Slashing

In Ethereum 2.0 there are three major slashing offences. One of them concerns the leader and two of them the attesters. A leader commits a slashable offence if they ever publish more than one valid block for a single slot sl. For another validator to prove such an offence they need to refer to the two blocks that stand in violation of this rule.

For the two slashing offences an attester can commit, we consider two distinct checkpoint edge votes. We will denote them as (s_1, t_1) and (s_2, t_2). For both slashing offences we are looking at cases where the two distinct votes are in contradiction to each other. This is the case when both edges can not exist in the same fork. The first slashable offence occurs when an attester votes twice for the same epoch. This is the case if $h(t_1) = h(t_2)$, where h is a function that returns the depth or position of the block in the checkpoint chain. Since both blocks would occupy the same position in the checkpoint chain they can not possibly be part of the same fork.

The second slashable offence occurs when an attester casts two votes where one surrounds the other. This is the case when $h(s_1) < h(s_2) < h(t_2) < h(t_1)$. Note that the index does not dictate the order in which the two votes were cast. Therefore the scenario $h(s_2) < h(s_1) < h(t_1) < h(t_2)$ is also in violation of the same rule. In both scenarios the two checkpoint edges can not be part of the same fork without violating the Casper FFG protocol. In order to prove either of the two slashing offences, a validator needs to refer to the two conflicting attestations as evidence.

In all cases if a slashing offence by a validator is proven, then a portion of their locked-up stake is slashed. The validator that presented the evidence is given a portion of that stake as a reward. The amount that is slashed may vary. One deciding factor for example is the amount of slashing offences committed during the epoch. The punishment is more severe when there are more violations. This is done in order to discourage collusion between validators. In any case a slashed validator will be ejected from the validator set after a set amount of epochs. The validator continues to participate in the protocol as an attester or leader until the ejection is in effect. During that time the validator may be subject to further slashings if there are further slashing offences on their part.

5.1.4 Chain-extension protocol

In this section we describe the chain-extension protocol of Ethereum 2.0. The chain-extension protocol is split in four subroutines. The first two handle the delivery of block and attestation messages respectively. The other two become active at the beginning and at the halfway point of a slot. We now describe the subroutines in more detail and give a pseudocode implementation of the chain-extension protocol in Algorithm 8.

On conclusion of previous slot At the beginning of a new slot sl every validator v_i checks if a new epoch e_j has started. If that is the case then the stake distribution $\mathbb{S}_j$ and the randomness η_j for this epoch are computed. The computation of the randomness is based on a RANDAO mix [12]. For Ethereum 2.0 every leader includes a random value ρ in the proposed block. All random values ρ of an epoch are added to the previous epoch randomness η_{j-1} through the *XOR* operation to form the next epoch randomness η_j. Note that in practice the epoch randomness η_j is computed at the end of epoch e_{j-5}. This allows validators to learn in advance, when they have an active role in an upcoming epoch. This way they can better plan, when to be online and participate in the protocol.

The remaining part of this subroutine is then only executed by the leader of the current slot sl. Here the leader of the slot computes and broadcasts a new block $B = (h, newattests, newslashings, \mathcal{D}, \rho, sl, \sigma)$. The hash h is the hash of the previous block in the local chain $\mathcal{B}$. The $newattests$ contains all new attestations that the leader has witnessed and were not included in any previous block. Similarly $newslashings$ contains the evidence for all new slashable offences the leader has discovered. Since the beacon chain does not contain transactions, the set $\mathcal{D}$ simply serves to store additional data. The randomness $\rho = Sign_{sk_i}(e_j)$ is used in the computation for the epoch randomness. It is a BLS-signature over the identifier e_j of the current epoch. In the BLS-signature scheme the signatures are deterministic, meaning that the same key sk_i and message m always lead to the same signature ρ. Here the BLS-signature is equivalent to a *VRF*. Since the message is predetermined, a leader is only able to compute one unique signature ρ. Additionally since it is a signature, the correct construction of ρ can be verified by other parties. Finally the block B also contains the identifier of the current slot sl and a block signature σ over $(h, newattests, newslashings, \mathcal{D}, \rho, sl)$. After the block is computed the leader will then broadcast the block to the other parties.

On halfway point during the slot Halfway through the slot the attesters become active. The main purpose here is for them to compute and broadcast their attestation. They wait until the halfway point of the slot to do this, in order to give the new block time to arrive. This way they can include the new block in their chain selection. The attestation $\alpha = (sl, (source, target), head, \sigma)$ consists of the two votes $(source, target)$ and $head$. Additionally it contains the identifier of the current slot sl and the signature σ over $(sl, (source, target), head)$. The vote for the checkpoint edge $(source, target)$ is chosen as $(LJ(\mathcal{V}), LEBB(\mathcal{B}))$. Here the $LJ(\mathcal{V})$ selects last justified checkpoint block based on the supermajority links that validator v_i has seen so far. $LEBB(\mathcal{B})$ selects the last epoch boundary block based on local chain that validator v_i has selected. This usually is the block from the first slot in the current epoch. Finally the vote for the current head is simply chosen as the head of the current local chain $\mathcal{B}$ selected by validator v_i.

On block message delivery When a new block message arrives the new block B is validated by verifying the signature σ. If block B passed the verification, then it is added to the validator's view $\mathcal{V}$. Afterwards the chain selection $LMD\_GHOST(\mathcal{V})$ is executed to select the new local chain $\mathcal{B}$. Since the new block contains $newattests$, the selection of the chain may have changed.

In the next step the validator v_i needs to tally the new checkpoint-edge votes contained in the new block. If a checkpoint edge reaches the supermajority of $2/3$ of the votes then the checkpoint is added to the view $\mathcal{V}$ as a supermajority link.

And lastly the validator checks for leader and attester slashing offences. For the new block B we search for a second block B' for the same slot sl. For the attestations in the block B we check if any of the new attestations stands in conflict with an attestation that is already part of the local view $\mathcal{V}$. If any slashing offences are found they are added to the $newslashings$ set. Those slashing offences are then included in the next block, when validator v_i is the leader of a slot.

On attestation message delivery In the subroutine for the delivery of a new attestation message we simply verify the signature σ of the attestation α. If the attestation α is valid then we add it to the local view $\mathcal{V}$ for further use.

5.1.5 Protocol attributes

Even tough Ethereum 2.0 is primarily a slashing-based proof-of-stake protocol, it still features a lottery-based leader selection. Similar to Ouroboros Classic, Ethereum 2.0 selects its leaders with a global function, that selects a single leader for each slot. Because of this forks are rare under normal circumstances. An adversarial leader could create a fork by broadcasting two different blocks during their slot, but that would also open them up for being slashed. This in turn would cause them to lose part of their locked up stake and thus would diminish their chances of being selected as leader in future slots. However since the leaders of an epoch are known by everyone ahead of time this makes them potential targets for the adversary. Therefore the adversary in Ethereum 2.0 must be subjected to a corruption delay.

Finalization in Ethereum 2.0 is handled by Casper FFG, a consensus protocol based on PBFT. Instead of finding consensus on each block, Casper FFG reaches consensus on one checkpoint block per epoch. A block that is not a checkpoint block is final as soon as there is a finalized checkpoint block after it in the chain. This usually happens when the checkpoint block of the following epoch is finalized. Therefore the time it takes for a non-checkpoint block to be finalized is three epochs under normal circumstances. Since Casper FFG is a PBFT based consensus protocol, Ethereum 2.0 requires that $2/3$ of all the locked-up stake is controlled by honest validators. Note that through slashing Ethereum 2.0 is able to regulate this ratio to a certain degree. Not only does the possibility of losing stake discourage adversarial behavior, but through slashing the power of caught adversarial validators is diminished until they are ejected from the chain-extension protocol.

Finally Ethereum 2.0 is designed to have a changing set of validators. Therefore it is possible for new validators to join and old validators to leave. Because of this Ethereum 2.0 records all attestations in the blocks. Therefore the entire voting history for both Casper FFG and LMD GHOST is recorded in the chain. This way a new validator is able to recompute and verify the entire history of the chain.

Algorithm 8 Pseudocode: Ethereum 2.0 chain extension (validator v_i)

state
- $\mathcal{B}$: local blockchain
- $\mathcal{V}$: local view of all blocks, attestations and supermajority links
- h: latest hash of local chain = $H(head(\mathcal{B}))$
- e_j: the current epoch of the chain-extension protocol
- sl: current slot of chain-extension protocol
- $\mathbb{V}_j$: set of validators for epoch e_j
- $\mathbb{S}_j$: stake distribution for epoch e_j
- $unordered$: set of additional data to be included in the block
- $newslashings$: set of slashing offences to be included in the block
- vk_i, sk_i: signing key pair for the blocks
- w_j: amount of stake of party p_i in regards to the current epoch e_j
- η_j: the randomness for epoch e_j
- τ_{slot}: duration of a slot

upon *deliver*($[\text{BLOCK}, B]$) **do**
- verify the signature σ over $(h, newattests, newslashings, \mathcal{D}, sl)$
- **if** B passed verification **then**
  - $\mathcal{V} \leftarrow \mathcal{V} \cup B$
  - $\mathcal{B} \leftarrow LMD\_GHOST(\mathcal{V})$
  - update supermajority links in $\mathcal{V}$ based on the results of Casper FFG based on $newattests$ in B
  - **if** there already exists a different valid block $B' \in \mathcal{V}$ for the same slot sl **then**
    - $s \leftarrow$ compute a new leader slashing offence with block pair (B, B') as proof
    - $newslashings \leftarrow newslashings \cup \{s\}$
  - check if any attestation $\alpha \in newattests$ stands in conflict to an earlier attestation $\alpha' \in \mathcal{V}$
  - **for** all such conflicting attestation pairs (α, α') **do**
    - $s \leftarrow$ compute a new attester slashing offence with attestation pair (α, α') as proof
    - $newslashings \leftarrow newslashings \cup \{s\}$

upon *deliver*($[\text{ATTESTATION}, \alpha]$) **do**
- verify the signature σ over $(sl, (source, target), head)$
- **if** α passed verification **then**
  - $\mathcal{V} \leftarrow \mathcal{V} \cup \{\alpha\}$

upon v_i concludes previous slot **do**
- **if** a new epoch e_j has started for $j \geq 2$ **then**
  - compute the stake distribution $\mathbb{S}_j$ for the next epoch
  - $\eta_j \leftarrow \eta_{j-1} \oplus mix_{j-5}$, where mix_{j-5} is the $\oplus$ aggregation of ρ values in blocks from epoch e_{j-5}
- **if** v_i is the leader of slot sl **then**
  - $newattests \leftarrow$ select maximal valid set of new attestations from $\mathcal{V}$
  - $newslashings \leftarrow$ select maximal valid slashing offences from $newslashings$
  - $D \leftarrow$ select maximal valid set of additional data from $unordered$
  - $\rho \leftarrow Sign_{sk_i}(e_j)$
  - $\sigma \leftarrow Sign_{sk_i}(h, newattests, newslashings, \mathcal{D}, \rho, sl)$
  - $B \leftarrow (h, newattests, newslashings, \mathcal{D}, \rho, sl, \sigma)$
  - $\mathcal{V} \leftarrow \mathcal{V} \| B$
  - $\mathcal{B} \leftarrow \mathcal{B} \| B$
  - $h \leftarrow H(head(\mathcal{B}))$
  - *broadcast*($[\text{BLOCK}, B]$)

upon $\tau_{slot}/2$ time has passed in the current slot sl **do**
- **if** v_i is an attester of slot sl **then**
  - $source \leftarrow LJ(\mathcal{V})$
  - $target \leftarrow LEBB(\mathcal{B})$
  - $head \leftarrow head(\mathcal{B})$
  - $\sigma \leftarrow$ signature over $(sl, (source, target), head)$
  - $\alpha \leftarrow (sl, (source, target), head, \sigma)$
  - *broadcast*($[\text{ATTESTATION}, \alpha]$)

6 Voting-based proof-of-stake protocols

In this section we discuss the voting-based protocols. We first go over the general concepts and structure of the voting family of stake-based consensus protocols. Afterwards we give a more detailed description of two representatives for voting-based approaches.

The core idea of voting-based protocols is that the responsibility to extend the chain is delegated to a selected group of parties. The members of this sub-group are selected by vote. While every party in the network is allowed to vote, the weight of a party's vote is tied to the amount of stake they possess. This can be accomplished by either giving each party one vote, that is weighted based on the party's stake, or by giving each party multiple votes based on their stake in the system. In both cases a party's voting power in the election is directly tied to the amount of stake the party has. The election is repeated after every epoch. This gives all parties the opportunity to shift their support based on the performance of the previously elected parties.

The two representatives from the voting family we discuss in this thesis are EOSIO and Neo. While they share the same basic concepts, their implementations of the voting process do deviate in some aspects. For example EOSIO does require the parties to lock up stake in order to be able to vote. Therefore in EOSIO's case only locked-up stake is considered when a party's voting power is computed. In Neo on the other hand all parties are able to vote and their entire stake is considered to determine their voting power. Furthermore in EOSIO voters have up to 30 votes based on their locked-up stake, while in Neo each voter gets exactly one weighted vote.

6.1 EOSIO

In this section we describe the EOSIO protocol as part of the family of voting-based proof-of-stake protocols. Due to the lack of scientific sources describing the consensus protocol of EOSIO, our description is largely based on the documentation of version 2.1 of EOSIO [5]. To supplement the information provided in the documentation, we referred to articles [2, 20] and GitHub issues [3] about the voting and block finalization protocols. It should be noted that while version 2.1 of EOSIO was released in 2020, all other sources were published back in 2018. Interestingly we found mentions of additional features in our 2018 sources that are not described in the documentation. This includes for example slashing offences to exclude adversarial block producers or an alternative protocol for block finalization. However for this

thesis we limit our description of EOSIO to what is described in the documentation of EOSIO v2.1. While a thorough analysis of EOSIO's source code would provide a more complete representation of the protocol, it is however not in the scope of this thesis.

EOSIO uses a combination of two protocols. The first is the chain-extension protocol delegated proof of stake (dPoS). As the name suggests dPoS delegates the chain extension to a selected sub-group of 21 parties. In EOSIO those parties are referred to as delegates, however we will use the term validator instead. The 21 validators are selected through a vote by the parties of the network. Additionally EOSIO uses a pipelined-byzantine-fault-tolerance protocol (pipelined BFT) for the finalization of blocks, which in the documentation is also referred to as asynchronous byzantine fault tolerance (aBFT). We now describe the voting process and the two protocols in more detail and will then give a description of the chain-extension protocol itself.

6.1.1 Voting process

In EOSIO the 21 validators that partake in the chain-extension protocol are elected by vote for every epoch. In order to vote in such an election a party must lock up a portion of their stake. The amount of stake they locked up determines their voting power in the election. For each token that a party staked, it receives one vote in the election for up to a maximum of 30 votes. The weight of those votes are still proportional to the total amount of tokens staked by the party. For example if a party stakes a 100 tokens, then it gets the maximum 30 votes, where each vote has a weight of 100. Furthermore the weight of a party's votes is subject to "decay". Although the term "decay" suggests that a party's voting power decreases over time, the opposite is true. A party is rewarded for actively participating in the elections by increasing the weight of their votes. If a party remains inactive and does not re-cast their votes, the weight of the votes stay the same. While the votes of inactive parties do not lose weight, they do lose voting power relative to active parties.

A party can cast their vote in a vote transaction containing a list of all candidates they vote for. When the result of a vote is computed each party's most recent vote is considered. Then the top 21 candidates are selected based on the weighted total of their received votes. Parties do not have to re-cast their vote for every election in order for it to count. However they are incentivized to do so, through the vote decay mechanism. Finally, we assume that there are mechanisms in place for parties to unlock their stake again, however there is no clear definition of such a mechanism in the provided documentation.

6.1.2 Delegated proof of stake

After the 21 validators for an epoch e_j are selected, those validators are then assigned slots in the epoch where they are the block producer. A block producer is EOSIO's term for the leader role. This schedule $\mathcal{S}_j$ is computed by sorting the elected validators based on their identifiers and assigning the slots according to that order. An epoch in EOSIO has 252 slots which means that each validator is the leader of 12 slots. Those 12 slots are assigned consecutively, meaning that the first 12 slots in an epoch are assigned to the first validator, the second set to the second validator and so on. During the slot the leader computes and broadcasts a new block to extend the chain. If a leader fails to broadcast a block during the time limit of a slot, then that slot remains without a block. If a leader fails to produce a block for multiple slots, then it is replaced with the party that placed 22nd in the last vote.

To resolve possible forks, EOSIO uses the chain-selection algorithm $maxValid_{LIB}(\mathcal{V})$. Here $\mathcal{V}$ represents a party's local view, which contains a record of all proposed blocks that the party has delivered. The method $maxValid_{LIB}(\mathcal{V})$ selects the longest valid chain, out of all chains in a party's view $\mathcal{V}$. In addition the selected chain must contain the last irreversible block (LIB). The LIB is defined as the most recently finalized block, which corresponds to the finalized block with the highest block height. As mentioned the block finalization is handled by the "pipelined BFT", which we discuss in the next section.

6.1.3 Pipelined byzantine fault tolerance

EOSIO uses "pipelined BFT" as its block-finalization protocol, which is derived from PBFT. Similar to other PBFT-like protocols, finding consensus on a new block is divided into three phases. In "pipelined BFT" these phases are named *propose*, *pre-commit* and *commit* phase. These phases correspond to the *pre-prepare*, *prepare* and *commit* phases in PBFT. However the main difference between PBFT and "pipelined BFT" is that the validators in "pipelined BFT" do not use dedicated messages for the consensus, but instead use their produced blocks to communicate. When a validator extends the chain it chooses where the block is added. By adding their block to a fork, a party confirms that they approve of the previous blocks in that fork. A party can specify the number of blocks, it chooses to confirm in the block. Setting the $confirmedBlocks$ field in a block to n, confirms the n previous blocks starting with their own newly added block. We can interpret the addition of a new block to the chain as the party broadcasting a *propose* message for that block. When a validator then extends the fork confirming that block, this can be interpreted as a *pre-commit* message for that block. Once a block is followed by at least $2N/3 + 1 = 15$ blocks by distinct validators, the consensus of this block enters the *commit* phase. Once the same chain is extended by a second set of 15 blocks, the validators reach consensus on the proposed block. The block is then considered final. In short, a block in EOSIO is considered final once two sets of 15 validators have confirmed it by extending the chain containing that block.

6.1.4 Chain-extension protocol

In this section we describe the chain-extension protocol of EOSIO. There are only two subroutines relevant to the chain extension. The first is triggered once a slot concludes and handles the execution of the next slot. The second handles the delivery of incoming block messages. The pseudocode for the chain-extension protocol is given in Algorithm 9.

On start of a new slot A new slot starts as soon as the previous one has ended. The first thing that needs to be checked is if that slot marks the beginning of a new epoch e_j. If that is the case then the validators for the epoch e_{j+2} need to be determined by evaluating the election result. This set of validators $\mathbb{V}_{j+2}$ is then included in the first block of the epoch e_j. This gives the block enough time to be finalized by the start of epoch e_{j+2}. The party then reads the validator set $\mathbb{V}_j$ for the current epoch from the local blockchain $\mathcal{B}$. Afterwards the schedule for the current epoch $\mathcal{S}_j$ is computed by sorting the validators and assigning a block of 12 slots to each of them.

The remainder of the subroutine is then only executed by the leader of the current slot sl. The leader computes and broadcasts a new block $B = (h, sl, p_l, \mathbb{V}_{j+2}, confirmedBlocks, \mathcal{D}, \sigma)$. Here the hash h denotes the hash of the previous block, sl stands for the current slot and p_l is the identifier of the leader that created the block. The set of validators for an upcoming epoch $\mathbb{V}_{j+2}$ is only added in the first slot of an epoch and is left as NULL for the others. The field $confirmedBlocks$ denotes the number of blocks, prior to B, that the validator approves. $ConfirmedBlocks$ covers all blocks between the party's last block and B. Note that the party's previous block is not included in the count, while the block B is. Lastly $\mathcal{D}$ denotes the list of transactions included in the block while σ is the block signature over $(h, sl, p_l, \mathbb{V}_{j+2}, confirmedBlocks, \mathcal{D})$. After the block is computed the leader broadcasts the block to the other parties.

On block message delivery When a new block message is delivered its signature and transactions are verified. If the block is valid then it is added to a party's local view $\mathcal{V}$. The local view contains all valid blocks, that a party has delivered along with their current status regarding finalization. Afterwards the party checks whether there are new finalized blocks based on the confirmations made by that delivered block B.

If there are new finalized blocks according to the "pipelined BFT" protocol, their status is updated in the local view $\mathcal{V}$. Finally a new local chain $\mathcal{B}$ is selected with the $maxValid_{LIB}(\mathcal{V})$ function.

Algorithm 9 Pseudocode: EOSIO chain extension (party p_i)

state
- $\mathcal{B}$: local blockchain
- $\mathcal{V}$: local view of all blocks and block finalizations
- h: latest hash of local chain = $H(head(\mathcal{B}))$
- e_j: the current epoch of the chain-extension protocol
- sl: current slot of chain-extension protocol
- $\mathbb{V}_j$: set of validators for epoch e_j
- $\mathcal{S}_j$: schedule for the leaders in epoch e_j
- $unordered$: set of additional data to be included in the block
- vk_i, sk_i: signing key pair for the blocks
- λ_{slot}: duration of a slot
- $lastConfirmed$: block height of the last block produced by p_i

upon *deliver*([BLOCK, B]) **do**
- verify the signature σ over $(h, sl, p_l, \mathbb{V}_{j+2}, confirmedBlocks, \mathcal{D})$
- verify all transactions in $\mathcal{D}$
- **if** B passed verification **then**
  - $\mathcal{V} \leftarrow \mathcal{V} \cup B$
  - compute new finalized blocks based on block confirmations of Block B
  - change the status of those blocks to *final* in $\mathcal{V}$
  - $\mathcal{B} \leftarrow maxValid_{LIB}(\mathcal{V})$
  - $h \leftarrow H(B)$

upon start of new slot **do**
- **if** a new epoch e_j has started **then**
  - $\mathbb{V}_{j+2} \leftarrow$ compute the Top N candidates // validator set $\mathbb{V}_j$ for e_j has already been determined
  - $\mathcal{S}_j \leftarrow$ compute leader schedule based on validators $\mathbb{V}_j$
- **if** p_i is the leader according to the schedule $\mathcal{S}_j$ **then**
  - $\mathcal{D} \leftarrow$ select maximum set of valid transactions from $unordered$
  - $confirmedBlocks \leftarrow (height(B) + 1) - lastConfirmed$
  - $\sigma \leftarrow (h, sl, p_i, \mathbb{V}_{j+2}, confirmedBlocks, \mathcal{D})$
  - $B \leftarrow (h, sl, p_i, \mathbb{V}_{j+2}, confirmedBlocks, \mathcal{D}, \sigma)$
  - $\mathcal{B} \leftarrow \mathcal{B}||B$
  - $h \leftarrow H(B)$
  - $lastConfirmed \leftarrow height(B)$
  - *broadcast*([BLOCK, B])

6.1.5 Protocol attributes

EOSIO solves the leader selection by having the parties of the network vote on the validators for the chain extension. Once the 21 validators are established, the rest of the scheduling is entirely deterministic. Not even the placement of the validators in the top 21 matters, because the scheduling is ordered by their identifiers rather then their placement. An interesting attribute of the scheduling is that all 12 slots of a validator are placed consecutive to each other. This seems to result in a slower finalization of blocks, since

a single block needs at least two sets of $2/3 + 1$ confirmations from other validators to be considered final. This means that is takes around $12 * 2 * (2N/3 + 1) = 360$ slots to finalize a block. With a round-robin-type scheduling a block could be finalized within around $2 * (2N/3 + 1) = 30$ slots instead. However it is possible that this point has since been addressed, but is not reflected in the EOSIO documentation [5]. Therefore we again emphasize the importance of a thorough analysis of the source code to get a clearer picture.

EOSIO uses a combination of dPoS and "pipelined BFT" for its chain extension. Since dPoS assigns exactly one leader to each slot, forks should only happen when the leader of a slot is adversarial. To resolve possible forks EOSIO uses the $maxValid_{LIB}$ chain selection method. Here the longest valid chain is selected that also includes the most recently finalized block. EOSIO uses "pipelined BFT" as a means to finalize blocks. In order to reach consensus, "pipelined BFT" requires more than $2/3$ of the validators to be honest. However it should be noted that it is not mentioned how this ratio translates to how much honest stake and therefore voting power is needed in order to secure enough honest validators. Therefore EOSIO is difficult to compare directly to other proof-of-stake protocols.

Finally EOSIO stores all information relevant to the chain-extension protocol within the blockchain. Therefore a party can recompute all relevant steps in the dPoS and "pipelined BFT" protocols. This allows all parties in the network to validate the entire history of the chain.

6.2 Neo

In this section we are going to discuss Neo as the second representative of voting-based protocols. The sources for scientific papers describing Neo are sparse. A analysis of an older version of Neo's consensus protocol can be found in the work of Wang et al. [22]. It describes the delegated byzantine fault tolerance (dBFT) protocol as implemented in Neo and formalizes two attacks that allow an adversary to create a fork in the chain. The suggested changes have since been considered in the new version of the protocol dBFT 2.0.

Our work is therefore mainly based on the online documentation of Neo version 3 (N3) [7]. In particular we focus on the the description of the dBFT 2.0 consensus protocol [9] [8]. Finally we consult Neo's source code [6] to supplement the information from the documentation.

As mentioned above, Neo uses dBFT as its chain-extension protocol. Delegated byzantine fault tolerance is a variant of PBFT. One of the main differences is the way dBFT selects its participants for the consensus. The participants or delegates are chosen by vote. Every party in the network can cast a vote in the election, where the weight of a party's vote depends on the stake held by that party. We now describe the voting process and the chain-extension protocol in more detail in the following sections.

6.2.1 Voting process

In Neo an election is being held every epoch to determine both the committee members and validators. The committee members in Neo's case are responsible for the governance of the Neo network. Based on the result of the election the top 21 candidates are selected as committee members. The top 7 candidates additionally earn the right to participate in the chain-extension protocol as validators. Therefore the validators are a subgroup of the committee members.

The election results are recomputed at the start of every epoch, which in Neo's case lasts for 21 slots. Every party in the network may vote in the elections. However in order to participate in the election as a candidate, parties need to register. This is done with a registration transaction, where a party pays a one-time fee in order to register as a candidate.

Even though the election result is recomputed every epoch, the parties do not have to re-cast their vote every time. Rather they vote by submitting a vote transaction to the blockchain, where they declare who

they vote for. When the result of an election is computed only each party's most recent vote transaction is considered. The weight of a party's vote directly correlates with the stake that the party owns at the moment of the evaluation.

Parties are incentivized to participate in the vote by obtaining rewards when the candidate they voted for is elected as a committee member or a validator. In fact voting for a validator earns the party double the award compared to a committee member. Not only does this reward system encourage casting a vote, but seems to incentivize parties to vote for candidates that are likely to win a validator role. Furthermore note that parties are able to compute the current election result before they cast their vote. Which means they are able to compute who is likely to win under the current circumstances.

6.2.2 Chain-extension protocol

As mentioned, the 7 validators that participate in the chain-extension protocol are evaluated at the beginning of each epoch. The epoch itself is then divided into 21 slots. Neo refers to the leader role in the protocol as primary or speaker. The leader is selected by evaluating $v_l = (\mathit{height}(\mathcal{B}) - v) \mod N$. Here $\mathit{height}(\mathcal{B})$ denotes the height of the blockchain, v describes the current view in the dBFT protocol and N denotes the number of validators, in this case 7. The N validators are indexed from 0 to $N - 1$. Therefore $(\mathit{height}(\mathcal{B}) - v) \mod N$ outputs the index of the selected leader. The view v describes the current attempt to reach consensus on the block proposed by the leader of the view. A view ends if the validators reach consensus on the proposed block or if the they decide to replace the current leader and start a new view. If the validators are unable to find consensus under the current leader, they can request to start a new view with a *change-view-request* message. In this case the variable v is increased by one, which results in a new leader being selected in the next view. A slot ends when the validators reach consensus on a new block. In this case the view v is reset to zero for the next slot.

We divide the description of the chain-extension protocol into three parts. The first describes the consensus protocol itself, in the second part we discuss how the validators can request a new leader through a view change and lastly we will describe the recovery mechanisms of the protocol. We give a pseudocode implementation of the protocol in Algorithms 10 and 11. Algorithm 10 contains the subroutines for the consensus, while Algorithm 11 contains the the change view and recovery subroutines.

6.2.2.1 Consensus protocol

The dBFT 2.0 consensus protocol consists of three phases. The *prepare request* phase, the *prepare response* phase and the *commit* phase. If the validators manage to finish all three phases successfully then they find consensus on a new block to add to the blockchain. If at any point a validator detects a problem, it can request a view change to restart the process with a different leader. Several subroutines are involved here, three of which handle the delivery for the *prepare request*, *prepare response* and *commit* messages. These subroutines can be found in Algorithm 11. The other two are triggered at the start of a new slot and at the start of a new view and can be found in Algorithm 10.

On conclusion of previous slot A new slot sl starts directly after the previous one concluded. At the beginning of a new slot a party checks if a new epoch e_j has started. If that is the case then the validators $\mathbb{V}_j$ for the new epoch are computed. This is done by collecting the most recent vote transaction of each party from the blockchain $\mathcal{B}$. A party's vote is weighted by the stake w_j it possesses at the beginning of epoch e_j. The weighted votes are then tallied and the top N candidates are selected as validators $\mathbb{V}_j$ for the epoch. The validators will then start the first view of the slot sl, where the view number v is reset to zero.

On the start of a new view This is the main subroutine of the dBFT 2.0 consensus protocol. Here the validators try to find consensus over the three phases of the protocol, the *prepare request*, the *prepare*

response and the *commit* phase. The subroutine starts with the *prepare request* phase. Here the leader of the current view is determined by evaluating $v_l = (\textit{height}(\mathcal{B}) - v) \mod N$. The leader will then propose a list of transactions $\mathcal{D}$ to the other validators. Note that the leader does not propose an entire block, but just the data to be included in the new block. The leader will then broadcast that data to the other validators in a *prepare request* message $(sl, v, p_i, h, \mathcal{D}, \sigma)$. Here sl denotes the current slot, v describes which view the message is for and p_i is the identifier of the sender. Note that these three variables are included in every consensus message along with a message signature σ. The variables unique to the *prepare request* are h the hash of the current head of the blockchain and $\mathcal{D}$ the list of transactions proposed by the leader.

Meanwhile the other validators are waiting for the *prepare request* message to arrive. The time that they wait is $\lambda_{timeout}$. If a validator does not deliver a *prepare request* message during that period of time then it will abort the current view and request a view change. We describe this procedure in the next section. If a valid *prepare request* does arrive then the validator will start the *prepare response* phase by computing and broadcasting a *prepare response* message $(sl, v, p_i, h_{pr}, \sigma)$. The hash h_{pr} is the hash of the *prepare request* message $H(prepareRequest)$ that validator v_i received. By broadcasting this *prepare response* a validator confirms, that it accepts the list of transactions $\mathcal{D}$ in the *prepare request* message.

In the second half of the *prepare response* phase all validators, including the leader, wait for *prepare response* messages to arrive. If they receive M *prepare responses* for the same *prepare request*, then they will move on to the *commit* phase. Here M denotes the number of validators needed to reach consensus. In Neo M is defined as $M = \lfloor 2N/3 \rfloor + 1$, which for $N = 7$ amounts to $M = 5$. If a validator waits longer than $\lambda_{timeout}$, the *prepare response* phase will end in a timeout. In this case the validator will abort the current view and request a view change.

Once a validator enters the *commit* phase it will then compute and broadcast a *commit* message $(sl, v, p_i, \sigma_{commit}, \sigma)$. The signature σ_{commit} is a signature over the content of the next block $(h, sl, \mathcal{D})$. With the *commit* message a validator confirms that it has seen enough *prepare response* messages to determine the consensus on the next block. Once a validator is committed to a block it can no longer commit to any other block during the same slot sl.

The second half of the *commit* phase works similar to that of the *prepare response* phase. All validators that have broadcast a *commit* message will then wait for M *commit* messages to arrive. If there are enough messages, then the validator can conclude the consensus protocol by computing and broadcasting the new block $B = (h, sl, \mathcal{D}, signatures)$. Here instead of a single block signature by the leader, we include all M signatures σ_{commit} from the *commit* messages. These $signatures$ then not only show the authenticity of the block, but also demonstrate that the block gathered enough support during the consensus protocol. Once the new block is broadcast, the next slot begins. Finally if there are not enough *commit* messages before the timeout, then the validator will compute and broadcast a *recovery* message and restart the timer. Since the validator has already committed to a block, it can not request a view change. Because a view change would lead to a new list of transactions being proposed. Instead a *recovery* message is sent with the goal to help the other validators progress to the *commit* phase. We discuss the recovery process in more detail in a later section.

On prepare request delivery On the delivery of a new *prepare request* message, the *prepare request* is validated. In particular this includes the validation of the transaction data $\mathcal{D}'$. The *prepare request* is accepted, if all the transactions in $\mathcal{D}'$ are valid. Once the *prepare request* is accepted it is saved to the validators local state. If the leader's proposed list of transactions $\mathcal{D}'$ did not pass the validation, then the validator requests a view change, in order to replace the current leader.

On prepare response delivery When a new *prepare response* is delivered it is first validated. If the message passes all validations then it is saved to a local set of *prepare responses*. Afterwards the timer for incoming *prepare responses* is reset. Therefore a timeout only takes place, if the time between two new *prepare response* messages is more than $\lambda_{timeout}$.

On commit delivery The delivery of new *commit* messages is handled similarly to that of the *prepare response* messages. New *commit* messages are validated when they are delivered. If the *commit* message passes all validations then it is added to the local set of delivered *commit* messages. Afterwards the timer for new *commit* messages to arrive is reset to $\lambda_{timeout}$.

6.2.2.2 Change view

In this section we describe the process for the validators to request and execute a view change. A view change replaces the current leader and starts a new view of the consensus protocol. Therefore validators will request a view change, when they detect a reason for why reaching consensus is not possible under the current leader. Those reasons include the leader proposing invalid transaction data or messages not reaching a validator in time. There are two subroutines involved in this process. The first is the *abortCurrentView*$(reason)$ method, which a validator invokes if they encounter a reason to change the current view. The second is the delivery of the *change-view-request* messages. Both of them can be found in Algorithm 11.

abortCurrentView() method In this method a validator first checks if reaching consensus is still possible with the remaining validators. For this purpose a party computes two sets of validators $\mathbb{V}_{failed}$ and $\mathbb{V}_{committed}$. $\mathbb{V}_{failed}$ is the set of validators, that have not sent a message in a while. This could be due to a crash or because they are cut off from the rest of the validators. The validators in set $\mathbb{V}_{committed}$ have already sent their *commit* message and are therefore no longer able to participate in the consensus. The validator then checks if there are still at least M active validators left in order to reach consensus. If that is the case then it will compute and broadcast a *change view request* $(sl, v+1, p_i, timestamp, reason, \sigma)$. The view counter v will only be incremented by one for the next view. In addition the *change view request* contains a $timestamp$ and a $reason$ for the view change. If the number of remaining validators is not enough to reach consensus then the validator will compute and broadcast a *recovery request* instead $(sl, v, p_i, timestamp, \sigma)$. The *recovery request* only features a $timestamp$ in addition to the other data contained in all consensus messages. The recovery process is described in a later section.

On change view request delivery When a new *change view request* arrives it is first validated. If the message is not valid then it is simply ignored. Otherwise it is added to the local set of delivered *change-view-request* messages. Next the requested new view number v_r is compared to the local view v. If the requested view number v_r is equal or smaller to the local view v, then the *change view request* is instead treated as a *recovery request* and the *handleRecoveryRequest* method is invoked. The reason here is that the sender seems to be stuck in a previous view and is therefore out of sync with the other validators. Otherwise if v_r is greater then v the processing of the *change view request* continues normally. Additionally validators that already committed to an outcome of the consensus are unable to participate in a view change and will therefore ignore the request. In all other cases the validator will then check if it has delivered at least M *change view requests* for the view v_r. If that is the case it will immediately start a new view of consensus with view v_r.

6.2.2.3 Recovery

Finally the recovery process is invoked if the validators are stuck and are unable to find consensus. Additionally the recovery process is also requested when a validator detects that they are out of sync. During the recovery the validators will share all valid messages that they delivered during this slot. All validators will then process the messages that they have not delivered themselves yet. This way all validators can catch up on messages that they missed and are able to continue with the consensus protocol.

Algorithm 10 Pseudocode: Neo chain extension (party p_i) Part 1

state
- $\mathcal{B}$: local blockchain
- h: latest hash of local chain = $H(head(\mathcal{B}))$
- e_j: the current epoch of the chain-extension protocol
- sl: current slot of chain-extension protocol
- $\mathbb{V}_j$: set of validators for epoch e_j
- $unordered$: set of transaction data to be included in the block
- $\mathcal{D}$: list of transactions included in the block
- vk_i, sk_i: signing key pair for the blocks
- $\lambda_{timeout}$: duration a party waits for messages
- v: the current view of dBFT protocol
- $prepareRequest$: the *prepare request* message of the current view
- $prepareResponses$: the set of *prepare response* messages of the current view
- $commits$: the set of *commit* messages of the current view
- $changeViewRequests$: the set of change view messages of the current view

upon p_i concludes previous slot **do**
- **if** a new epoch e_j has started **then**
  - $\mathbb{V}_j \leftarrow$ compute the Top N candidates based on every party's most recent vote
- **if** $p_i \in \mathbb{V}_j$ **then**
  - $prepareRequest, prepareResponses, commits, changeViewRequests \leftarrow \emptyset$
  - $v \leftarrow 0$
  - **start** new view with v set to 0

upon start of a new view v for the current slot sl **do**
- **if** p_i is the leader for the view v **then**
  - $\mathcal{D} \leftarrow$ select maximum valid transactions from $unordered$
  - $prepareRequest \leftarrow (sl, v, p_i, h, \mathcal{D})$
  - $\sigma \leftarrow Sign_{sk_i}(sl, v, p_i, h, \mathcal{D})$
  - *broadcast*([PREPARE REQUEST, $(sl, v, p_i, h, \mathcal{D}, \sigma)$])
- **else**
  - **wait**($\lambda_{timeout}$) // wait for *prepare request* message to arrive
  - **if** $prepareRequest$ is set **then**
    - $h_{pr} \leftarrow H(prepareRequest)$
    - $\sigma \leftarrow Sign_{sk_i}((sl, v, p_i, h_{pr}))$
    - *broadcast*([PREPARE RESPONSE, $(sl, v, p_i, h_{pr}, \sigma)$])
  - **else** // timeout
    - *abortCurrentView*("timeout for prepare request")

  - **wait**($\lambda_{timeout}$) // wait while gathering *prepare response* messages
  - **if** there are M *prepare response* messages for the same hash h_{pr} **then**
    - $\sigma_{commit} \leftarrow Sign_{sk_i}((h, sl, \mathcal{D}))$
    - $\sigma \leftarrow Sign_{sk_i}((sl, v, p_i, \sigma_{commit}))$
    - *broadcast*([COMMIT, $(sl, v, p_i, \sigma_{commit}, \sigma)$])
  - **else** // timeout
    - *abortCurrentView*("timeout for prepare responses")

  - **wait**($\lambda_{timeout}$) // wait while gathering *commit* messages
  - **if** there are M *commit* messages for the same hash h_{pr} **then**
    - $signatures \leftarrow$ select all block signatures σ_{commit} from $commits$
    - $B \leftarrow (h, sl, \mathcal{D}, signatures)$
    - $\mathcal{B} \leftarrow \mathcal{B}||B$
    - $h \leftarrow H(B)$
    - *broadcast*([BLOCK, B])
  - **else** // timeout
    - $messages \leftarrow (prepareRequest, prepareResponses, commits, changeViewRequests)$
    - $\sigma \leftarrow Sign_{sk_i}(sl, v, p_i, messages)$
    - *broadcast*([RECOVERY, $(sl, v, p_i, messages, \sigma)$])
    - restart clock for incoming *commit* messages

On recovery request delivery When a new *recovery request* is delivered then it is immediately validated. If the *recovery request* is valid then it is passed to the *handleRecoveryRequest* method.

handleRecoveryRequest() method In the *handleRecoveryRequest* a validator decides whether or not they will reply to the request with a recovery message. If the validator has already broadcast a *commit* message during this slot then it will answer the *recovery request*. Otherwise the validator will only answer if their index v_i is among the F indices that follow the senders index v_s. F in this case is the number of maximum amount of faulty validators, that dBFT 2.0 can handle ($F = 2$ in N3). Presumably this condition is inserted to avoid the network being flooded with to many *recovery* messages. However the exact motivation behind this condition is not communicated in Neo's documentation. In any case if a validator has decided to answer the request, it will then compute and broadcast a *recovery* message $(sl, v, p_i, messages, \sigma)$. Here $messages$ is a collection of all valid *prepare request*, *prepare response*, *commit* and *change-view-request* messages that the validator delivered during this slot.

On recovery message delivery On the arrival of a new *recovery* message the validator will first verify the message's validity. If the message is valid the validator will compare the view v_r from the message with the local view v in order to decide which messages to process. If the local view v is smaller then that of the message v_r, then that means that the validator is behind compared to the sender. Therefore the validator will process the change view messages contained in the *recovery* message in order to catch up.

If the local view v is bigger, then that of sender v_r this means that the validator is ahead. The validator in this case will process the *prepare request* and *prepare response* messages, in order to determine whether or not to switch back to the sender's view v_r. Afterwards the validator will then also process the *commit* messages. And finally if the two views are the same then the validator will directly process the *commit* messages.

While processing the messages in the *recovery* message the validators will repeat the same steps as if the messages were delivered normally. If possible the validators will then continue the consensus protocol with the new information from the *recovery* message.

6.2.3 Protocol attributes

In Neo the leader selection is not based on random selection. Rather Neo determines the validators that participate in the chain-extension protocol through voting. The validators then take turns to act as a leader for the duration of an epoch. Since the election results are public knowledge, the elected validators are at risk of being targeted by an adversary. In addition Neo limits the participants to 7, which means that it can only tolerate up to 2 adversarial parties. Therefore an adversary only needs to control 3 validators in order to prevent the honest validators from reaching consensus.

For the chain extension Neo uses the dBFT 2.0 protocol, which is a variant of PBFT. Therefore Neo does not feature any forks. Since dBFT 2.0 is a byzantine-consensus protocol, more than $2/3$ of the participants need to be honest. It is worth noting that there is no mention in the documentation of how this translates to the honest stake needed in order to ensure at least 5 honest validators being elected.

Finally since blocks include the block signatures of M validators, all parties in the network are able to verify that more than $2/3$ validators agree with a block. Therefore it is possible for a single party to validate the entire chain by themselves with no additional input.

6.3 Comparison: EOSIO and Neo

In this section we make a small comparison between the two voting-based protocols EOSIO and Neo. We compare their voting processes as well as their chain-extension protocols.

Algorithm 11 Pseudocode: Neo chain extension (cont.) (party p_i) Part 2

upon *deliver*([PREPARE REQUEST, $(sl, v, p_s, h, \mathcal{D}', \sigma)$]) **do**
  verify signature σ over $(sl, v, p_s, h, \mathcal{D}')$
  verify transaction data $\mathcal{D}'$ in $prepareRequest$
  if all transaction $x \in \mathcal{D}'$ are valid **then**
    $prepareRequest \leftarrow (sl, v, p_s, h, \mathcal{D}', \sigma)$
    $\mathcal{D} \leftarrow \mathcal{D}'$
  else
    abortCurrentView("invalid transactions")

upon *deliver*([PREPARE RESPONSE, $(sl, v, p_s, h_{pr}, \sigma)$]) **do**
  verify σ over (sl, v, p_s, h_{pr})
  if prepare response is valid **then**
    $prepareResponses \leftarrow prepareResponses \cup \{prepareResponse\}$
    restart clock for incoming prepare response messages

upon *deliver*([COMMIT, $(sl, v, p_s, \sigma_{commit}, \sigma)$]) **do**
  verify message signature σ over $(sl, v, p_i, \sigma_{commit})$
  verify commit signature σ over $(h, sl, \mathcal{D})$
  if commit is valid **then**
    $commits \leftarrow commits \cup \{commit\}$
    restart clock for incoming *prepare response* messages

function *abortCurrentView*($reason$)
  if $N - (|\mathbb{V}_{failed}| + |\mathbb{V}_{committed}|) < M$) **then** //remaining validators unable to reach consensus
    $\sigma \leftarrow$ *Sign*$_{sk_i}(sl, v, p_i, timestamp)$
    broadcast([RECOVERY REQUEST, $(sl, v, p_i, timestamp, \sigma)$])
  else
    $\sigma \leftarrow$ *Sign*$_{sk_i}(sl, v+1, p_i, timestamp, reason)$
    broadcast([CHANGEVIEW REQUEST, $(sl, v+1, p_i, timestamp, reason, \sigma)$])

upon *deliver*([CHANGEVIEW REQUEST, $(sl, v, p_s, timestamp, reason)$]) **do**
  if *change view request* is valid **then**
    $changeViewRequests \leftarrow changeViewRequests \cup \{changeViewRequest\}$
    if $v_r \leq v$ **then** // received *change view request* for a previous view
      handleRecoveryRequest($changeViewRequest$)
    else if p_i has not yet sent a *commit* message for current slot sl **then**
      if p_i delivered M change view messages for the same view v_r **then**
        start new view with v_r

upon *deliver*([RECOVERY REQUEST, $(sl, v, p_s, timestamp)$]) **do**
  if *recovery request* is valid **then**
    handleRecoveryRequest($recoveryRequest$)

function *handleRecoveryRequest*($(sl, v, p_s, timestamp)$)
  if p_i has already sent a commit **or** $p_i \in [p_{sender} + 1, p_{sender} + F] \mod N$ **then**
    $messages \leftarrow (prepareRequest, prepareResponses, commits, changeViewRequests)$
    $\sigma \leftarrow$ *Sign*$_{sk_i}(sl, v, p_i, messages)$
    broadcast([RECOVERY, $(sl, v, p_i, messages, \sigma)$])

upon *deliver*([RECOVERY, $(sl, v_r, p_s, messages)$]) **do**
  if *recovery* message is valid **then**
    if $v < v_r$ **then**
      process all new $changeViewRequests$ in $messages$
    else
      if $v > v_r$ **then**
        process $prepareRequest$ from $messages$
        process all new $prepareResponses$ from $messages$
      process all new $commits$ from $messages$

Even tough both EOSIO and Neo use voting as a way to select the participants for their chain-extension protocols, their voting processes feature some key differences. This starts with the electorate. While in Neo any party is able to vote in an election, EOSIO requires a party to lock up stake in order to vote. Furthermore the voting power in the two protocols is computed differently. In Neo a party receives one vote that is weighted by the party's stake. In EOSIO on the other hand a party can receive up to 30 votes depending on their invested stake. Additionally the weight of those votes is proportional to the party's locked-up stake and is further influenced by vote decay. Vote decay is the main incentive in EOSIO for voters to actively participate in the elections. This is because voters are rewarded with a higher voting power whenever they cast a vote. In contrast, Neo rewards voters that voted for a winning candidate, with a portion of the generated stake from the chain extension.

What both protocols have in common however is that it remains unclear how those definitions of voting power and voting incentives influence the result of the vote. Specifically there is no mention of how much honest stake or voting power is likely needed to ensure that more then $2/3$ of the elected candidates are honest. Additionally selecting participants in the chain-extension protocol through a public vote means that the validators are publicly known as well. This makes them more at risk to be targeted by an adversary. Furthermore the number of validators in both cases is rather low compared to other protocols. At 7 validators for Neo and 21 for EOSIO the two protocols can only deal with 2 and 6 adversarial validators respectively.

In terms of their chain extension Neo only uses dBFT while EOSIO uses a combination of dPoS and "pipelined BFT". Since Neo uses a form of BFT in its chain extension, blocks in Neo are final as soon as they are added to the chain. In EOSIO however the finalization of blocks is delayed and only happens after about 360 slots or 2 epochs.

7 Hybrid-voting-slashing proof-of-stake protocols

In this section we discuss a protocol that combines ideas from the voting and slashing families. We first go over the general concepts that are adapted into this hybrid approach. Finally we describe a representative of the hybrid-voting-slashing family.

Like in voting-based approaches, here a sub-group of parties is elected to participate in the chain-extension protocol. A party's voting power in the election is based on their stake. Here the candidates or in some cases voters need to lock up stake in order to participate in the voting process. Like in slashing based protocols, this locked-up stake serves as collateral for when the party misbehaves. This has the goal of keeping a party honest, since in most cases the stake gained from misbehaving is not worth the punishment.

For this thesis we are looking at COSMOS as a representative of voting-slashing hybrids. In terms of slashing COSMOS does target the elected candidates as well as the voters. Therefore not only the misbehaving party is subject to the slashing, but also the parties that voted for it. Therefore in COSMOS voters are held accountable for the behavior of the candidates they support.

7.1 COSMOS

In this section we discuss the voting-slashing-hybrid protocol COSMOS. We base our description of COSMOS on its whitepaper [4]. We supplement this with additional information from the validators FAQ on the COSMOS git repository [1]. Additionally COSMOS uses Tendermint as its chain-extension protocol. For Tendermint we use the works of Buchman et al. [11] as well as the documentation of Tendermint [10]. Note that even though both versions of Tendermint are largely the same, they do have some differences. This mainly concerns the mechanics of locking and unlocking a value during the consensus. Since the documentation [10] is the more recent source of the two, we will base our description of the Tendermint consensus protocol on that.

As mentioned, COSMOS uses Tendermint as its chain-extension protocol. Tendermint itself is a variant of practical byzantine fault tolerance (PBFT). The main difference to PBFT is that Tendermint integrates the round change and recovery mechanisms of PBFT into the normal flow of the *pre-prepare*, *prepare* and *commit* phases. COSMOS only allows for a limited number of parties to participate in the Tendermint consensus. These participants, referred to as validators, are selected through a vote by the parties in the

network. COSMOS combines the ideas of voting and slashing by holding the voters accountable for the actions of their candidate. When a validator misbehaves it suffers the consequences of slashing, however voters that supported that validator are affected by the slashing as well. So the slashing in COSMOS has not only the goal of keeping the validators honest, but also to encourage voters to support validators that they trust. We now discuss the voting process and the slashing mechanisms in more detail in the following sections and finally give a description of the Tendermint chain-extension protocol.

7.1.1 Voting process

Voting in COSMOS is a continuous process, meaning that the validators are evaluated after each new block. The validator spots in COSMOS are limited. That limit started out at 100 validators and gets raised every year to an overall maximum of 300 validators. To participate in the vote a party needs to lock up some of their stake. The amount of stake can be chosen freely by the party and represents the weight of the party's vote. In COSMOS this process is called bonding and can be achieved through making a special bonding transaction. A party can cast exactly one vote per bonding transaction. However a party can cast additional votes by making multiple bonding transactions. Note that for each bonding transaction an additional portion of a party's stake is locked up. A vote will always go towards exactly one candidate. Voters can either vote for themselves or any other candidate. In COSMOS there are no conditions to becoming a candidate. Therefore it is theoretically possible for a party to become a validator without voting for themselves.

The evaluation of the result is done after each new block. Based on all the votes the candidates are sorted based on the weighted total of the votes they received. Then the top N candidates are selected as the new validators. Here N denotes the number of validators that COSMOS currently allows. However an important difference to other voting-based protocols is that the validator's weighted total corresponds directly to its voting power in the Tendermint protocol. Finally if a party does drop out of the top N, then all stake bonded to that party is automatically unbonded. This means that all votes for that party are revoked and the locked-up stake is returned to the corresponding parties after a set unbonding period. Note that a party may also revoke their vote manually by making a unbonding transaction.

7.1.2 Slashing

In COSMOS there are four situations that lead to a validator being slashed. Three of them are based on the validator itself committing a slashable offence, while the fourth is caused by a validator being hacked. The three slashable offences, which a validator can commit are double-signing, unavailability and non-voting. Double-signing describes the situation where a validator signs two conflicting votes during the Tendermint consensus. In order for the slashing to come into effect, a second party can gather the two conflicting votes as evidence and submit them in a dedicated slashing transaction. The consequence for double-signing is a portion of the locked-up stake being slashed as well as the validator being unbonded. This leads to the guilty party losing its role as a validator. The slashing for unavailability comes into effect if a validator does not participate in the consensus protocol for several blocks. In order to determine a party's participation, a record of the votes during the last round of the consensus is included in each block. A party is considered inactive if the party's vote is not included in that record. The punishment for unavailability is proportional to the amount of blocks that a party was inactive for. Additionally if the number of missed blocks goes over a set threshold, then the inactive validator is unbonded. Finally non-voting describes the situation where a validator did not participate in a governance vote. This only leads to a minor portion of the stake being slashed and does not lead to the exclusion of the validator from the chain-extension protocol. In all three instances, the slashing is targeted at all parties which currently support the validator. This may not include the validator itself depending on whether it voted for itself or not. Note that only the stake that is tied to the party's vote for the misbehaving validator is affected by the slashing. A party's unbonded stake

and locked-up stake for other votes are not affected.

The fourth cause for a validator and its voters being slashed is that the validator was hacked. COSMOS features a bounty system, where hackers can claim a bounty after hacking a validator. This is done through a bounty transaction, which a hacker makes in the name of the hacked validator by signing the transaction with the validator's private signature key. The bounty consists of the slashed stake of all voters which supported the hacked validator with their vote. Additionally the validator is removed from its role and the remaining stake is unbonded. This bounty system discourages parties from pooling all the voting power on a few validators. Because a large amount of voting power leads to a higher bounty and therefore makes the validator a more attractive target for hackers. All in all, the slashing conditions in COSMOS should motivate parties to vote for candidates that they trust and to divide up their votes on multiple candidates instead investing all their stake into a single candidate.

7.1.3 Chain-extension protocol

For the chain extension, COSMOS uses the Terndermint consensus protocol. Tendermint itself is a variant of PBFT. Therefore in Tendermint the validators try to find consensus on the next block over several rounds. A round in Tendermint consists of three separate phases the *propose*, the *prevote* and the *precommit* phase. The three phases correspond to the *pre-prepare*, *prepare* and *commit* phases in the PBFT protocol. What sets Tendermint apart from other PBFT protocols is that it integrates the round change and recovery mechanisms into the normal flow of its rounds. Additionally Tendermint features a *commit* step, where the new found block is broadcast by the validators and a newHeight step, where the validators setup the consensus for the next block. In addition to these three subroutines we describe how Tendermint handles the delivery of its *propose*, *prevote* and *precommit* messages. The pseudocode for Tendermint is given in Algorithm 12 and Algorithm 13. Algorithm 12 contains the newHeight and *commit* steps as well as the delivery of the messages. Meanwhile Algorithm 13 describes how a round of consensus is handled.

On start of newHeight step This subroutine is used to prepare the consensus for the next block. It starts out with the party waiting on additional *precommit* messages from the last round of the previous consensus. These *precommit* messages are stored in the local variable $lastCommits$ to be included in the next block. There are two reasons for this. The first is to prove that the validators did indeed reach consensus on the previous block and the second is to record a validator's participation in the consensus. After this the new validator set $\mathbb{V}$ for the upcoming block height H is determined based on the local blockchain $\mathcal{B}$. Afterwards the party resets its state to prepare for the next consensus. Finally if the party is a validator for the new block height H, then it will start the first round of consensus with round R set to zero.

On start of new round of consensus One round of the consensus protocol consists of three phases. The *propose* phase, where the leader of the round computes and proposes a new block. The *prevote* phase, where the validators signal their approval of the proposed block by broadcasting a *prevote* message. Lastly the *precommit* phase, where the validators signal that they are ready to commit to a block with a *precommit* message. In contrast to other PBFT-based consensus protocols, in Tendermint the validators will run through all three phases in each round even if it is already clear that a consensus can not be reached. Furthermore the validators will always send *prevote* and *precommit* messages even if they do not agree with the proposed block. Instead they either vote for a block from a previous round or the value NIL.

A round starts of with the *propose* phase. Here a validator evaluates the deterministic function $F(H, R)$ to determine the leader of this round. The inputs here are the current block height H and the current round R. The function F selects the next leader through weighted round-robin, where the weight corresponds to a validators voting power. If the validator is the leader of this round it will then propose a block B. This block B can either be a newly computed block or a block that was proposed in a previous round. This decision depends on whether or not the leader has already precommitted to a block in a previous round. If

a validator is currently precommitted to a block the validator is considered to be locked. Therefore if the leader is still locked on a previous block, it will include that block in its proposal. Note that a validator is considered locked if the variable $lockedRound > -1$. Otherwise if a leader is not locked then it will compute and broadcast a new block $B = (h, H, lastCommits, \mathcal{D}, \sigma_{block})$. Here h denotes the hash of the previous block, H stands for the current block height, $lastCommits$ is the set of *precommits* that lead to the consensus on the previous block, $\mathcal{D}$ is a list of transactions and σ_{block} is the block signature.

Finally the leader then computes and broadcasts the *propose* message $(H, R, B, PoLC, \sigma)$. Here H and R describe the current block height and round, B denotes the proposed block and σ the message signature. Finally the $PoLC$ stands for proof of lock change. This is a set of *prevote* messages from a previous round. The $PoLC$ is only set if in a previous round there was $2/3$ majority of prevotes for a value v. Here a $2/3$ majority means that more than $2/3$ of the overall voting power voted for the value v. With the $PoLC$ the leader may convince the other validators to unlock their current lock. The $PoLC$ is only included if the leader has previously witnessed such a $2/3$ majority in a *prevote*. Otherwise it is set to NIL.

Meanwhile the other validators wait for the *propose* message to arrive. If no valid proposal arrives during the waiting time of $\lambda_{timeout}$, then the validator will vote for the value $v_i = \text{NIL}$ in the *prevote*. Otherwise if a valid proposal does arrive the validator will vote for the proposed block B, provided the it is not locked into another value. Should the validator still be locked, it will vote for the $lockedValue$ instead. Finally the validator may be convinced through the $PoLC_{propose}$ in the *propose* message to unlock itself prior to that decision. A validator will unlock if the provided $PoLC_{propose}$ is more recent then its own lock. In that case the local variables $lockedRound$ and $lockedValue$ are reset. After the validator has decided on a value v_i it will then compute and broadcast the *prevote* message (H, R, v_i, σ). Here H and R are again the current block height and round, v_i is the value the validator decided to vote for and lastly σ is the message signature. Note that for efficiency Tendermint only includes a hash of the block B in the prevote and *precommit* messages. We however simplified our pseudocode implementation by including the entire block in the two messages.

Next, all validators including the leader start the *prevote* phase by waiting on the *prevote* messages to arrive. The validators wait for a duration of $\lambda_{timeout}$. Afterwards the validators decide on the value they vote for in the *precommit*, based on the *prevote* messages they delivered. If there is no $2/3$ majority winner based on the *prevotes*, then a validator will vote for the value $v_i = \text{NIL}$ in the *precommit*. Otherwise a validator will first store the votes for the winning value in the local $PoLC$ variable. If the winning vote is a block B then the validator decides to vote for B in the *precommit*. In addition it creates a lock for this value by setting $lockedValue = B$ and $lockedRound = R$. If however the winning value was NIL the validator instead decides to vote for NIL in the *precommit*. Additionally it unlocks any existing lock by setting $lockedValue = \text{NIL}$ and $lockedRound = -1$. Once a validator has decided on a value it then goes forward with computing and broadcasting the *precommit* message (H, R, v_i, σ). The *prevote* and *precommit* messages have the same structure and only are differentiated by their message type.

Finally the validators proceed to the *precommit* phase and wait for the *precommit* messages of other validators to arrive. If during the $\lambda_{timeout}$ waiting time a block B gathers a $2/3$ majority of *precommit* votes, then the validators have reached consensus. In this case they then start the commit step with the block B. If there is no winning value or the winning value is NIL, then the validators start the next consensus round $R + 1$ instead.

On start of commit step The commit step is rather simple. In it the validators add the consensus block to their local chain and then broadcast the block to the rest of the network. Afterwards the next consensus is started by invoking the $newHeight$ step for the next block height $H + 1$.

On propose message delivery When a new *propose* message is delivered the message itself and the proposed block B are validated. If both are valid then the message is stored in the local variable $proposal$.

On prevote message delivery When a new *prevote* message is delivered it is first validated. Once the message has passed validation it is then added to the other *prevotes*. Additionally there is a special event that the validators are looking out for, which concerns the delivery of prevotes from a future round $R + x$. If at any point a validator delivers a $2/3$ majority of *prevotes* from a future round $R + x$, then it skips ahead to the *prevote* phase of that round $R + x$. This event is referred to as a common exit condition in Tendermint and can happen at any point during a consensus round. This event adds a sort of catch-up mechanism, if a validator should ever get out of sync with the other validators. Lastly note that for this common exit condition to trigger the *prevotes* only have to be from the same round, the value does not matter in this case.

On precommit message delivery When a new *precommit* message is delivered, it is first validated. Once the message has passed validation it is then added to the other *precommits*. There is a similar common exit condition here as for the *prevotes*. If a validator ever delivers a $2/3$ majority of *precommits* for a future round $R + x$, then it skips ahead to the *precommit* phase of that round $R + x$. In addition there is a second common exit condition. This one is triggered as soon as a $2/3$ majority for a block B in the current round R is delivered. In that case the validator starts the commit step for block B immediately. This condition can be triggered at any point during the current round R and lets the validator skip the rest of the round.

7.1.4 Protocol attributes

For the chain extension COSMOS uses the Tendermint consensus protocol. The leader selection in Tendermint is a weighted round-robin, that selects exactly one leader for each round of consensus. The leader selection method is similar to a priority queue in its design and does not feature random selection. Since Tendermint is a BFT consensus protocol it finalizes blocks instantly, forks do not occur in COSMOS under normal circumstances. Since the validators and the leaders are both publicly known ahead of time, this could make them potential targets for an adversary. However COSMOS does feature a bounty system for successfully hacking a validator. This leads to voters being discouraged from pooling all their votes on a few candidates and therefore the voting power is spread more evenly among the 100+ validators. Therefore while validators are still potential targets for adversaries, they are less likely to stand out in terms of voting power.

Since Tendermint is a PBFT-based consensus protocol, COSMOS needs more than $2/3$ of the voting power to be in honest hands in order to function correctly. Since a validator's voting power is based on the weighted total of all the votes it received, this more directly translates to the condition that $2/3$ of the bonded stake must be honest. Furthermore COSMOS uses slashing to keep validators and voters honest.

Since the final *precommit* messages of each block are stored on the chain, all parties in the network can verify consensus. However the protocol designers do state that in order for a party to avoid being the victim of a long-range attack, it can not be offline for longer then the unbonding period. Similarly a party does need the help of a trusted party in order to join or rejoin the network.

Algorithm 12 Pseudocode: COSMOS/Tendermint chain extension (party p_i) Part 1

state
- $\mathcal{B}$: local blockchain
- h: latest hash of local chain = $H(head(\mathcal{B}))$
- $\mathbb{V}$: current set of validators
- $unordered$: set of transaction data to be included in the block
- $\mathcal{D}$: list of transactions included in the block
- vk_i, sk_i: signing key pair for the blocks
- $\lambda_{timeout}$: duration a party waits for messages to arrive
- H: block height of the consensus block
- R: the current round of Tendermint protocol
- $lockedValue$: the most recent value the validator has *precommit* to
- $lockedRound$: the round in which $lockedValue$ was set
- $PoLC$: the set of *prevote* messages from round R_{PoLC} with $2/3$ support for value v_{PoLC}
- $lastCommits$: the set of *commit* signatures for previous block
- $proposal$: the *propose* message of the current round
- $preVotes$: the set of *prevote* messages of the current block height H
- $preCommits$: the set of *precommit* messages of the block height current H

//Tendermint Chain extension starts with newHeight step
upon start of newHeight step for new block height H **do**
- **wait**($\lambda_{timeout}$) // wait while gathering remaining *precommit* messages
- $lastCommits \leftarrow$ select from $preCommits$ all *precommits* from the final vote

- $\mathbb{V} \leftarrow$ compute the Top N validators based on local blockchain $\mathcal{B}$
- $R \leftarrow 0$
- $lockedValue \leftarrow$ NIL
- $lockedRound \leftarrow -1$
- $PoLC, proposal, preVotes, preCommits \leftarrow \emptyset$
- **if** $p_i \in \mathbb{V}$ **then**
  - **start** new round with R set to 0 // see Algorithm 13

upon start of commit step for block B and block height H **do**
- $\mathcal{B} \leftarrow \mathcal{B}||B$
- $h \leftarrow H(B)$
- *broadcast*([BLOCK, (B)])
- $H \leftarrow H + 1$
- start newHeight step for next block height H

upon *deliver*([PROPOSE, $(H, R, B, PoLC, \sigma)$]) **do**
- verify the message signature σ over $(H, R, B, PoLC)$
- verify the block signature σ_{block} over $(h, H, lastCommits, \mathcal{D})$
- verify all transactions in $\mathcal{D}$ of block B
- **if** *propose* message is valid and B passed verification **then**
  - $proposal \leftarrow (H, R, B, PoLC, \sigma)$

upon *deliver*([PREVOTE, (H, R, v_i, σ)]) **do**
- verify the message signature σ over (H, R, v_i)
- **if** prevote message is valid **then**
  - $preVotes \leftarrow preVotes \cup \{(H, R, v_i, \sigma)\}$
  - **if** at any point a $2/3$ majority of *prevotes* for the same round $R + x$ is delivered **then**
    - jump to the start of *prevote* phase for round $R + x$

upon *deliver*([PRECOMMIT, (H, R, v_i, σ)]) **do**
- verify the message signature σ over (H, R, v_i)
- **if** *precommit* message is valid **then**
  - $preCommits \leftarrow preCommits \cup \{(H, R, v_i, \sigma)\}$
  - **if** at any point a $2/3$ majority of *precommits* for a value v is delivered **then**
    - start new commit step with block $B = v$
  - **if** at any point a $2/3$ majority of *precommits* for the same round $R + x$ is delivered **then**
    - jump to the start of *precommit* phase for round $R + x$

Algorithm 13 Pseudocode: COSMOS/Tendermint chain extension (party p_i) Part 2

upon start of a new round R for the current block height $height$ **do**
  if p_i is the leader for the round r **then**
    if p_i is still locked **then**
      $B \leftarrow lockedValue$
    else
      $\mathcal{D} \leftarrow$ select maximum valid transactions from $unordered$
      $B \leftarrow (h, H, lastCommits, \mathcal{D}, \sigma block)$
    $\sigma \leftarrow Sign_{sk_i}(H, R, B, PoLC)$
    $proposal \leftarrow (H, R, B, PoLC, \sigma)$
    broadcast([PROPOSE, $(H, R, B, PoLC, \sigma)$])
  else
    wait($\lambda_{timeout}$) // wait for *propose* message to arrive
    if $proposal$ is set **then**
      if $PoLC_{proposal}$ unlocks p_i **then**
        $lockedValue \leftarrow nil$
        $lockedRound \leftarrow -1$
      if p_i is still locked **then**
        $v_i \leftarrow lockedValue$
      else
        $v_i \leftarrow B$
    else // timeout
      $v_i \leftarrow$ NIL
    $\sigma \leftarrow Sign_{sk_i}(H, R, v_i)$
    broadcast([PREVOTE, (H, R, v_i, σ)])

  wait($\lambda_{timeout}$) // wait while gathering *prevote* messages
  if there is a $2/3$ majority of *prevotes* for a value v in round R **then**
    // start of *prevote* phase
    $PoLC \leftarrow$ select all *prevotes* for value v in round R from $preVotes$
    if value v is a block B **then**
      $v_i \leftarrow B$
      $lockedValue \leftarrow v_i$
      $lockedRound \leftarrow R$
    else // v is NIL
      $v_i \leftarrow$ NIL
      $lockedValue \leftarrow$ NIL
      $lockedRound \leftarrow -1$
  else // timeout
    $v_i \leftarrow$ NIL
  $\sigma \leftarrow Sign_{sk_i}(H, R, v_i)$
  broadcast([PRECOMMIT, (H, R, v_i, σ)])

  wait($\lambda_{timeout}$) // wait while gathering *precommit* messages
  if there is a $2/3$ majority of *precommits* for a value v and round R **then**
    // start of *precommit* phase
    if value v is a block B **then**
      start new *commit* step with block $B = v$
    else // v is NIL
      start new round with for $R + 1$
  else // timeout
    start new round with for $R + 1$

8 Comparison/Overview

In this section we compare all the stake-based consensus protocols we discussed in this thesis. We give a summary of that comparison in Table 8.1 and will highlight some observations during this section.

Algorand Like the majority of the protocols discussed in this thesis, Algorand relies on BFT as a core building block in its chain-extension protocol. However Algorand is unique in that it uses a form of binary BFT rather then PBFT. Furthermore all parties in the network participate in the binary BFT but only a different sub-group of them become active during each round. The active participants for each round of the binary BFT are determined through weighted random selection. This allows them to stay anonymous until they cast their vote. As a result Algorand is a somewhat permissionless BFT consensus protocol.

Cardano When we look at Table 8.1, we see that Cardano stands out compared to the other protocols. It is the only protocol that does not feature BFT in any way. Instead Cardano's design seems much more like a direct translation of a proof-of-work protocol. Similar to proof-of-work protocols, Cardano only needs more then $1/2$ of the stake to be honest. Additionally the fact that a block in Cardano is final after it is k blocks deep in the chain is similar to the finalization criteria in proof-of-work protocols as well.

Ethereum 2.0 Ethereum 2.0 is one of the four protocols that uses a variant of PBFT. However instead of applying PBFT directly to determine the next block, Ethereum 2.0 uses it only for checkpoint finalization. Ethereum 2.0 also spans what would be considered a round in PBFT over a period of two epochs. While this does increase the time to finalize a block it also grants the opportunity to have more validators participate in the consensus. This large number of validators decreases the likelihood of an adversary controlling more than $1/3$ of them. In addition the inclusion of slashing should further strengthen that effect.

Neo Neo's dBFT 2.0 is the variant that departures the least from PBFT. Through voting it determines the group of validators that take part in the consensus. The consensus protocol itself fairly close in its construction to PBFT. However it is worth mentioning that Neo reduced the number of validators from 21 down to 7 in its newest version N3. This means that Neo can only tolerate a maximum of 2 adversarial

	Permissioned	Synchronization	Honest stake	Finalization	BFT based
Algorand	No	partially	$n \geq 2f + 1$	instant	Yes
Cardano	No	global clock	$n \geq f + 1$	k blocks	No
Ethereum 2.0	Yes	partially	$n \geq 2f + 1$	3 epochs	Yes
Neo	Yes	partially	$n \geq 2f + 1$(*)	instant	Yes
EOSIO	Yes	partially	$n \geq 2f + 1$(*)	2 epochs	Yes
COSMOS	Yes	partially	$n \geq 2f + 1$	instant	Yes

(*) Number of validators not stake

Table 8.1: Comparison between all protocols.

validators. Furthermore there is no information on how much honest voting power is required to prevent that more then $1/3$ of the elected validators are adversarial.

EOSIO EOSIO's use of their PBFT-variant "pipelined BFT" is very similar to that of Ethereum. The main purpose of "pipelined BFT" is again the finalization of blocks. It should be noted though that in EOSIO there are only 21 validators involved in consensus, which is considerably less than in Ethereum 2.0. While we saw hints of slashing mechanisms during our research there are no mentions of slashing in the documentation. Finally, as with Neo there is no mention of how adversarial voters may influence the election.

COSMOS COSMOS handles the election process differently from Neo and EOSIO. There are two major differences in its design. The first is that rather then all elected validators having the same voting power, a validator's voting power corresponds to the total weight of the received votes. The second is that voters are held responsible for the actions of their candidate. Therefore voting for trustworthy candidates is more important in COSMOS. These two design choices help to draw a much clearer connection of how the required $2/3$ honest majority for the consensus translates to the required honesty of the voters.

9 Conclusion

In this thesis we provided an overview over some of the most prominent stake-based consensus protocols. Furthermore we provide descriptions and pseudocode with common terminology, notations and layout for each of those protocols. We hope that our work serves as a basis for further analysis of stake-based consensus protocols.

Of course, there are still some prominent protocols, such as Solana or Nxt, that were not included in this thesis due to time constraints. Therefore one way to build upon our work would be to include further protocols. In particular it would be interesting to see if there are other hybrid combinations that we have not encountered during our research.

Another way to extend our work would be to do a deeper analysis for protocols that do not provide a scientific paper, such as Neo or EOSIO. This would clear up some uncertainties we encountered due to the lack of information. This way we would be able to provide a more accurate description of those protocols.

Bibliography

[1] Cosmos - validators FAQ. https://github.com/cosmos/cosmos/blob/master/VALIDATORS_FAQ.md, 2018. Accessed: 20-02-2022.

[2] EOS block producer voting guide. https://medium.com/coinmonks/eos-block-producer-voting-guide-fba3a5a6efe0, 2018. Accessed: 11-02-2022.

[3] Fix DPOS loss of consensus due to conflicting last irreversible block. https://github.com/EOSIO/eos/issues/2718, 2018. Accessed: 11-02-2022.

[4] Cosmos - whitepaper. https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md, 2019. Accessed: 20-02-2022.

[5] EOSIO documentation - v2.1 consensus protocol. https://developers.eos.io/welcome/latest/protocol-guides/consensus_protocol, 2020. Accessed: 11-02-2022.

[6] Neo v2 source code on github (stable). https://github.com/neo-project/neo/tree/master-2.x/neo, 2021. Accessed: 03-02-2022.

[7] Neo v3 documentation. https://docs.neo.org/docs/en-us/index.html, 2021. Accessed: 03-02-2022.

[8] Neo v3 documentation: Consensus protocol. https://docs.neo.org/docs/en-us/basic/consensus/consensus_protocol.html, 2021. Accessed: 03-02-2022.

[9] Neo v3 documentation: dbft 2.0 algorithm. https://docs.neo.org/docs/en-us/basic/consensus/consensus_algorithm.html, 2021. Accessed: 03-02-2022.

[10] Tendermint - byzantine consensus algorithm. https://docs.tendermint.com/master/spec/consensus/consensus.html, 2021. Accessed: 20-02-2022.

[11] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[12] Vitalik Buterin. Vitalik's annotated ethereum 2.0 spec. https://github.com/ethereum/annotated-spec/blob/master/phase0/beacon-chain.md, 2020. Accessed: 11-01-2022.

[13] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[14] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. Combining GHOST and casper. *CoRR*, abs/2003.03052, 2020.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[16] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[17] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, page 454, 2017.

[19] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[20] Daniel Larimer. DPOS BFT— pipelined byzantine fault tolerance. `https://eos.io/news/dpos-bft-pipelined-byzantine-fault-tolerance/`, 2018. Accessed: 11-02-2022.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. `http://bitcoin.org/bitcoin.pdf`.

[22] Qin Wang, Jiangshan Yu, Zhiniang Peng, Van Cuong Bui, Shiping Chen, Yong Ding, and Yang Xiang. Security analysis on dbft protocol of NEO. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 20–31. Springer, 2020.

Erklärung

gemäss Art. 30 RSL Phil.-nat. 18

Name/Vorname: Bürk Timo

Matrikelnummer: 06-920-961

Studiengang: Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Titel der Arbeit: Blockchain consensus protocols based on stake

LeiterIn der Arbeit: Prof. Dr. Christian Cachin

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum Bern, 13.04.2022

Unterschrift