Kamilla Stevenson and Oda Skoglund

Design of Novel Energy-Efficient and Privacy-Preserving Blockchain Consensus Mechanisms

Master's thesis in Communication Technology and Digital Security
Supervisor: Danilo Gligoroski
Co-supervisor: Mayank Raikwar
June 2021

Kamilla Stevenson and Oda Skoglund

Design of Novel Energy-Efficient and Privacy-Preserving Blockchain Consensus Mechanisms

Master’s thesis in Communication Technology and Digital Security
Supervisor: Danilo Gligoroski
Co-supervisor: Mayank Raikwar
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title:	Design of Novel Energy-Efficient and Privacy-Preserving Blockchain Consensus Mechanisms
Student:	Kamilla Stevenson, Oda Skoglund

Problem description:

Proof of Work (PoW), a popular consensus mechanism used in blockchains, such as Bitcoin, has led to vast energy consumption by compelling users to solve computationally hard problems. The solution serves as the proof that allows users to propose a block to extend the blockchain. As an alternative to PoW, Proof of Stake (PoS) was proposed to solve this problem by selection based on stake. While PoS offers an alternative to the energy wastefulness of PoW, it is not able to offer the same level of privacy since user identity is an integral part of the proof.

Is it possible to design consensus mechanisms that are as energy-efficient as PoS and as privacy-preserving as PoW?

Formal studies into the feasibility of privacy in PoS yield promising proposals for Private Proof of Stake (PPoS). Upon initial analysis, some issues in the PPoS proposals have come to light. To clearly identify such issues, a thorough analysis of the proposals will be completed with regards to privacy, efficiency, and inclusion of all aspects relevant to distributed consensus. In response to the analysis, an investigation into methods that may prove beneficial to achieve privacy and maintain efficiency in PoS will be conducted. Finally, improvements to previous PPoS proposals and/or an entirely new PPoS proposal will be presented and evaluated.

Date approved:	2021-02-10
Supervisor:	Danilo Gligoroski , IIK
Co-supervisor:	Mayank Raikwar, IIK

Abstract

Consensus in Blockchains, commonly achieved by wasteful Proof of Work (PoW) protocols, is subject to re-evaluation as emerging Proof of Stake (PoS) protocols pose a viable alternative. However, these PoS protocols, although viable, cannot compete with the privacy offered by PoW since PoS inherently reveal both the identity and stake of the stakeholders.

We conducted a literature study of the four currently available PPoS proposals with a record of scientific publication: *Proof-of-Stake Protocols for Privacy-Aware Blockchains* by Ganesh et al., *Anonymous Lottery In the Proof-of-Stake setting* by Baldimtsi et al., *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* by Kerber et al., and *Zether: Towards Privacy in a Smart Contract World* by Bünz et al. An analysis focusing on privacy and performance identified issues with the PPoS proposals.

Of the identified issues, we chose to improve upon the performance of Baldimsti et al.'s proposal through the following novel proposals.

1. A PPoS that applies homomorphic encryption to remove the need to send multiple unlinkable proofs in the multi-stake setting.
2. A scheme with a trade-off between stake privacy and performance, offering an alternative method of tackling privacy in the multi-stake setting.

As a proof of concept, both proposals are instantiated with the PoS protocol Algorand. For this purpose, a literature study and analysis of Algorand were conducted, with a specific focus on the privacy of identity and stake.

The proposals are evaluated in terms of performance, privacy, and security. In terms of performance, Baldimsti et al.'s proposal has linear complexity, Proposal 1 has constant complexity, and Proposal 2's performance is determined by the stakeholder and comes at the cost of privacy. While Baldimsti et al.'s proposal and Proposal 1 keep the identity and stake of stakeholders private, Proposal 2 permits stakeholders with desires for varying degrees of privacy to interact on the same blockchain. We specify paths for future work within both our proposals and present further ideas for improvements to performance.

Sammendrag

Konsensus i Blockchains, ofte oppnådd med energikrevende Proof of Work (PoW) protokoller, er gjenstand for re-evaluering, som en følge av at nye Proof of Stake (PoS) protokoller vokser frem som gode alternativ. Disse PoS-protokollene kan likevel ikke konkurrere med personvernet som tilbys av PoW, siden de iboende avslører både identiteten og stake til interessentene.

Vi gjennomførte en litteraturstudie av alle tilgjengelige PPoS forslagene med vitenskapelig publisering: *Proof-of-Stake Protocols for Privacy-Aware Blockchains* av Ganesh et al., *Anonymous LotteryIn the Proof-of-Stake setting* av Baldimtsi et al., Ouroboros Crypsinous: *Privacy-Preserving Proof-of-Stake* av Kerber et al., and *Zether: Towards Privacy in a Smart Contract World* av Bünz et al. En analyse med fokus på personvern og ytelse identifiserte utfordringer med PPoS-forslagene.

Av de identifiserte utfordringene valgte vi å forbedre ytelsen til Baldimsti et al. sitt forslag gjennom de følgende nye forslagene.

1. En PPoS som bruker homomorf kryptering for å fjerne behovet for å sende flere (ikke-linkbare) bevis i multi-stake-setting.
2. En ordning med en trade-off mellom personvern av stake og ytelse, som tilbyr en alternativ metode for å takle personvern i multi-stake setting.

Som et bevis på konseptet instansierer vi begge forslagene med PoS-protokollen Algorand. Av denne grunn gir vi en litteraturstudie og analyse av Algorand med et spesielt fokus på personvernet til identitet og stake.

Forslagene vurderes basert på ytelse, personvern og sikkerhet. Når det gjelder ytelse, har Baldimsti et al. sitt forslag lineær kompleksitet, forslag 1 har konstant kompleksitet, og forslag 2 sin ytelse bestemmes av interessenten og kommer på bekostning personvern. Mens Baldimsti et al. sitt forslag og forslag 1 holder interessentenes identitet og stake private, tillater forslag 2 interessenter som ønsker varierende grad av personvern å samhandle på samme blockchain. Vi spesifiserer områder for videre arbeid i begge forslagene og presenterer videre ideer for forbedringer av ytelsen.

Preface

This thesis marks the culmination of our master's degree in *Communication Technology and Digital Security* under the *Department of Information Security and Communication Technology*, *Faculty of Information Technology and Electrical Engineering*, at *The Norwegian University of Science and Technology (NTNU)*. A special thanks goes out to our supervisor, Mayank Raikwar, and responsible professor, Danilo Gligoroski, for their excellent guidance and support throughout the work on the thesis and pre-project. We would also like to thank our families for the love, support, and encouragement.

Contents

Appendices

List of Figures

List of Tables

List of Algorithms

List of Symbols

rt_{pk}	The root of the Merkle tree, $MTree(pk)$, which stores the public keys.
λ	Security parameter.
π_i	Algorand, [5], and Proposal 1: Denotes party P_i's proof of selection. Proposal 2: Denotes the list of party P_i's multiple proofs of selection.
π_{NIZK}	Zero-knowledge proof of selection.
b_{tag}	A value used to denote selection in the single-stake setting in [5]. $b_{tag} = 1$ if the party is selected and $b_{tag} = 0$ if not.
$block$	As its name suggests, denotes a block in the blockchain.
C_i^v	Party P_i's commitment to v_i.
cm_i	Party P_i's commitment to $stake_i$.
$counts$	A hash table containing the current total votes for each block.
ctx	A context that captures the current state of the blockchain.
$ctx.last\_block$	The hash of the "newest" block / previous block added in the blockhain.
$FHE.pk_i$	The public FHE key; the homomorphic encryption key for party P_i.
$FHE.sk_i$	The secret FHE key; the homomorphic decryption key for party P_i.

hash	The same as *sorthash*.
index	[5]: denotes each unit of wt_i. Proposal 1: denotes each unit of j_i. Proposal 2: denotes the index in the *votingpowers* list.
j_i	Total voting power for party P_i.
j'_i	The homomorphic encryption of the total voting power for party P_i.
m_i	The message gossiped by party P_i, containing i.a. msg_i.
msg_i	The message composed by party P_i gossiped in m_i.
P_i	Party i.
params	Global parameters used for homomorphic encryption.
pk_i	The public key of party P_i.
privacy	A variable between 1-10 chosen by the party itself. 10 denotes a high privacy and 1 a low privacy.
privacy_level	Calculated by *CominationFunc()* based on the party's chosen *privacy* variable and j_i. A high value denotes good privacy and a low value denotes worse privacy.
role	Distinguishes the different roles that a party may be selected for in Algorand, e.g., block proposer or verifier.
round	Algorand grows the blockchain in rounds. For each round a block is added to the blockchain.
rt_{cm}	The root of the Merkle Tree, $MTree(cm)$, which stores the commitments to stake.
$rt_{\vec{V}_{tag}}$	The root of the Merkle tree, $MTree(\vec{V}_{tag})$, which stores $\vec{V}_{tag}$.

$seed$	A publicly known random seed.
sk_i	The secret key of party P_i.
$sorthash$	The hash returned from the VRF in $Sortition()$ in Algorand.
$stake_i$	Party P_i's stake.
$step$	Algorand grows the blockchain in rounds. Each round in consists of several steps.
tag	Defines when a new selection process is performed.
$totalStake$	The total stake of the parties in the network.
v_i	Party P_i's inverse trapdoor permutation of $\vec{V_{tag}}[i]$ computed using the party's trapdoor secret key.
$\vec{V}_{tag}$	A public vector for tag, consisting of n uniformly random elements, where n is the total number of parties. Stored as a Merkle tree $MTree(\vec{V}_{tag})$ with root $rt_{\vec{V}_{tag}}$.
$\vec{V}_{tag}[i]$	Party P_i's element at position i in the vector $\vec{V}_{tag}$.
$value$	The hash value of the block.
$votes$	The votes, equal to the voting power, associated with a message and proof.
$votes'$	The value $votes$ homomorphically encrypted.
$votingpowers$	The list of a party's distributed voting powers.
W	The weight of all parties in the system.
w	The zero-knowledge proof witness.
wt_i	Party P_i's weight, proportional to their stake in the network.
x	The zero-knowledge proof statement.

List of Acronyms

ABC Anonymous Broadcast Channel.

aSVC Aggregatable Subvector Commitment.

AVRF Anonymous Verifiable Random Function.

BA Byzantine Agreement.

BGV Brakerski-Gentry-Vaikuntanathan.

CRS Common Reference String.

FHE Fully Homomorphic Encryption.

NIZK Non-Interactive Zero-Knowledge.

NTNU Norwegian University of Science and Technology.

PoS Proof of Stake.

PoW Proof of Work.

PPoS Private Proof of Stake.

PPT Probabilistic Polynomial-Time.

PRF Pseudorandom Function.

R1CS Rank-1 Constraint Satisfaction.

SNARK Succinct Non-interactive Argument of Knowledge.

VRF Verifiable Random Function.

zk-SNARK zero-knowledge Succinct Non-interactive Argument of Knowledge.

Chapter 1

Introduction

The growth and adoption of cryptocurrencies and, consequently, blockchain technologies have been rapid over the past decade. This is evident in the progression from a non-existing cryptocurrency market in 2008 to a market consisting of 5464 cryptocurrencies with a total financial market worth of 1.76 trillion, as of the 3rd of June 2021 [17]. Cryptocurrencies, such as Bitcoin, rely on the existence of distributed ledgers, commonly known as blockchains. Such blockchains depend on consensus mechanisms to extend the blockchain, i.e., add a block to the chain. At the heart of consensus mechanisms lies selection, the process by which parties determine who in the network gets to decide the next block in the blockchain.

To date, PoW, with a market cap dominance of around 65% (as of the 3rd of June 2021), remains the most dominant consensus mechanism implemented in blockchains since the creation of Bitcoin in 2009 [56, 17]. At its core, PoW selection prompts parties to solve a computationally hard puzzle whose solution is the proof of selection. Unfortunately, the quest for solutions to such puzzles requires ever-increasing computation power resulting in enormous energy consumption [24, 47].

As a solution to the energy wastage issue, PoS was introduced informally in an online Bitcoin forum in 2011 [1]. PoS implements a selection based on stake, i.e., the proportion of money a party holds, to determine who can extend the blockchain. The identity of the selected party is an integral part of the proof of selection; thus, an issue of privacy arises as the proof reveals the identity of the selected. Adversaries can also deduct the stake of the parties by a frequency analysis of how often parties are selected. In order to offer a competitive alternative to PoW, PoS must be privacy-preserving as well. Hence, our research question is as follows,

Is it possible to design consensus mechanisms that are as energy-efficient as PoS and as privacy-preserving as PoW?

In an increasingly environmentally conscious world, the future of PoW based cryptocurrencies is becoming ever more unstable as PoS offers a “greener” alternative [24]. For example, in May 2021, Tesla took a stance on the issue by retracting the promise that their cars can be bought with Bitcoin, citing high environmental cost as the reason [35]. This stance caused Bitcoin to fall 37.5% in May 2021 [30] and had comparable effects on other PoW based cryptocurrencies [35]. It is worth noting that Bitcoin prides itself on the fact that 74% of BTC are mined with renewable energy [7]. On the other hand, it is still a simple fact that this energy could serve another purpose if Bitcoin mining facilities were not using it. Nevertheless, we want to emphasize the importance of the efforts made by PoW based cryptocurrencies to use renewable energy which may help promote both energy reallocation and the development of renewable energy [33].

Formal studies into PPoS have emerged within the scientific community as recently as 2019 [25, 36] and 2020 [5, 12]. These studies aim to preserve the efficiency of PoS without revealing the identity and stake of the selected. Note that a multitude of privacy-preserving PoS proposals can be found on, for example, Github [20, 2]. However, we limit the scope of our thesis to those promoting reviews from the cryptographic community with a record of scientific publication. The four available PPoS proposals within this scope are *Proof-of-Stake Protocols for Privacy-Aware Blockchains* [25], *Anonymous Lottery In the Proof-of-Stake setting* [5], *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* [36], and *Zether: Towards Privacy in a Smart Contract World* [12]. A preliminary review in the pre-project of this thesis [52], uncovered a few possible issues and we hypothesize that a more thorough review will reveal additional issues or weaknesses of the proposals.

In summary, our overall goal is to design novel (PPoS) consensus mechanisms that are as energy-efficient as PoS and as privacy-preserving as PoW. To reach our objective, we will review and analyze all currently available PPoS consensus mechanisms and, based on identified issues, make proposals for modifications that improve these mechanisms. Note that we limit our scope to consensus and, in doing so, refrain from looking at the anonymization of transactions. Various implementations of anonymous transactions already exist, such as Zcash [50] and Monero [46]. We further restrict the scope of the thesis with the following assumptions; the network structure is synchronous, there is only one round of selection, and the majority of parties are honest.

1.1 Background

The topics necessary for understanding the thesis are Blockchain, PoW, and PoS.

1.1.1 Blockchain

The alias Satoshi Nakamoto conceptualized, in 2008, the first blockchain database in his novel white paper titled *Bitcoin: A Peer-To-Peer Electronic Cash System* [43]. In the following year the Bitcoin network commenced with blockchain as a core component [3]. Since then, the number of cryptocurrencies has increased tremendously, with Bitcoin remaining the most popular [17].

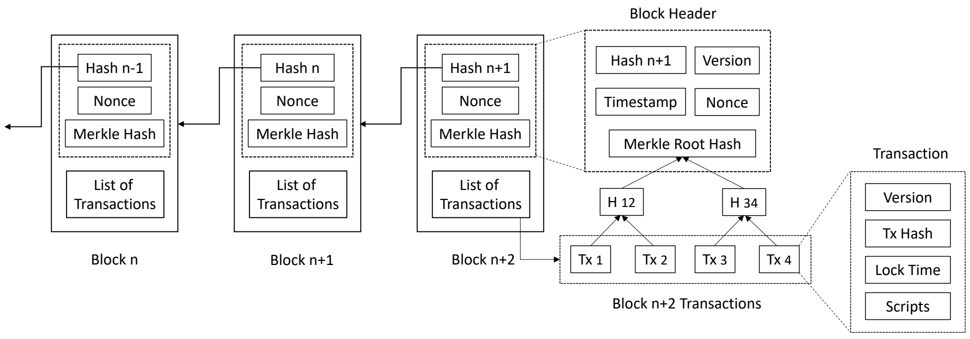

Figure 1.1: An Illustration of Blockchain. The figure illustrates that a blockchain is a chain of blocks composed of transactions. Each block references the hash of the proceeding block (source: [47]).

By definition, "a blockchain is a distributed, decentralized and immutable ledger that stores transaction history" [23]. To elaborate, a blockchain is a distributed ledger across a peer-to-peer network, or rather a database [48] shared and synchronized across numerous locations, geographically spread across countries, organizations and sites. A blockchain is decentralized, as no central authority presides over the blockchain, i.e., no individual or group manages and controls access to the blockchain. The previously mentioned transaction history is continuously appended to the ledger and is accessible by any party on the network. Essentially, as seen in Figure 1.1, a blockchain consists of blocks, each identified by a hash, sequentially connected by referencing the hash of the preceding block. This combination of hashing and referencing is advantageous as it makes the chain immutable. In other words, it is nearly impossible for any party to tamper with a block without invalidating all succeeding blocks. Apart from the use of a hash function, blockchain employs different cryptographic concepts to ensure the security and privacy of the blockchain.

To clarify, as detailed by [3], the main components and core concepts of a blockchain are as follows:

- *Transactions*: the data, created, signed, and finally broadcast by participating nodes to the rest of the network;
- *Blocks*: the collection of transactions into groups. These blocks have set storage capacities. Once reached, a block is validated and appended to the blockchain;
- *Blockchain*: a ledger of all created blocks within the network, linked or "chained" together using hash codes. That is, each block contains the hash of the preceding block;
- *Consensus mechanism*: a mechanism for selecting which blocks to append to the blockchain. Subsections 1.1.2 and 1.1.3 thoroughly cover two such mechanisms.

According to Raikwar et al. [47], blockchains can be classified depending on the implementation design, administration rules, data availability, and access privileges. In academic literature, blockchains have been classified as "public" and "private", while from the administrative point of view, they are described as "permissioned" and "permissionless". In total, the literature recognizes four types of blockchains (Table 3 in [47]) by coupling public, private, permissioned, and permissionless properties. The permissionless platforms are accessible by any party, as no permissions are required to join the network and submit transactions. Note that this thesis will only concern itself with permissionless blockchains. A major challenge for permissionless blockchains lies in achieving consensus on a block to extend the blockchain and several viable consensus mechanisms exist.

1.1.2 Proof of Work

PoW is a mechanism used to provide consensus on the validity of each new block applied to the blockchain. The mechanism dates back to 1992 when Dwork and Naor [22] introduced the concept as a technique to combat junk mail. When Nakamoto presented Bitcoin in 2008 [43], PoW was popularly adapted as a consensus mechanism for blockchains [34].

As explained in [3], PoW relies on the assumption that the majority of the computing power belongs to honest parties in the network. As illustrated in Figure 1.2, some of the parties in the network, miners, compete to solve a complex puzzle. When a miner solves this puzzle, like Alice in Figure 1.2, the miner broadcasts their chosen block to the network together with the solution as the hash of the block. As the solution is easy to verify, all parties can then judge the validity of the block by verifying that the solution is correct. The miner who has published a block with a valid solution receives a reward to motivate the parties to contribute as miners.

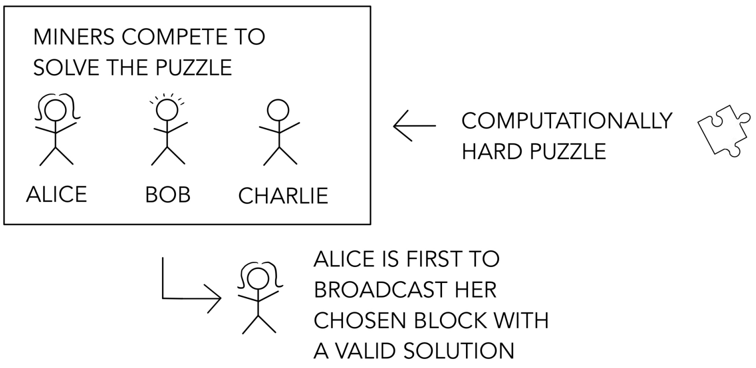
MINERS COMPETE TO SOLVE THE PUZZLE
ALICE
BOB
CHARLIE
COMPUTATIONALLY HARD PUZZLE
ALICE IS FIRST TO BROADCAST HER CHOSEN BLOCK WITH A VALID SOLUTION

Figure 1.2: A Simplified Illustration of Selection in PoW. The parties compete to solve a computationally hard puzzle. Alice, the first to solve the puzzle, broadcast her chosen block with the solution.

Since a lot of computing power is required to solve the puzzle, utilization of PoW has led to enormous energy consumption. de Vries [21] estimated that the lower bound of Bitcoins energy consumption was around 2.55 gigawatts when published in 2018 and predicted a high increase in the future. The computational power needed to solve the puzzles has also led miners to collaborate in establishing mining pools. According to Cong et al., this "severely escalate the arms race in PoW blockchains, whose real consequence is an enormous additional amount of energy devoted to mining" [18].

Another issue with PoW consensus mechanisms is the possibility of persistent forks. Forks arise when parties in the network disagree on the history of blocks in the blockchain [49]. A paper by Saleh [49] explains that persistent forks occur when miners are incentivized to maintain different versions of the blockchain. The miners want to maintain the version of the blockchain in which they have validated a block, because of the reward they will receive if this blockchain persists. Two mitigation strategies used to avoid forks increase the time needed to confirm a block. The first sets a lower bound for the computational difficulty of the puzzle to require a sufficiently long time to add one block [29]. The second waits to confirm a block until several more blocks have been added to ensure high confidence that the block was added to the "correct" blockchain. In Bitcoin, ten minutes are required to grow the blockchain by one block, and six blocks are recommended as an appropriate waiting time to ensure high confidence, resulting in approximately an hour to confirm a block. As such, fork mitigation in PoW mechanisms significantly increases the time needed to confirm a new block.

1.1.3 Proof of Stake

The history of PoS began in 2011 when the idea was first introduced in a Bitcoin forum [1]. More formally, King and Nadal [38] provided the first PoS proposal in 2012 as an alternative to PoW. In the following years, the number of proposed PoS protocols has increased rapidly. The PoS based blockchain protocols and cryptocurrencies such as Ouroboros [37], Algorand [29], Cardano [13], and Tezos [31] have emerged. Ethereum, the second-largest cryptocurrency to date lies at the forefront of the transition from PoW to PoS [24].

In contrast to PoW, PoS is based on the assumption that trustworthy parties own a majority of the stake in the network. Therefore, the security of the blockchain is maintained by having stakeholders propose and validate new blocks. The stakeholders have a strong incentive to validate blocks correctly, since their stake would become worthless if the security and, thus, trust in the system vanishes, [8].

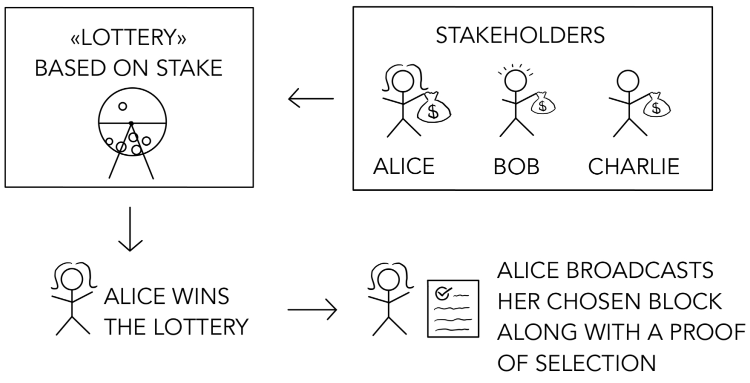

Figure 1.3: A Simplified Illustration of Selection in PoS. A lottery is used to illustrate PoS selection. The stakeholders win with a probability proportional to their stake. Alice wins and, thus, she gets to decide the next block added to the blockchain.

To simplify, for each round, stakeholder(s) are chosen, with a probability equal to the fraction of stake they hold in the network, to validate or propose the next block before it is added to the blockchain [25]. Figure 1.3 illustrates the PoS selection as a lottery. In the figure, Alice wins the lottery and broadcasts her chosen block along with a proof of selection (winning). According to Baldimtsi et al., the selection function must fulfill the properties "privately evaluated, publicly verifiable and fair" [5]. In other words, the selection must be fair and parties in the network must be able to evaluate whether they have been chosen as a proposer or validator locally. All parties must then be able to confirm the validity of the validated or proposed block. Other parties use the selected's public key to assess the validity of the received block or message, consequently revealing the identity of the selected to all parties.

PoS mechanisms can either be single-stake or multi-stake. Baldimtsi et al. [5] propose a PPoS for both the single-stake and multi-stake setting and explain that each public key can be associated with several units of stake in the multi-stake setting. In contrast, "each public key is associated with one unit of stake" [5] in the single-stake setting. Thus, for the single-stake setting, stakeholders must hold a public key for each unit of stake they own, while in the multi-stake setting, only one public key per stakeholder is required. Note that for the scope of this thesis, we look at the multi-stake setting.

Slot-based and committee-based are two types of PoS protocols. Ganesh et al. [25] explain that being selected in slot-based PoS means being able to create a new block for a slot, while in committee-based PoS, being selected means being part of a committee. In contrast to slot-based PoS, selection in committee-based PoS can encompass different roles, e.g., proposing a new block or voting on a proposed block.

PoS mechanisms either support the static stake setting or the dynamic stake setting. As explained in [25], while the distribution of stake is permanent after initialization for the static stake setting, the parties' stake can change with time, and new parties can join the network for the dynamic stake setting. For the scope of this thesis, we look at the static-stake setting.

Most cryptocurrencies implementing PoS provide the parties with an economic incentive to become validators as they receive a reward after validating a block. Saleh [49] explains that the reward must be significantly lower than the stake the validators hold in the network, driving the validators to validate the block correctly, giving way to trust and consensus. The low block reward in PoS mechanisms avoids PoW's issue of persistent forks, mentioned in Section 1.1.2. While the validators in PoW do not have to be stakeholders and have no incentive to resolve forks, the validators in PoS hold stake in the network and, consequently, have the incentive to resolve forks. As a result, PoS also avoids the fork mitigation consequence of a long block confirmation time.

In summary, PoS consensus mechanisms provide a solution to the vast energy consumption of PoW based blockchains by using stake rather than computing power to "win the lottery". However, by making the identity and stake an integral part of the proof, the identity, e.g., the public key of the validator, must be revealed for the proof of selection to be publicly verifiable.

1.2 Contributions

We now provide an overview of our contributions.

A literature study of the currently available PPoSs. I.e., a study of all available PPoS proposals with a record of scientific publication: *Proof-of-Stake Protocols for Privacy-Aware Blockchains* [25] by Ganesh et al., *Anonymous Lottery In the Proof-of-Stake setting* [5] by Baldimtsi et al., *OuroborosCrypsinous: Privacy-Preserving Proof-of-Stake* [36] by Kerber et al., and *Zether: Towards Privacy in a Smart Contract World* [12] by Bünz et al.

An analysis of the aforementioned PPoSs with a specific focus on privacy and efficiency. The analysis identifies issues and areas of improvement within the PPoSs. Note that we exclude Zether [12] as the existing literature is not extensive enough to warrant analysis.

A literature review and analysis of Algorand [29] with a particular focus on privacy. Namely, an analysis of where the PoS protocol reveals identity and stake.

A proposal for improvements to Baldimsti et al.'s PPoS proposal [5]. We effectively remove the need for multiple unlinkable proofs in the multi-stake setting through the use of homomorphic encryption.

The paper *Efficient Novel Privacy Preserving PoS Protocol* attached in Appendix A.1. The paper presents the above proposal and has been accepted for publication to the Blockchain and Internet of Things Conference (BIOTC). The final version of the paper will be published in the International Conference Proceedings by ACM.

A proposal for modifications to Baldimsti et al.'s PPoS proposal [5] to create a privacy performance trade-off in the multi-stake setting. This trade-off scheme allows for a flexible implementation of privacy in PoS, where stakeholders decide how private they want to keep their stake.

An instantiation of both proposals with the PoS based cryptocurrency Algorand [29].

An evaluation of both proposals with regards to performance and privacy.

Some ideas for future work came to mind over the course of this thesis. While we went with the two ideas exemplified by our proposals, we identified two further directions for research presented as future work.

1.3 Methodology

The methodology for the thesis is as follows.

1.3.1 Literature Study

An initial literature study was conducted with regards to the topics of PoW, PoS, and PPoS, as presented in Section 1.1. Next, we completed a literature study of the four currently available PPoS proposals with a record of scientific publication: *Proof-of-Stake Protocols for Privacy-Aware Blockchains* [25], *Anonymous Lottery In the Proof-of-Stake setting* [5], *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* [36] and *Zether: Towards Privacy in a Smart Contract World* [12]. These proposals are presented in Chapter 2. As this is a theoretical thesis, literature studies were performed continuously throughout the semester.

1.3.2 Mathematical Background

Mathematical background on the topics of Succinct Non-interactive Arguments of Knowledge (SNARKs) and Homomorphic Encryption, necessary to understand the thesis, are presented in Chapter 3. Note that the study of these topics was conducted when appropriate; while SNARK were studied at the beginning of our work, homomorphic encryption was studied after the idea for our first proposal emerged.

1.3.3 Analysis of PPoSs

Analyses of the PPoS proposals [25], [5], and [36] were conducted. The proposals were analyzed, focusing on identifying issues in the proposals with regards to privacy, performance, and the inclusion of all aspects relevant to distributed consensus. These analyses are presented in Chapter 4. Note that [12] was not analyzed due to a lack of details and formalization. Following the analyses, a discussion on which PPoS proposal to improve upon was completed. This discussion and final decision on [5], presented in Section 4.3, were based on the issues identified in the analyses.

1.3.4 Analysis of Algorand

Baldimtsi et al.'s proposal [5] defines an ideal functionality and applies it to the selection function of the PoS protocol Algorand [29]. Thus, we decided to keep Algorand as the underlying PoS protocol for our future PPoS proposals. First, the PoS protocol Algorand was studied. Following the study, an analysis of Algorand was completed. The analysis focused on identifying what parts of Algorand expose the participants' identity or stake, and thus, need to be modified to make Algorand private. Our findings are presented in Section 4.4.

1.3.5 Proposals for improvement

Two new PPoS proposals built upon Baldimtsi et al.'s proposal [5] with Algorand [29] as the underlying PoS protocol are presented in Chapter 5. We focused on improvements to [5]'s efficiency challenges identified in its analysis, in addition to the privacy issues identified in the analysis of Algorand. The modifications made to [5]'s protocols and Algorand's procedures are detailed for both proposals. Section 5.1 presents our first PPoS proposal. This proposal uses homomorphic encryption to improve the performance of [5] while still achieving complete privacy. Section 5.2 presents our second proposal which provides a trade-off between privacy and performance. This proposal aims to provide flexibility to the parties while mitigating the performance issues of [5].

1.3.6 Evaluation of the improved PPoS proposals

An evaluation of both the privacy and performance of the new PPoS proposals was conducted and is presented in Section 5.3.

Chapter 2
State of the Art

A description of all currently available PPoS proposals, *Proof-of-Stake for Privacy-Aware Blockchains* [25], *Anonymous Lottery In the Proof-of-Stake setting* [5], *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* [36], and *Zether: Towards Privacy in a Smart Contract World* [12], is provided. We only consider PPoS proposals with a record of scientific publication. Note the emphasis on how the proposals aim to achieve privacy.

2.1 *Proof-of-Stake Protocols for Privacy-Aware Blockchains*

Initially proposed in 2019 by Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi, the paper *Proof-of-Stake Protocols for Privacy-Aware Blockchains* [25] is one of the first two studies into the feasibility of PPoS. Ganesh et al. design a protocol, $Protocol^{\epsilon,LE}$, to realize their ideal functionality. To summarize, the protocol allows stakeholders to check whether they have been selected locally and, if appropriate, create a zero-knowledge proof of selection. The proposal achieves privacy by including stake as part of a zero-knowledge proof of selection. In order to anonymize the identity of the selected, they utilize an anonymous broadcast channel.

The protocol $Protocol^{\epsilon,LE}$ has three phases, *Initialization*, *Lottery and Publishing*, and *Get Information*. Each party gets a list of stakeholders' committed stakes and signature verification keys in the *Initialization* phase. Stakeholders additionally acquire their signing key and relative stake. The list successfully hides the stakeholders' identity since a signature verification key does not reveal any information about the owner of the corresponding signing key. In addition, the privacy of the stakeholders' stake is preserved as a commitment to a stake does not reveal any information about the stake. In the *Lottery and Publishing* phase, stakeholders consult a predicate function to determine whether or not they have been selected. Note that an underlying selection function, defined by some preexisting PoS protocol, determines the selected.

To prove selection, the selected must prove ownership of the winning stake. The stakeholder proves in zero-knowledge that they were selected, ensuring the privacy of the stake. The signed message and proof of selection are published over an Anonymous Broadcast Channel (ABC). Without an ABC, broadcasting the message reveals the broadcaster, thus revealing the selected's identity. The use of an ABC also hides the frequency with which a stakeholder is selected. Finally, in the *Get Information* phase, stakeholders output their relative stake when requested to do so by the environment.

To summarize, the proposal achieves privacy by utilizing a zero-knowledge proof that separates the selected's stake from validating the proof of selection. In other words, the validator does not need the selected's stake to validate the proof, thus, maintaining the privacy of the stake. The selected broadcasts their proof of selection over an ABC to keep their identity hidden. The proposal also provides an instantiation with the PoS protocol, Ouroboros Praos, in which they replace the Verifiable Random Function (VRF) of Ouroboros Praos with an Anonymous Verifiable Random Function (AVRF).

2.2 *Anonymous Lottery In the Proof-of-Stake setting*

The paper *Anonymous Lottery In the Proof-of-Stake setting* by Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou [5] is a recent PPoS proposal from 2020. The proposal achieves privacy by separating the identity and stake of the stakeholder from validating the proof of selection. They present a protocol proposal for both the single and multi-stake settings.

Baldimtsi et al. [5] define an ideal functionality and apply it to the selection function of the PoS protocol Algorand [29], resulting in the protocol $\Pi^{Eligible}_{Anon-Selection}$, Algorithm 5.3.The protocol is made up of four main parts, *Initialization()*, *EligibilityCheck()*, *CreateProof()*, and *Verify()*. First, each party, P_i, runs the *Initialization()* protocol, seen in [5, p. 12], to generate their public keys, pk_i, and secret keys, sk_i, which i.a. includes a trapdoor permutation key pair $(TRP.pk_i, TRP.sk_i)$ for a trapdoor permutation f.

Following initialization, for each tag, the stakeholders run *EligibilityCheck()*, seen in [5, p. 13], to check if they are selected, i.e., if they are eligible to publish a message to vote on or propose a block. A *tag* defines when a new selection process is performed. *EligibilityCheck()* first runs a *ProcessRO* algorithm which computes a vector $\vec{V}_{tag}$. $\vec{V}_{tag}$'s inverse $v_i = f^{-1}_{TRP.ski}(\vec{V}_{tag}[i])$ is computed using the party's trapdoor secret key. v_i is connected to the selected's identity and is, thus, hidden from other parties. *EligibilityCheck()* runs the *Eligible()* protocol, seen in [5, p. 11] and [5, p. 19] for the single-stake setting and the multi-stake setting, respectively.

Eligible() uses v_i to decide b_{tag} in the single-stake setting and to calculate wt_i in the multi-stake setting. $b_{tag} = 1$ if the party is selected for the tag, and $b_{tag} = 0$ if not. In the multi-stake setting, $wt_i > 0$ if the party is selected, and $wt_i = 0$ if not.

If eligible, i.e., $b_{tag} = 1$ or $wt_i > 0$, the stakeholder runs the *CreateProof()* protocol, Algorithm 5.7. *CreateProof()* creates a zero-knowledge proof, π_{NIZK}, to avoid using and revealing the public key (identity) and stake of the selected to the receivers when verifying the proof of selection. The zero knowledge proof, π_{NIZK}, is part of a larger proof, π_i. The proof, π_i, additionally contains i.a. a commitment, C_i^v, to v_i to prove knowledge of the pre-image of one of the $\vec{V}_{tag}[i]$ making them eligible. In the multi-stake setting, the proof also contains a commitment, cm_i, to the party's stake, $stake_i$. *CreateProof()* returns the proof, π_i, which is then sent along with a message and the *tag*. Note that for the multi-stake setting, the party sends wt_i number of messages and proofs. As wt_i is proportional to the parties' stake, the proofs are unlinkable as not to reveal the parties' stake to the receivers. For all parties to be able to verify the received proof(s), a *Verify()* protocol, Algorithm 5.9, takes the *tag*, the message, and the proof as input and returns whether or not this proof is valid for the given message and *tag*. Note that the private information of the selected, i.e., the public key or stake, is not needed to validate the proof and is therefore not revealed to the receivers.

To summarize, utilizing a zero-knowledge proof allows the other parties to verify the proof of selection by confirming that the prover possesses a witness, i.a. a public key and stake, that confirms the statement *the prover was chosen in the selection algorithm* without revealing the witness. This simple statement is divided into nine statements to check when creating the zero-knowledge proof for the single-stake setting and twelve statements for the multi-stake setting. The number of statements to check is greater for the multi-stake setting since the stake is also part of the zero-knowledge proof. We refer to [5] for more details.

2.3 *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake*

Concurrent and independent of Ganesh et al., Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas present a formal model for a privacy-preserving PoS based distributed ledger in the paper *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* [36]. Furthermore, they propose a protocol, Ouroboros Crypsinous, that realizes this model. The proposed protocol is inspired by the slot-based PoS, Ouroboros Genesis [4], and has a Zerocash-like [50] transaction system, i.e., a privacy-preserving transaction system.

In the simplest of terms, Ouroborous Crypsinous runs variants of Ouroborous Genesis and Zerocash together, creating its own unique distributed ledger. Ouroborous Crypsinous moves the leadership proof, i.e., proof of selection, of Ouroborous Genesis into zero-knowledge and proves stake through a one-to-one Zerocash transfer. Similar to the proposal by Ganesh et al. [25], Ouroborous Crypsinous replaces the VRF of Ouroborous Genisis with an AVRF and informally claims to broadcasts over an ABC.

Ouroboros Crypsinous formalizes a privacy-enhanced transaction ledger that is secured in the UC (Universal Composability) setting. Therefore it can be applicable to both PoW and PoS settings. For the leader election in Ouroboros Crypsinous, each party holds Zerocash-style coins where each coin competes to be a leader in a consensus slot. A coin is chosen to be a leader if its respective pseudorandom value generated using a VRF meets a target. The party holding the winning coin sends a Non-Interactive Zero-Knowledge (NIZK) proof of its winning coin, which also proves that the coin is unspent. The authors prove the security of their protocol in an adaptive setting where even if an adversary corrupts some parties during the execution of the protocol, it does not gain any advantage with respect to privacy.

2.4 *Zether: Towards Privacy in a Smart Contract World*

The paper *Zether: Towards Privacy in a Smart Contract World* by Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh [12] from 2020 present Zether: "a fully-decentralized payment mechanism" described as a smart contract that "keeps the account balances encrypted and exposes methods to deposit, transfer and withdraw funds to and from accounts through cryptographic proofs" [12].

Bünz et al. [12] mention several applications of Zether. Importantly, for the scope of this thesis, they apply Zether to achieve a PPoS proposal. They propose to use a zero-knowledge proof of selection similar to their proposed zero-knowledge proof for anonymous Zether transfer. Each stakeholder participating in the "lottery", i.e., selection, encrypts, using ElGamal, a lottery ticket, and their stake under their public key. For the zero-knowledge proof, Bünz et al. [12] use their new and proposed zero-knowledge proof mechanism Σ-Bullets with the ability to prove statements on encrypted values. The winner, i.e., selected, proves in zero-knowledge that they know a winning ticket without revealing it.

Note that details on their PPoS proposal are not provided; Bünz et al. [12] leave formal analysis and further investigation into this proposal for future work.

Chapter 3 Mathematical background

The mathematical background necessary for understanding the thesis primarily concerns the topics of homomorphic encryption and SNARKs, and is presented in the following sections.

3.1 Homomorphic Encryption

Homomorphic encryption (HE) enables parties to carry out simple arithmetic operations, i.a. addition and multiplication, on encrypted data without first decrypting it. In plain terms, if the variables a and b are homomorphically encrypted,

$$HE.Enc(a) = a' \text{ and } HE.Enc(b) = b',$$

then the homomorphic addition of a' and b', once decrypted, equals the addition of a and b, i.e.

$$HE.Add(a', b') = c' \text{ and } HE.Dec(c') = a + b.$$

3.1.1 *Definitions*

Before detailing the homomorphic encryption scheme used in this thesis, we provide a definition of homomorphic encryption. The definition is provided by [11] and primarily mimics the exposition by Brakerski and Vaikuntanathan [57].

The homomorphic (public-key) encryption scheme,

$$HE = (HE.Setup, HE.KeyGen, HE.Enc, HE.Dec, HE.Eval),$$

is defined as a quintuple of the following Probabilistic Polynomial-Time (PPT) algorithms:

- **Setup.** The setup algorithm

  $$params \leftarrow HE.Setup(1^{lambda})$$

  with the security parameter, *lambda*, outputs *params*, i.e., the global parameters of the encryption scheme.

- **Key generation.** The secret and public key generation algorithms,

  $$sk \leftarrow HE.SecretKeyGen(params)$$

  and

  $$pk \leftarrow HE.PublicKeyGen(params, sk),$$

  take as input the global parameters *params* (and the secret key *sk*) and outputs the secret key and public key, respectively.

- **Encryption.** The encryption algorithm,

  $$c \leftarrow HE.Enc(params, pk, \mu),$$

  with the inputs, global parameters *params*, public key *pk*, and message $\mu \in R_{\mathcal{M}}$, outputs a ciphertext c.

- **Decryption.** Conversely, the decryption algorithm

  $$\mu^* \leftarrow HE.Dec(params, sk, c)$$

  with the inputs, global parameters *params*, secret key *sk*, and ciphertext c, outputs a message $\mu^* \in R_{\mathcal{M}}$.

- **Homomorphic Evaluation.** The evaluation algorithm,

  $$c_f \leftarrow HE.Eval(pk, f, c_1, ..., c_l)$$

  with the inputs, public key *pk*, a function $f : R^l_{\mathcal{M}} \rightarrow R_{\mathcal{M}}$ which is an arithmetic circuit over $R_{\mathcal{M}}$, and a set of l ciphertexts $c_1, ..., c_l$, outputs a ciphertext c_f.

Note that “the message space $\mathcal{M}$ of the encryption schemes will always be some ring $R_{\mathcal{M}}$, and the functions to be evaluated will be represented as arithmetic circuits over this ring, composed of addition and multiplication gates” [11].

3.1.2 The BGV FHE scheme

This thesis implements the Fully Homomorphic Encryption (FHE) scheme *Brakerski-Gentry-Vaikuntanathan (BGV)* by Brakerski et al., described in the paper *(Leveled) FHE without Bootstrapping* [11]. The security of BGV stems from the hardness of the ring-LWE (learning with error) problem. Against known lattice attacks, BGV achieves 2^λ security. The scheme consists of a quintuple of polynomial-time algorithms, $FHE.Setup()$, $FHE.KeyGen()$, $FHE.Enc()$, $FHE.Dec()$, and $FHE.Eval()$. The latter invokes $FHE.Add()$ and $FHE.Mult()$, of which our application is only interested in $FHE.Add()$. Similarly, our application does not make use of $FHE.Dec()$.

$FHE.Setup()$ is the first algorithm to be performed. It takes as input the security parameter, λ, and a parameter, A, which indicates "the number of levels of arithmetic circuit that we want our FHE scheme to be capable of evaluating" [11]. The parameter A is further defined in Subsection 3.1.2.1. $FHE.Setup()$ produces i.a. a ladder of global parameters, $params_a$, for the input level of the circuit, $a = A$, to the output level of the circuit, $a = 0$. Next, $FHE.KeyGen()$ is called, with the ladder of parameters, $params_a$, as input, to generate the public and secret keys, $FHE.pk_a$ and $FHE.sk_a$. For the sake of clarity we will refer to $params_a$, $FHE.pk_a$, and $FHE.sk_a$ as $params$, $FHE.pk$, and $FHE.sk$, respectively, throughout this thesis. Note that the global parameter $params$ is input to all the BGV algorithms (excluding $FHE.Setup()$), even when not specified. To homomorphically encrypt some message, m, a party runs $FHE.Enc()$ with inputs $FHE.pk$, and m to produce a ciphertext, c. The decryption of the ciphertext is done by $FHE.Dec()$, with inputs $FHE.sk$, and c, but as mentioned, this is irrelevant to our application.

Since our application is only interested in $FHE.Add()$, we assume modifications to $FHE.Eval()$ such that $FHE.Mult()$ is discontinued, and the circuit, f, is composed solely of addition gates rather than layers of alternating addition and multiplication gates. Thus, $FHE.Eval(FHE.pk, f, c_1, ..., c_n)$ only encompasses $FHE.Add(FHE.pk, c_1, c_2)$, which returns the homomorphically encrypted sum of c_1 and c_2. Since our application only uses $FHE.Eval()$ for the addition of two ciphertexts, we assume further modifications such that $FHE.Eval()$ takes as input two ciphertexts rather than n ciphertexts. It is important to note that if the two ciphertexts are not encrypted under the same secret key, $FHE.Eval()$ will invoke a ciphertext refreshing procedure, $FHE.Refresh()$, in order to make it so before running arithmetic operations. Running $FHE.Refresh()$ often has negative effects on performance; therefore, it is preferable to perform homomorphic evaluations on ciphertexts encrypted by the same public key whenever possible. $FHE.Refresh()$ also serves a second purpose in the reduction of noise.

3.1.2.1 Noise Handling

Traditionally, if noise accumulates due to the consecutive execution of arithmetic operations on homomorphically encrypted values, a bootstrapping procedure is performed. However, the version of the BGV scheme utilized in this thesis does not require such bootstrapping. Instead, it supports homomorphic evaluations up to a predefined level, A, and a refreshing procedure, $FHE.Refresh()$, which reduces noise [11]. Note that while $FHE.Refresh()$ is needed after every homomorphic multiplication, homomorphic addition increases noise more slowly and may not require $FHE.Refresh()$ at all. We define A as the maximum amount of additions necessary for the instantiation of our proposal with Algorand and assume that such a level is possible. Based on the existing documentation [11], we assume this is plausible because (1) we will only perform homomorphic additions, and (2) we are not concerned with homomorphic decryption and thus as long as the accumulated noise does not impede a greater than comparison, we do not mind it.

3.1.3 Homomorphic Comparison Protocols

In addition to the arithmetic operations, i.e., addition and multiplication, comparison-based computations on homomorphically encrypted ciphertexts are possible [54]. Recent works [15, 10, 16] show ways to perform such comparisons efficiently and with optimal complexity. This thesis applies these comparison properties over homomorphically encrypted data in the greater-than and equality comparison protocols shown in Algorithms 3.1 and 3.2, respectively. The former takes as input the homomorphically encrypted values x' and y' and homomorphically evaluates whether x is greater than y. The evaluation is done using a Greater-Than comparison circuit, $GT_{\mathbb{R}}$, which returns 1 if x is greater than y and 0 if not.

Algorithm 3.1 Greater-Than Comparison Protocol

```
Protocol FHE.GreaterThan($x', y'$)
1: Evaluate the Greater-Than circuit, $GT_{\mathbb{R}}(x', y')$, and receive $b$
2: Return $b$
```

Similarly, the homomorphic equality test takes as input the homomorphically encrypted values x' and y' and homomorphically evaluates whether x is equal to y. Some equality comparison circuit, $EQ_{\mathbb{R}}$, is used to handle this evaluation, returning 1 if x is equal to y and 0 if not.

Algorithm 3.2 Equality Comparison Protocol

```
Protocol FHE.Equality($x', y'$)
1: Evaluate the Equality circuit, $EQ_{\mathbb{R}}(x', y')$, and receive $b$
2: Return $b$
```

3.2 SNARKs

SNARKs are a method of proving that something is true without revealing any other information. Namely, a prover wants to convince a verifier that some statement is true without revealing the secret information w used to prove the statement.

Developments within the field of zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs), [44, 19, 32], have enabled the implementation of zk-SNARKs in practice. To illustrate the use of SNARKs, we refer to three algorithms presented in Baldimtsi et al.'s paper [5]. These three probabilistic polynomial-time algorithms define a NIZK proof system as follows:

- $crs \leftarrow NIZK.Setup(1^{\lambda})$: Produces a Common Reference String (CRS).
- $\pi \leftarrow NIZK.Prove(crs, stmt, \omega)$: Produces a proof π.
- $0/1 \leftarrow NIZK.Verify(crs, stmt, \pi)$: Verifies the proof. Outputs 1 if the proof is accepted and 0 if not.

Note that λ, x, and w denote the security parameter, statement, and witness, respectively [5]. The cleverness of SNARKs and, thereby, NIZK systems lies in that the witness, i.e., the prover's secret information, is used to prove the statement without being revealed to the verifier.

Polynomials are at the very core of SNARKs. According to *Why and How zk-SNARK Works: Definitive Explanation* by Petkus [45], first, the statement to prove is reduced to *the language of math* and then further reduced into *the realm of polynomials*. The paper explains that after reduction, the prover knows a (secret) w such that for all a, two polynomials are equal, for instance, $z(a) * w(a) = p(a)$. In simpler terms, as explained by [45]:

- The verifier evaluates his polynomial locally with a chosen random value for a.
- The verifier provides the prover with a and asks the prover to evaluate the polynomial in question.
- After evaluating his polynomial at a, the prover gives the result to the verifier.
- Finally, the verifier checks whether the local result is the same as the prover's result. If the results are equal, the statement is proven with high confidence.

This confidence is based on an important property of polynomials: “If we have two non-equal polynomials of degree at most d, they can intersect at no more than d points” [45]. Thus, a single error in $w(a)$ makes the prover’s polynomial, $z(a) * w(a)$, different from the verifier’s polynomial, $p(a)$, and the error is visible almost everywhere. The probability that the verifier accidentally chooses a as one of the d shared points is considered negligible, given a certain range of a, for example, from 1 to 10^{77} [45].

To avoid any issues of the prover cheating when the prover knows a, the verifier sends the homomorphically encrypted value of a, that is a', instead of the plaintext a to the prover. [45] explains that without knowledge of a, it is difficult for the prover to find an illicit polynomial equal to the verifier’s polynomial. Moreover, the paper clarifies that the prover now homomorphically evaluates his polynomial using a'. The verifier receives the prover’s evaluated polynomial and checks equality in the encrypted form.

The non-interactivity of SNARKs is achieved by a trusted setup, as the encrypted a can be reused if (the plaintext) a is not revealed. The paper demonstrates that either a single honest party or multiple parties together generate a composite CRS such that neither of the parties, i.e., none of the verifiers or provers, know the secret a. Simplified, the CRS contains the encrypted a. The study goes on to mention that when the CRS is generated, “any prover and any verifier can use it in order to conduct non-interactive zero-knowledge proofs” [45]. Thus, after distributing the encrypted a to all parties, the protocol boils down to the prover evaluating her polynomial with the encrypted a and sending the result (proof). The verifier only participates in the final step to verify this result (proof).

While the methods used in the SNARKs protocol mainly contribute to a fast verification process, [45] explains that zero-knowledge almost comes for “free”. The verifier does not gain any information about the unknown polynomial, $p(a)$, since she only checks the equality in the encrypted form and does not know a. When the prover sends his evaluation to the verifier, he proves that he knows the coefficients of the polynomial because knowing a polynomial is knowing its coefficients. In practice, if the possible coefficients of this polynomial are few, the verifier may brute-force until the result equals the prover’s response. This issue is “easily” solved by adding a random encrypted number to both sides, i.e., shifting by some random number. Consequently, SNARKs achieve zero-knowledge as the verifier neither obtains a, $p(a)$, nor the secret information $w(a)$.

Chapter 4 Analysis

PoS encounters challenges in its attempts to achieve privacy. We review and analyze all currently available PPoS proposals, paying particular attention to identifying issues that may be improved upon to achieve a practically implementable PPoS. Possible areas of improvement within the PPoS proposals are identified based on the issues discovered in the analyses. Finally, we conclude which PPoS proposal and improvement area to pursue further. Algorand is detailed and analyzed as a result of this conclusion.

4.1 The Problem of Privacy in PoS

As alluded to in Chapter 1, PoS encounters challenges PoW does not encounter when implementing privacy-preserving measures. These challenges arise because PoS and PoW experience a crucial difference regarding their consensus. As detailed in the pre-project of this thesis, this difference regards "the connection, or lack thereof, between a [selected] party's identity and the proof of [selection]. That is to say, in PoW, the proof of [selection] can be completely disconnected from the identity of the [selected], as the proof is the solution to a computationally hard problem. On the other hand, in PoS, it is impossible for a party to completely disconnect their identity from the proof of [selection], as the party's identity is part of the proof. This inability to separate identity from the proof of [selection] makes existing PoS protocols, i.e. [14, 37], incompatible with privacy-preserving cryptocurrencies such as Zcash [50] and Monero [46]" [52]. For these cryptocurrencies, maintaining the anonymity of their users is of the utmost importance; therefore, switching to a PoS protocol for efficiency purposes is not an option since the PoS protocols currently available reveal the identity and stake of the selected party.

As a final note, it is worth mentioning that if a Blockchain using PoS wants to achieve complete privacy, it may need to consider the reward mechanism implemented. For example, a frequency analysis of the distribution of rewards could reveal the frequency with which parties are selected, thus, revealing their stake. Therefore, the design and implementation of a PPoS on a Blockchain must consider the reward mechanism.

4.2 Analysis of Formal Studies into PPoS

We analyze the currently available PPoS proposals published within the scientific community [25, 5, 36], specifically focusing on identifying issues pertaining to privacy, efficiency, and inclusion of all aspects relevant to distributed consensus. Note that Zether [12] is not analyzed, as the existing documentation insufficient for such an analysis.

4.2.1 *Proof-of-Stake Protocols for Privacy-Aware Blockchains*

The proposal by Ganesh et al. [25] aims to anonymize the stake and identity of stakeholders. To anonymize the stake, stakeholders make a commitment to their stake and create a zero-knowledge proof of selection on the committed stake. Stakeholders anonymize their identity through the use of an ABC. Without an ABC, an adversary can deduce a stakeholder's stake by analyzing the frequency in which the underlying PoS selects the stakeholder. Such a frequency analysis is possible since the probability of being selected is proportional to the stakeholder's stake in the system.

The reliance on an ABC to hide the identity of a stakeholder poses several issues. The first issue arises in the cost of both the implementation and maintenance of an ABC [51]. This significantly impacts the efficiency of the PPoS system, additionally affected by adversarial behaviors. The second issue stems from the intuition that "potential identity leaks from the network-layer can be removed by employing anonymous broadcast channels" [39]. As pointed out in [39], by Kohlweiss et al., this intuition is flawed, as their proposed attacks render ideal ABCs insufficient in protecting the identity of stakeholders. The flaw is covered thoroughly in Section 4.2.3, since [39] primarily concerns itself with Ouroboros Crypsinous [36]. While Ganesh et al.'s proposed ideal functionality for an ABC differs from that of [39], [25] still permits an adversary to send messages to targeted parties. "This adversarial capability is sufficient to mount the attacks [proposed] in" [39].

It is important to note that the ideal VRF functionality described by Ganesh et al. as part of their private lottery framework, although similar, is not the same as the VRF implemented in the framework's adaptation to the PoS protocol Ouroboros Praos. In the instantiation of Ouroboros Praos, the VRF specified by the framework would reveal the frequency in which an account is selected, thus revealing information about the party's stake [25]. To solve this, Ganesh et al. introduce a new cryptographic primitive referred to as AVRF. An AVRF is "a VRF in which there exist multiple verification keys for the same secret key, and where it is hard, given two valid proofs for different inputs under different verification keys, to tell whether they were generated by the same secret key or not" [25]. While this AVRF could be of independent interest, it is not the VRF proposed by the framework, and thus, it may weaken the validity of the instantiation of said framework.

Notably, applying the proposed PPoS mechanism to committee-based PoS protocols, such as Algorand, although possible, may prove to be somewhat inefficient. For example, in Algorand, detailed in Subsection 4.4.1, once selected to participate in a committee as a proposer or validator, each party generates a proof of selection. Next, proposers propose a block by gossiping its hash along with their proof. Validators then vote on the block, similarly attaching their proof to their vote. Notice that after votes are submitted, all participants verify the votes they have received. The process repeats, with a new committee of validators, until the votes for a block hash reach some threshold. Altogether Algorand requires a lot of proof generation and validation to select a single block. Thus, the modifications necessary to make Algorand private, by the proposed framework, will increase the generation and validation times, effectively decreasing the efficiency of the protocol.

Although Ganesh et al. [25] do not define how their proposal coincides with reward distribution, they state that the anonymous distribution of rewards is possible if the cryptocurrency allows for anonymous transactions and anonymous account creation.

4.2.2 *Anonymous Lottery In the Proof-of-Stake Setting*

An analysis of the proposal by Baldimtsi et al. [5] identifies two potential issues. First, the large size of the zero-knowledge proofs in both the single- and multi-stake settings negatively influences performance. Secondly, the need for multiple unlinkable proofs in the multi-stake setting only emphasizes this performance issue.

The issue of proof size arises both in the single-stake setting and in the multi-stake setting. There are nine statements to include in the single-stake setting when creating the zero-knowledge proof and, thus, to check when validating the proof. Regarding the single-stake setting of their proposal, Baldimtsi et al. claim that "the zk-SNARKs used in [their] implementations will require approximately 129K R1CS constraints" [5]. In the multi-stake setting, a verification of the zero-knowledge proof checks twelve statements. The paper also notes that this setting requires more than 22K additional Rank-1 Constraint Satisfaction (R1CS) constraints. Hence, the multi-stake setting requires more than 150K R1CS constraints per proof.

The large proof size has consequences in terms of verification time. As noted in the previous section, Algorand [29] requires a lot of proof generation and validation to select a single block. As such, the verification time is vital when applying the ideal functionality of [5] to Algorand, as the proof size greatly affects the verification time.

Another issue with [5]'s PPoS proposal is the need for multiple unlinkable proofs for each of wt_i's indexes in the multi-stake setting. While the total number of selected sub-users (a.k.a. indexes) is attached to one proof in Algorand [29], Baldmitsi et al.'s application to Algorand requires one proof for each of the selected indexes. The choice to give a separate unlinkable zero-knowledge proof for each selected *index* hides the total number of selected indexes, proportional to the stake, from the receiver. However, this choice causes the multi-stake setting to encounter an even larger performance issue since it not only requires more statements to be checked for each proof, but also requires one zero-knowledge proof verifying all twelve statements for each of wt_i's indexes. Additionally, even though their protocol in the multi-stake setting depends on unlinkable proofs, the paper does not define unlinkability.

Another consequence of the large proof size is an increased communication cost. For example, regarding the application to Algorand, as per the gossip protocol, each participant relays the messages received to a small group of participants who repeat this process. For this reason, especially in the multi-stake setting with a zero-knowledge proof for each of wt_i's indexes, many messages and proofs are sent. Thus, in combination with the large proof size of the zero-knowledge proofs in this proposal, the number of proofs and messages could cause bandwidth problems.

Finally, it is worth mentioning that Baldimtsi et al.'s proposal [5] only considers the static-stake setting and does not account for the distribution of rewards relating to consensus. Thus, an investigation into the dynamic-stake setting, where new participants can join after initialization, could be of interest. In addition, in a PPoS consensus mechanism where the privacy of the user is of importance, a privacy reward mechanism should be considered to include all aspects of a consensus mechanism.

4.2.3 *Ouroboros Crypsinous*

An analysis of Kerber et al.'s PPoS blockchain protocol, Ouroboros Crypsinous, [36] identifies two potential issues, (1) the reliance on an ABC and (2) the lack of applicability to other PoS blockchains.

Similarly to the proposal by Ganesh et al., as stated in [39], Kerber et al. "recognizes that protocol messages travel over a public network [...] and the adversary can learn information about the identity of an elected party through the leakage of the network channel, e.g., by associating a certain block to a certain IP address. They informally claim that if the underlying communications were carried over an *anonymous* broadcast channel instead, then the network meta-data of the sender is hidden, hence breaking the link between a block and its sender. Since this claim is informal, no particular anonymous channel functionality is provided" [39]. The recently published paper, *On the Anonymity Guarantees of Anonymous Proof-of-Stake Protocols* [39], by Kohlweiss et al., shows that an ABC in Ouroboros Crypsinous is insufficient in hiding the stakes of stakeholders. They do so through a series of attacks, ultimately highlighting that the network leakage, supposedly combated by the ABC, is still exploitable in the network. It is worth noting that, since "there exists no implementation of a privacy-preserving PoS blockchain" [39], the proposed attacks were performed on Zcash [50], which is a PoW based blockchain, with similarities to Ouroboros Crypsinous.

Unlike the PPoS proposals [25, 5], which aim to create proposals applicable to different underlying PoSs and blockchains, Kerber et al. aim to develop a privacy-preserving blockchain built on Ouroboros Genesis [4] and Zerocash [50]. Note that Ouroboros Crypsinous maintains anonymity throughout its reward distribution. However, since Ouroboros Crypsinous is reliant on the build of the blockchain, it is inherently inapplicable to other PoS blockchains. That is, Ouroboros Crypsinous cannot be applied to make other PoSs private.

4.3 Areas Open to Improvement Within the PPoS proposals

For both *Proof-of-Stake Protocols for Privacy-Aware Blockchains* [25] and *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake* [36], analyzed in Sections 4.2.1 and 4.2.3, respectively, the most pressing issue is the reliance upon an ABC to anonymize identity. Kohlweiss et al. [39] made apparent that an ABC does not eliminate the threats of inherent network leakage. Thus, a PPoS proposal reliant on such a channel cannot be trusted to provide complete privacy for its stakeholders. In order to mitigate this issue, we would need to circumvent the use of an ABC.

Another issue encountered in both [25] and [36] lies in their applicability to other PoS blockchains. While the former may apply to other slot-based PoS blockchains, its efficiency in application to committee-based PoS blockchains remains unexplored. An investigation into this application and efficiency could be interesting but strays from the path of our research question. In contrast, the latter proposal is for a blockchain with a privacy-preserving consensus mechanism, and as such, does not aim to be applicable to other PoS blockchains. Note that the privacy of its consensus mechanism is directly dependent on the construction of the ledger. While this proposal is fascinating, we deem the construction of a blockchain outside the scope of our thesis.

While Ouroboros Crypsinous [36] is the only proposal to account for the anonymous distribution of rewards, [25] and [5] leave reward distribution to the underlying PoS blockchain. In doing so, they maintain applicability to “all” such ledgers and leave modification of the reward mechanism to future work. Similarly to [25] and [5], we deem that the privacy preservation of reward distribution should be left to the underlying blockchain in order to maintain applicability. Additionally, we reckon that there exist a variety of plausible ways of making a privacy-preserving reward mechanism. Moreover, not all blockchains opt to distribute rewards relating to consensus, as they do not all require the same incentives; thus, such reward mechanisms are entirely unrelated to consensus and, thereby, outside the scope of our thesis.

The main issues encountered in *Anonymous Lottery In the Proof-of-Stake setting* [5], analyzed in Subsection 4.2.2, regards performance. These issues include the size of the proofs in the single- and multi-stake settings and the number of proof generated and sent in the multi-stake setting. The combination of these issues significantly impacts performance in the multi-stake setting since a party sends one large proof for each unit of their selected stake, resulting in the generation, gossip, and verification of many proofs. We hypothesize that we can decrease the number of proofs sent in the multi-stake setting.

In conclusion, the areas open to improvement are as follows;

- the circumvention of the use of an ABC in [25]
- the circumvention of the use of an ABC in [36]
- the mitigation of the need for multiple unlinkable proofs in the multi-stake setting of [5]
- the reduction of proof size in [25] and [5]

Both the first and second improvement areas concern the mitigation of inherent network leakage without using an ABC. We believe that improvements to this area would involve considerable modifications to the overall design of the proposals. Regarding the third improvement area, we have various ideas for how to mitigate the need for multiple unlinkable proofs. For this reason, we conclude to pursue the third improvement area further, keeping the fourth improvement area in mind while doing so. Since we want to improve upon the proposal by Baldimtsi et al. [5], a closer look at the PoS scheme it instantiates, namely Algorand, is in order. The analysis of Algorand focuses on identifying where the PoS blockchain reveals identity and stake.

4.4 Algorand

We provide a description of the PoS based cryptocurrency Algorand detailing the process in which Algorand decides upon which parties get to propose the next block for the blockchain and how consensus on a block is reached. Then, we analyze what areas of Algorand require modification to achieve the privacy-preservation of stake and identity. This thesis considers the procedures described in the paper *Algorand: Scaling Byzantine Agreements for Cryptocurrencies* from 2017 by Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, Nickolai Zeldovich [29].

4.4.1 Description

Using cryptographic sortition, Algorithm 4.1, Algorand [29] chooses a fraction of the parties to participate in committees as potential block leaders or verifiers. The chosen block leaders propose a new block in a given *round*. After the proposal of new blocks, the chosen verifiers verify and vote on the proposed blocks in *step*s.

Algorithm 4.1 chooses a fraction of parties based on their weight, w, representing their stake. Each chosen party is provided with the parameter j, a hash, $hash$, and a proof of their priority (proof of selection), π, calculated using a VRF with their secret key. A party can be chosen more than one time. The hash variable determines j, i.e., how many sub-users are selected. For example, if a party is chosen as a verifier with $j = 2$, the verifier has two votes.

Algorithm 4.1 Sortition

```
1: procedure SORTITION($sk, seed, \tau, role, w, W$)
2:     $\langle hash, \pi \rangle \leftarrow VRF_{sk}(seed || role)$
3:     $p \leftarrow \frac{\tau}{W}$
4:     $j \leftarrow 0$
5:     while $2^{\frac{hash}{hashlen}} \notin \left[\sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p)\right]$
6:         do $j \leftarrow j + 1$
7:     return $hash, \pi, j$
8: end procedure
```

Note that W denotes the weight of all parties, *seed* is a publicly known random seed, τ denotes a threshold that determines the expected number of parties selected for a *role*, and *role* distinguishes the different roles that a party may be selected for (e.g., block proposer or verifier).

Algorithm 4.2 BA$\star$

```
1: procedure BA$\star$($ctx, round, block$)
2:     $hblock \leftarrow Reduction(ctx, round, H(block))$
3:     $hblock_\star \leftarrow BinaryBA\star(ctx, round, hblock)$
4:     // Check if we reached "final" or "tentative" consensus
5:     $r \leftarrow CountVotes(ctx, round, FINAL, T_{FINAL}, \tau_{FINAL}, \lambda_{STEP})$
6:     if $hblock_\star = r$ then
7:         return $\langle FINAL, BlockOfHash(hblock_\star)\rangle$
8:     else
9:         return $\langle TENTATIVE, BlockOfHash(hblock_\star)\rangle$
10: end procedure
```

The overall Byzantine Agreement (BA) algorithm, $BA\star()$, Algorithm 4.2, runs for each *round* and primarily consists of the *Reduction()* and $BinaryBA\star()$ procedures [29, p. 59-60]. Both *Reduction()* and $BinaryBA\star()$ runs in *step*s, while the *Sortition()* procedure, Algorithm 4.1, is run for each *step*. Hence, the $BA\star()$ procedure chooses new committee members for each *step*. Note that *ctx* denotes a context that captures the current state of the ledger, and *block*, as its name suggests, denotes a block.

The *Reduction()* procedure consists of two steps. The first step uses the *Sortition()* procedure, Algorithm 4.1, to select a small fraction of parties to form a committee of potential block leaders. They propose blocks by “gossiping,” i.e., broadcasting, their public key, *pk*, and a signed message. The message contains i.a. *value*, the hash of the block they propose to add to the blockchain, a hash, *hash*, and the proof of selection, π, returned from *Sortition()*, Algorithm 4.1. After a short waiting time for all parties to receive the potential block leaders' messages, the receivers verify the signed messages and received proofs using the sender's gossiped *pk*. Next, the receivers (1) calculate the sender's “voting power”, j, (2) count j votes for the block hash *value*, and (3) compare the current votes for *value* with a threshold of votes.

In the second and final step of *Reduction()*, the procedure chooses a new group of committee members as verifiers. They gossip a signed message with, i.a. the block hash *value*, for which they received at least some threshold of votes in the previous step. I.e., they gossip (vote for) the block hash that enough potential block leaders proposed. Again, after a short waiting time, all parties count the votes for each *value* (received from the verifiers). *Reduction()* returns the *value* that received at least some threshold of votes.

Following the *Reduction()* procedure, all parties in the network initialize a BA algorithm, $BinaryBA\star()$, with the block hash returned from *Reduction()*, as seen in Algorithm 4.2. The BA algorithm is used to reach consensus on a block and executes in *step*s. For each *step*, the *Sortition()* procedure, Algorithm 4.1, appoints a group of parties to form a committee of verifiers that gossip a message, i.e., "their vote" on a block hash, *value*. This is repeated with new committee members until enough parties in the committee reach consensus on a block hash, *value*. Note that the committee members vote for the block hash, *value*, that they receive at least some threshold of votes for in the previous *step*. Final consensus is reached, and the block with block hash *value* is added to the blockchain if a threshold of the committee members vote for *value*.

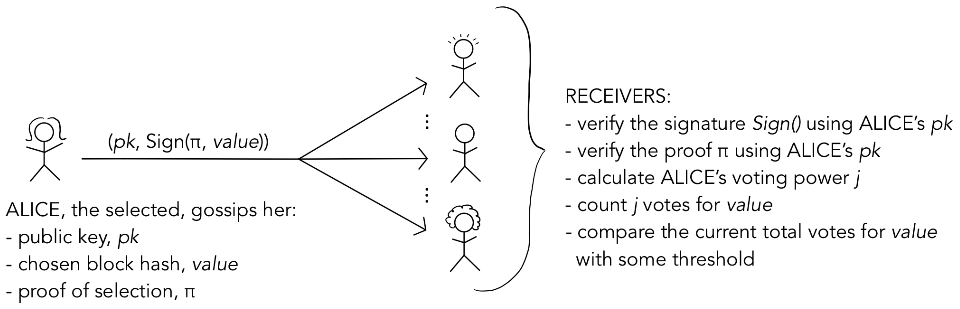
(pk, Sign(π, value))
ALICE, the selected, gossips her:
- public key, pk
- chosen block hash, value
- proof of selection, π
RECEIVERS:
- verify the signature Sign() using ALICE's pk
- verify the proof π using ALICE's pk
- calculate ALICE's voting power j
- count j votes for value
- compare the current total votes for value with some threshold

Figure 4.1: A Simplified Illustration of Consensus in Algorand The figure shows how Alice, a selected potential block leader or verifier, gossips her public key, *pk*, and a signed message with i.a. her proof of selection, π, and her chosen block hash, *value*. First, the receivers verify Alice's signed message and then her proof using the received *pk*. Next, they calculate her voting power j, count j votes for the block hash *value*, and compare the current total votes for *value* with some threshold.

Figure 4.1 shows a selected party, Alice, in a *step* of the BA algorithm, $BA\star()$. Alice, a committee member, selected as either a potential block leader or verifier, gossips her signed message containing i.a. her proof of selection, π, and her chosen block hash, *value*, together with her public key, *pk*. The receivers of Alice's gossiped message first verify both the signed message and the proof of selection using Alice's *pk* before calculating her voting power j. The receivers then (1) count j votes for the block hash, *value*, (2) update their received total votes for the block hash *value*, and (3) compare it to some threshold.

Going into more detail, each step, in the *Reduction()* and $BinaryBA\star()$ procedures, runs the procedures *CommitteeVote()* and *CountVotes()*, Algorithms 5.13 and 5.18. The *CommitteeVote()* procedure calls *Sortition()*, Algorithm 4.1, to evaluate whether or not the party is selected as a committee member. If chosen, the party gossips "a signed message containing the value passed to *CommitteeVote()*, which is typically the hash of some block" [29], e.g., their vote for some *value*. After some time, all parties run the *CountVotes()* procedure to count the votes for each *value* in the messages received. For each message received, *CountVotes()* calls the *ProcessMsg()*

procedure, Algorithm 5.16, which returns the votes, i.e., the j value, associated with this message. *ProcessMsg()* runs the procedure *VerifySort()*, Algorithm 5.15, which uses the sender's public key to verify the proof of selection, π, and to calculate j of the sender. *CountVotes()* maintains a table with the current total votes for each *value*, and if a *value* receives more votes than the threshold, it returns this *value*. The parties input the *value* returned from *CountVotes()* to the *CommitteeVote()* procedure in the next step, and if chosen as a committee member, they gossip (vote for) this *value*. The process continues as described in each step until consensus is reached.

In order to stimulate the adoption and growth of the Algorand Blockchain Network, monetary incentives in the form of a variety of rewards are distributed to stakeholders by a reward mechanism [40]. As outlined in [28], the initial reward mechanism distributes rewards accrued by an account in the Algorand network when a transaction involving that account is confirmed. In late November 2020, a proposal [42] was made for a new reward mechanism, in which parties commit to participating in some governance of the system. Thus, the transition to the new reward mechanism may already be underway, but no further documentation on the reward mechanism is available.

4.4.2 Analysis

The issue of privacy, explained in Section 4.1, arises in the PoS based cryptocurrency protocol, Algorand [29].

A vital privacy issue occurs in the selection process, in Algorithm 4.1, as the proof of selection is computed using the secret key of the chosen party (committee member) and requires their public key to verify the proof. As such, this selection process reveals the public key and thus the identity, of the committee members (the potential block leaders or verifiers) in each *step* of the BA algorithm. Thus, the proof verification (and creation) cannot be reliant upon revealing public keys if the identities of the parties are to remain hidden. The fact that the public key is gossiped along with the proof, also reveal the stake of the winner by frequency analysis.

The message is signed before it is gossiped along with the public key. Thus, the public key is not only needed for proof verification but “other users check that the signature is valid before relaying it” [29] in the gossip protocol. As the public key is needed to verify the signature, the identity of the sender is also revealed in the gossip protocol.

Another privacy issue of Algorand is the reliance upon the public weights, related to the stake, of all parties. The public context, *ctx*, consists of the parties' weights and, thus, reveals the stake of all the participating stakeholders. The weight is retrieved from *ctx* by the public key of the parties as $ctx.weight[user.pk]$ and is provided as input to the *Sortition()* and *VerifySort()* procedures. Even though the parties provide their own weight as input to the *Sortition()* procedure, their weight is public so that they cannot cheat in the lottery. For the *VerifySort()* procedure, receivers use the public weight of the senders to calculate each sender's j value, i.e., the number of selected sub-users (voting power) of the senders. Thus, to keep the weight private, this voting power must be calculated another way. Also, as j is related to the stake of the stakeholder, this should not be revealed to the receiving party.

Currently, there is little documentation publicly available detailing the initial reward mechanism beyond what can be found on Algorand's official website [40] and some blog posts [28, 41]. Due to the lack of details on the initial reward mechanism, we will refrain from analyzing the reward mechanism. With regards to the newly proposed reward mechanism [42], due to insufficient public details, we again conclude that we do not have a sufficient basis to complete a thorough analysis. Since we cannot complete an analysis of the reward mechanism, we will not attempt to make modifications to the reward mechanism in future sections. However, we stress that to have a PPoS, the distribution of block rewards must also preserve privacy.

The next chapter presents our two modified proposals that address Algorand's privacy issues identified above and the issues identified in the analysis of Baldimtsi et al.'s proposal [5] in Section 4.2.2.

Chapter 5
Proposals for Improvement

In this chapter, we present our two novel proposals for improvements to the proposal by Baldimtsi et al. [5]. As a proof of concept, we instantiate our proposals with Algorand [29]. Our first proposal, Proposal 1, aims to maintain the privacy of [5] while improving upon [5]'s performance in the multi-stake setting by using homomorphic encryption to remove the need for multiple unlinkable proofs per selected party. The second proposal, Proposal 2, aims to improve upon the same issue by providing a trade-off between performance and the privacy of the stake. We evaluate both proposals in terms of performance, privacy, and security.

5.1 Proposal 1

In the analysis of Baldimtsi et al.'s proposal [5], in Subsection 4.2.2, the need for multiple unlinkable proofs in the multi-stake setting is identified as an issue that significantly impacts performance and is further enhanced by the large proof size. Our proposal, Proposal 1, removes the need for multiple unlinkable proofs in the multi-stake setting, thereby improving the performance of [5]. Proposal 1 applies homomorphic encryption to the total voting power, j_i, making it possible to (1) send the total voting power along with a single proof and (2) keep count of votes without revealing the unencrypted voting power. Additionally, through the use of a homomorphic greater-than comparison, parties can tell when a threshold of votes is reached.

We start with a thorough overview of the scheme detailing how homomorphic encryption is used to realize the removal of multiple proofs. Next, we introduce ideal functionalities for the setup of the homomorphic encryption scheme and for a homomorphic greater-than comparison. Then, we modify [5]'s protocols to account for the inclusion of homomorphic encryption. Finally, we modify Algorand's [29] procedures to accommodate for the homomorphically encrypted voting power and [5]'s protocols, as modified by us.

We strongly recommend using the outline in Subsection 5.1.5 to aid with understanding while reading the proposal. Note that, throughout the chapter, the areas that require modification in the original protocols and procedures are marked in red. Similarly, modifications are marked in red in the modified protocols and procedures.

5.1.1 Scheme

Proposal 1 homomorphically encrypts the voting power, j_i, to remove the need for multiple unlinkable proofs in the multi-stake setting of Baldimtsi et al.'s proposal [5]. As a result, the proposal maintains both privacy and countability, while the voting power is associated with a single message, msg_i, and proof, π_i. While privacy is provided by the fact that j_i is homomorphically encrypted, countability is provided through a property of homomorphic encryption, which makes it possible to perform basic arithmetic operations on ciphertexts. Our proposal uses the BGV homomorphic encryption scheme by Brakerski et al. [11] and two homomorphic comparison tests, detailed in Section 3.1.

In greater detail, the idea behind our proposal is as follows. First, the voting power, j_i, of the prover is homomorphically encrypted, $FHE.Enc(j) = j'$, and inserted into the message, msg_i. Thus, msg_i is composed of the encrypted voting power, j'_i, the hash value of the previous block, *ctx.last_block*, and the block hash the prover wants to vote on, denoted *value*. Next, the prover generates a proof, π_i, that includes a zero-knowledge proof, π_{NIZK}, on the *tag* and message and gossips (tag, π_i, msg_i). Upon receiving (tag, π_i, msg_i), a receiver verifies the proof and, assuming that the proof is valid for the message and *tag*, adds the encrypted votes, j'_i, for *value* to a table, *counts*, used to keep count of the votes received for each block. The addition is accomplished by running $FHE.Eval()$ with j'_i and the current total votes for *value* in the *counts* table. The latter value is updated to the result of the addition.

To check if the current total votes for *value* have reached a threshold of votes a homomorphic greater-than comparison, $FHE.GreaterThan()$, Algorithm 3.1, is run to compare the votes for *value* with the threshold. In our instantiation, the threshold is defined by Algorand [29]. Note that we also make use of a homomorphic equality check [16], $FHE.Equality()$, Algorithm 3.2, within the zero-knowledge proof in order to account for a specific adversary attack covered in Subsection 5.1.3.4.

Figures 5.1 and 5.2 show a simplified part of Baldimtsi et al.'s proposal [5] and our modified proposal, respectively, with Algorand [29] as the underlying PoS protocol. The figures highlight differences in the gossiped message and how the votes are counted.

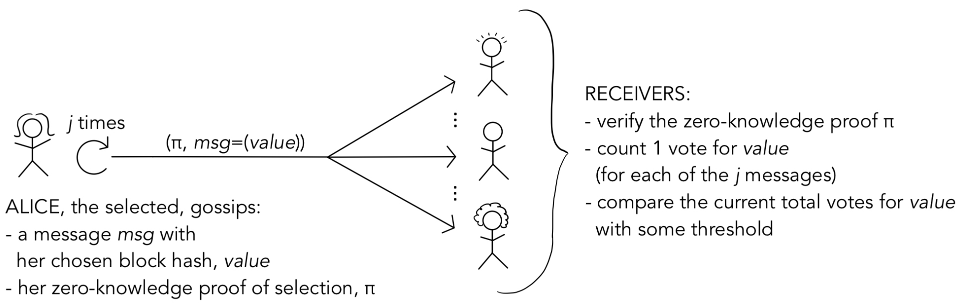
j times
(π, msg=(value))
ALICE, the selected, gossips:
- a message msg with her chosen block hash, value
- her zero-knowledge proof of selection, π
RECEIVERS:
- verify the zero-knowledge proof π
- count 1 vote for value (for each of the j messages)
- compare the current total votes for value with some threshold

Figure 5.1: A Simplified Illustration of Consensus in Baldimtsi et al.'s proposal [5] with Algorand [29] as the underlying PoS protocol in the multi-stake setting. The figure shows how Alice, a selected potential block leader or verifier, gossips her zero-knowledge proof of selection, π, together with a message, msg, containing her chosen block hash, $value$. First, the receivers verify Alice's zero-knowledge proof of selection, π. Then, they count one vote for the block hash $value$ and compare the current total votes for $value$ with some threshold. The receivers repeat this j times, for each of Alice's j gossiped messages.

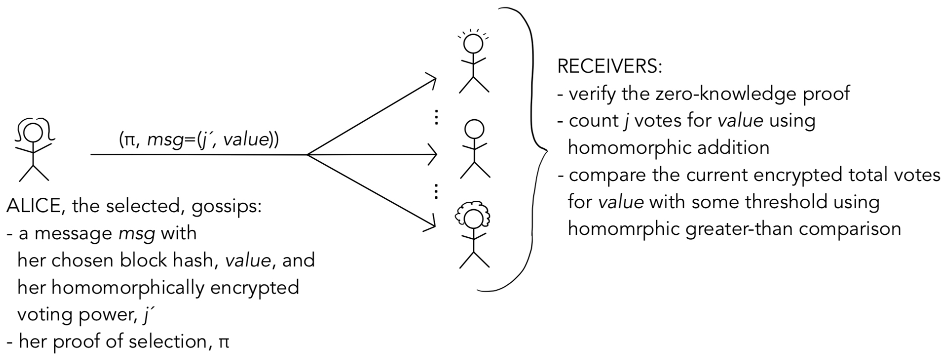
(π, msg=(j´, value))
ALICE, the selected, gossips:
- a message msg with her chosen block hash, value, and her homomorphically encrypted voting power, j´
- her proof of selection, π
RECEIVERS:
- verify the zero-knowledge proof
- count j votes for value using homomorphic addition
- compare the current encrypted total votes for value with some threshold using homomrphic greater-than comparison

Figure 5.2: A Simplified Illustration of Consensus in Proposal 1 with Algorand [29] as the underlying PoS protocol in the multi-stake setting. The figure shows how Alice, a selected potential block leader or verifier, gossips her zero-knowledge proof of selection, π, and a message, msg, containing her chosen block hash, $value$, and encrypted voting power, j'. First, the receivers verify Alice's zero-knowledge proof. Then, they count j votes for the block hash $value$ using homomorphic addition and compare the current encrypted total votes for $value$ with some threshold using homomorphic greater-than comparison.

In [5], as illustrated in Figure 5.1, Alice sends a zero-knowledge proof, π, and accompanying message, msg, with her chosen block hash, $value$, j times. In contrast, in our proposal, as illustrated in Figure 5.2, Alice only sends one zero-knowledge proof, π, accompanied by message, msg, additionally containing her homomorphically encrypted voting power, j'. Another difference is apparent on the receiver side when counting votes and comparing the current total votes for the block hash *value* to some threshold. Instead of using normal addition and a normal greater-than check as in [5], the receivers in Proposal 1 use homomorphic addition, $FHE.Eval()$, introduced in Subsection 3.1.2, and a homomorphic greater-than comparison, $FHE.GreaterThan()$, Algorithm 3.1.

5.1.2 Ideal Functionalities

For this scheme, we define and detail two ideal functionalities necessary to realize the setup for homomorphic encryption, *FHE.Setup()*, and the homomorphic greater-than comparison, *FHE.GreaterThan()*.

5.1.2.1 Ideal Functionality for Setup of Homomorphic Encryption

We decided to perform the setup, *FHE.Setup()*, and key generation, *FHE.KeyGen()*, necessary for homomorphic encryption within an ideal functionality to ensure that the public encryption key is shared and the secret decryption key is deleted. Having a shared public key increases the efficiency of the proposed scheme since it removes the need to call *FHE.Refresh()* before homomorphic additions and greater-than comparisons to ensure that the ciphertexts are under the same key. Thus, the ideal functionality mitigates the performance concerns of using *FHE.Refresh()*. Deleting the secret key is of the utmost importance because a shared secret decryption key would reveal voting powers. Since our application does not need decryption, the secret key, $FHE.sk_i$, is deleted by the ideal functionality after key generation to avoid related security concerns.

By design, the ideal functionality, $\mathcal{F}_{Setup}^{Homomorphic}$, Algorithm 5.1, runs *FHE.Setup()* to generate a ladder of parameters, denoted *params*, and *FHE.KeyGen()* to generate public and secret keys, $FHE.pk_i$ and $FHE.sk_i$, respectively. Both protocols are outlined in Subsection 3.1.1 and taken from [11]. The functionality stores $FHE.pk_i$, deletes $FHE.sk_i$, and outputs $FHE.pk_i$ upon query.

Algorithm 5.1 $\mathcal{F}_{Setup}^{Homomorphic}$

The ideal functionality is parameterized by the security parameter λ and the number of levels of arithmetic circuit wanted A. The functionality maintains the FHE public key $FHE.pk_i$.

```
Generate $params \leftarrow FHE.Setup(1^\lambda, 1^A)$
Generate $(FHE.sk_i, FHE.pk_i) \leftarrow FHE.KeyGen(params)$
Store $FHE.pk_i$ and $params$, and delete $FHE.sk_i$
 1: Upon receiving $(GetEncKey)$ from party $P_i$
 2: Output $FHE.pk_i$ and $params$ to $P_i$
```

5.1.2.2 Ideal Functionality for Homomorphic Greater-Than Comparison

Note that the greater-than comparison provided to participants should only give information about whether an encrypted value is greater than the threshold. I.e., it should not be possible to compare two chosen encrypted values as this would cause a security issue; the participant could compare two j_i' values or even a j_i' value with a chosen encrypted number, deducing the value of j_i. Thus, the $CountVotes()$ procedure calls an ideal functionality for the homomorphic greater-than comparison, giving the current votes for a block hash ($value$), i.e., only one chosen encrypted value as input (see line 15 in Algorithm 5.19). This ideal functionality carries out the comparison by comparing the votes to the threshold and returns true or false (1 or 0). Thus, the participants are not able to compare two chosen values.

Algorithm 5.2 $\mathcal{F}_{Greater-Than}^{Homomorphic}$

The ideal functionality is parameterized by the number of levels of arithmetic circuit wanted A, a ladder of parameters $params$, a fraction of the expected committee size T, and the expected number of parties selected for the committee τ.

```
Upon receiving $(counts'[value])$ from party $P_i$
  1: $x' \leftarrow counts'[value]$
  2: $y' \leftarrow FHE.Enc(params, T \cdot \tau + 1)$
  3: Call $FHE.GreaterThan(x', y')$        ▷ returns $b = 1$ if $x > y$, and $b = 0$ otherwise
  4: Output $b$ to party $P_i$
```

5.1.3 Modifications to the Protocols of Baldimtsi et al.

This section covers our modifications to the protocols proposed by Baldimtsi et al. [5] in the paper *Anonymous Lottery In the Proof-of-Stake setting.* Note that the *Initialization()* protocol [5, p. 12] is not reviewed as no modifications are necessary. We refer to Section 2.2 and the paper [5] for more details on the original protocols.

5.1.3.1 Anonymous Selection Protocol

Algorithm 5.3 shows Baldimtsi et al.'s [5] original *Anonymized Selection Protocol*, $\Pi_{Anon-Selection}^{Eligible}$, for the single-stake setting. This overall protocol shows the order in which a party, P_i, calls the protocols *Initialization()*, *EligibilityCheck()*, *CreateProof()*, and *Verify()*.

Although the overall protocol, Algorithm 5.3, is only provided in the single-stake setting, Baldimtsi et al. explain that adaptation to the multi-stake setting is possible by replacing b_{tag} with weight, wt_i, and creating a zero-knowledge proof for each unit of the weight, i.e., for each $index$. Each proof accompanies a msg_i and tag and has a voting power of one. We show modifications from the single-state setting protocol, Algorithm 5.3, to our modified multi-stake setting protocol, Algorithm 5.4.

Algorithm 5.3 $\Pi^{Eligible}_{Anon-Selection}$

```
A party $P_i$ executes the protocol $\Pi^{Eligible}_{Anon-Selection}$ in the following way:
1: Call Initialization($P_i, sid$) to get $(pk_i, sk_i)$
2: To publish a message $msg_i$ in tag :
3:     Call EligibilityCheck($P_i, sid$,tag) to get
       $b_{tag}$,$\vec{V}_{tag}$ and $v_i$.
4:     if $b_{tag} = 1$ then
           call CreateProof($P_i, sid, msg_i$,tag, $v_i, \vec{V}_{tag}$)
           to get $\pi_i$
5:     Output ($msg_i$,tag, $\pi_i$)
6: To verify a message($msg$,tag, $\pi$) in tag:
7:     Call Verify($sid$,tag, $\pi$)
       and output the bit it returns
```

As explained in Section 5.1.1, we want to create a single proof for the total voting power, j_i, of a party, P_i. Note that j_i corresponds to [5]'s weight, wt_i. Similar to [5]'s adaptation to the multi-stake setting, we replace the instances of b_{tag}. The first instance, on line 3 of Algorithm 5.3, is replaced with j_i in Algorithm 5.4. Similarly, on line 4, we evaluate if a party, P_i, has voting power by replacing $b_{tag} = 1$ with $j_i > 0$. To simplify the presentation of both this proposal and Proposal 2, we remove P_i and sid as input to the protocols on lines 1, 3, 4, and 7. Thus, we do not have P_i and sid as input in any of the following protocols by [5]. However, note that all the following algorithms, both the modified [5] protocols and the modified Algorand procedures in both proposals, are run by a party P_i.

Algorithm 5.4 $\Pi^{Eligible}_{Anon-Selection}$ [1]

```
A party $P_i$ executes the protocol $\Pi^{Eligible}_{Anon-Selection}$ in the following way:
1: Call Initialization() to get $(pk_i, sk_i)$
2: To publish a message $msg_i$ in $tag$ :
3:     Call EligibilityCheck($tag$) to get
       $j_i$,$\vec{V}_{tag}$ and $v_i$.
4:     if $j_i > 0$ then
           call CreateProof($msg_i, tag, v_i, \vec{V}_{tag}, j_i$)
           to get $\pi_i$
5:     Output ($msg_i, tag, \pi_i$)
6: To verify a message($msg, tag, \pi$) in $tag$:
7:     Call Verify($tag, \pi$)
       and output the bit it returns
```

5.1.3.2 Setup()

Before the overall protocol, Algorithm 5.4 commences, setup is required. First, $Setup(1^\lambda)$ is run as detailed in [5, p. 11]. Next, our ideal functionality, $\mathcal{F}_{Setup}^{Homomorphic}$, Algorithm 5.1, performs the computations necessary to enable homomorphic encryption in the system. The parties query the ideal functionality to acquire the public key for homomorphic encryption, $FHE.pk_i$, and a ladder of parameters, $params$, used as input for homomorphic encryption.

5.1.3.3 EligibilityCheck() and Eligible()

The *Eligible()* protocol by Baldimtsi et al. [5] is called by a party to determine if they are eligible to speak for tag, i.e., to determine if they are selected to vote on or propose a block. In the original protocol, *Eligible()* calculates and returns the weight wt_i. If $wt_i > 0$, the party is selected for tag. Following Algorand's [29] naming, we change the weight, wt_i, to the voting power, j_i, as seen in our modified *Eligible()* protocol, Algorithm 5.5.

Algorithm 5.5 Eligible()^1

```
Protocol Eligible{$v_i, stake_i$,tag}
1: $p \leftarrow \frac{\tau}{totalStake}$
2: $j_i \leftarrow 0$
3: while $2^{\frac{v_i}{len(v_i)}} \notin \left[\sum_{k=0}^{j_i} B(k;w,p), \sum_{k=0}^{j_i+1} B(k;w,p)\right]$
4:     do $j_i \leftarrow j_i + 1$
5: return $j_i$
```

In accordance with the changes in *Eligible()*, the modified version of *Eligibility-Check()*, Algorithm 5.6, has a variable name change from wt_i to j_i. As the name change from wt_i to j_i is the only modification in *Eligible()* and *EligibilityCheck()*, we only show the modified protocols.

Algorithm 5.6 EligibilityCheck()^1

```
Protocol EligibilityCheck($tag$)
1: Call $ProcessRO(tag)$ and receive $\vec{V}_{tag}$
2: Compute $v_i = f_{TRP.sk_i}^{-1}(\vec{V}_{tag}[i])$
3: Call $Eligible(v_i, stake_i, tag)$ and receive $j_i$
4: return $j_i, v_i, \vec{V}_{tag}$
```

5.1.3.4 Zero-knowledge proof statements

The inclusion of a homomorphically encrypted voting power associated with only one proof requires modifications to the zero-knowledge proof statements in Baldimtsi et al.'s proposal [5]. Below, the original statement x, the witness w, and the twelve statements to check are shown, and the parts we will modify are marked in red.

$$\pi \leftarrow NIZK.Prove(crs, x, w)$$

- **Statement** $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg, C^{v}_{i,index}, \vec{V}_{tag})$
- **Witness** $w = (i, wt_i, stake_i, index, PRF_{sk_i}, v_i, \sigma, s_{prf}, pk_i, path_{pk}, path_{\vec{V}_{tag}}, path_{cm}, cm_i)$ where $pk_i = (TRP_{pk_i}, SIG.vk_i, C_{prf})$

  R(x,w)=1 if and only if:

  1. $C^{v}_{i,index} = F(PRF.sk_i, v_i||tag||index)$
  2. $index \in [1, wt_i]$
  3. $C^{prf}_{i} = Com(PRF.sk_i; s_{prf})$
  4. $cm_i = Com(stake_i)$
  5. $V_i = f_{TRP.pk_i}(v_i)$
  6. $V_i = \vec{V}_{tag}[i]$
  7. $Eligible(v_i, stake_i, tag) = wt_i$
  8. $\sigma = SIG.Sign(SIG.sk_i, msg||tag)$
  9. $SIG.Ver(SIG.vk_i, \sigma, msg||tag) = 1$
  10. $validPath_h(path_{pk}, rt_{pk}, pk_i) = 1$
  11. $validPath_h(path_{\vec{V}_{tag}}, rt_{\vec{V}_{tag}}, \vec{V}_{tag}[i]) = 1$
  12. $validPath_h(path_{cm}, rt_{cm}, cm_i) = 1$

Notably, the *index* variable is obsolete as we only give one proof associated with the total voting power. Therefore, the *index* variable in the witness, the *index* variables in line 1, and the entire line 2 are removed. Also, note that the total voting power, wt_i, is renamed j_i. This name change is visible in the witness, in line 7 above, and in line 6 in the modified statements below.

Line 7, colored red in the modified statements below, is a new statement added to account for a dishonest selected party, i.e., an adversary that changes j_i' before gossiping. The integrity of j_i from the witness, w, is preserved because of the check on line 6 below, as with Baldimtsi et al.'s original proposal. Thus, j_i is encrypted and compared with j'_i, retrieved from the received message as $msg[2]$. The comparison is made using *FHE.Equality()* to confirm that j'_i was unaltered before gossiping. Unlike the homomorphic greater-than comparison, the homomorphic equality comparison does not need to be in an ideal functionality because it is called within the zero-knowledge proof, making it impossible for an adversary to call *FHE.Equality()* with any other value than those specified. $params$, a new variable in x, is needed for the homomorphic encryption on line 7 below.

Although lines 8 and 9 look unaltered, the signature provides integrity of the encrypted voting power, j'_i, during gossip as this is part of the message, msg_i, included in the signature. Note that Merkle trees are central to the built upon proposal [5]. The vector $\vec{V}_{tag}$, the public keys, and commitments to the parties stakes are stored as Merkle trees with roots $rt_{\vec{V}_{tag}}$, rt_{pk}, and rt_{cm}, respectively. We refer to [5] for more details.

$$\pi \leftarrow NIZK.Prove(crs, x, w)$$

– **Statement** $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C_i^v, \vec{V}_{tag}, params)$

– **Witness** $w = (i, j_i, stake_i, PRF_{sk_i}, v_i, \sigma, s_{prf}, pk_i, path_{pk}, path_{\vec{V}_{tag}}, path_{cm}, cm_i)$ where $pk_i = (TRP_{pk_i}, SIG.vk_i, C_{prf})$

R(x,w)=1 if and only if:

1. $C_i^v = F(PRF.sk_i, v_i||tag)$
2. $C_i^{prf} = Com(PRF.sk_i; s_{prf})$
3. $cm_i = Com(stake_i)$
4. $V_i = f_{TRP.pk_i}(v_i)$
5. $V_i = \vec{V}_{tag}[i]$
6. $Eligible(v_i, stake_i, tag) = j_i$
7. $FHE.Equality(msg_i[2], FHE.Enc(params, FHE.pk_i, j_i)) = 1$
8. $\sigma = SIG.Sign(SIG.sk_i, msg_i||tag)$
9. $SIG.Ver(SIG.vk_i, \sigma, msg_i||tag) = 1$
10. $validPath_h(path_{pk}, rt_{pk}, pk_i) = 1$
11. $validPath_h(path_{\vec{V}_{tag}}, rt_{\vec{V}_{tag}}, \vec{V}_{tag}[i]) = 1$
12. $validPath_h(path_{cm}, rt_{cm}, cm_i) = 1$

5.1.3.5 CreateProof()

The changes made above to the zero-knowledge proof require changes in the *CreateProof()* protocol. As Baldimtsi et al. [5] only define this protocol explicitly in the single-stake setting, the "original" *CreateProof()* protocol in Algorithm 5.7 is written by us accounting for the changes described for the multi-stake setting in their paper.

Algorithm 5.7 CreateProof()

```
Protocol CreateProof($P_i, sid, msg_i, tag, v_i, \vec{V}_{tag}, wt_i$)
 1: Let $rt_{\vec{V}_{tag}}$ be the root of $MTree(\vec{V}_{tag})$
 2: Let $path_{\vec{V}_{tag}[i]}$ be the path to $\vec{V}_{tag}[i]$ in $MTree(\vec{V}_{tag})$
 3: Let $rt_{pk}$ be the root of $MTree(pk)$
 4: Let $path_{pk_i}$ be the path to $pk_i$ in $MTree(pk)$
 5: Let $rt_{cm}$ be the root of $MTree(cm)$
 6: Let $path_{cm}$ be the path to $cm_i$ in $MTree(cm)$
 7: Compute $\sigma_i = SIG.Sign(SIG.sk_i, msg_i||tag)$
 8: $\pi_i = []$
 9: for each $index \in [1, wt_i]$
10:     Compute $C^v_{i,index} = F(PRF.sk_i, v_i||tag||index)$
11:     Let $x_{i,index} = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C^v_{i,index}, \vec{V}_{tag})$
12:     Let $w_{i,index} = (i, wt_i, stake_i, index, PRF.sk_i, v_i, \sigma_i, pk_i, path_{pk_i}, path_{\vec{V}_{tag}[i]}, path_{cm}, cm_i)$
13:     Compute $\pi_{NIZK,index} := NIZK.Prove(crs, x_{index}, w_{i,index})$
14:     Set $\pi_{i,index} := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C^v_{i,index}, \pi_{NIZK,index})$
15:     Add $\pi_{i,index}$ to $\pi_i$
16: Output $\pi_i$
```

Algorithm 5.8 CreateProof()[1]

```
Protocol CreateProof($msg_i, tag, v_i, \vec{V}_{tag}, j_i, params$)
 1: Compute $C^v_i = F(PRF.sk_i, v_i||tag)$
 2: Let $rt_{\vec{V}_{tag}}$ be the root of $MTree(\vec{V}_{tag})$
 3: Let $path_{\vec{V}_{tag}[i]}$ be the path to $\vec{V}_{tag}[i]$ in $MTree(\vec{V}_{tag})$
 4: Let $rt_{pk}$ be the root of $MTree(pk)$
 5: Let $path_{pk_i}$ be the path to $pk_i$ in $MTree(pk)$
 6: Let $rt_{cm}$ be the root of $MTree(cm)$
 7: Let $path_{cm}$ be the path to $cm_i$ in $MTree(cm)$
 8: Compute $\sigma_i = SIG.Sign(SIG.sk_i, msg_i||tag)$
 9: Let $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C^v_i, \vec{V}_{tag}, params)$
10: Let $w = (i, j_i, stake_i, PRF.sk_i, v_i, \sigma_i, pk_i, path_{pk_i}, path_{\vec{V}_{tag}[i]}, path_{cm}, cm_i)$
11: Compute $\pi_{NIZK} := NIZK.Prove(crs, x, w)$
12: Set $\pi_i := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C^v_i, \pi_{NIZK})$
13: Output $\pi_i$
```

The modified *CreateProof()* protocol is shown in Algorithm 5.8. We change the name for the voting power from wt_i to j_i. Then, as only one proof per j_i is needed for each selected party, i.e., not one proof per $index \in [1, j_i]$, the for-loop and the *index* variables are removed. As a result, our modified version, Algorithm 5.8, more closely resembles the single-stake setting version in [5, p. 14]. The new input variable *params* is needed for the zero-knowledge proof and provided in the statement x on line 9.

5.1.3.6 Verify()

Baldimtsi et al. only provide the *Verify()* procedure in the single-stake setting in [5, p. 14]. Thus, the "original" Verify() protocol, Algorithm 5.9, is based on their single-stake setting protocol with the changes described for the multi-stake setting, For this, the addition of rt_{cm} on lines 2 and 3 was needed.

Algorithm 5.9 Verify()

```
Protocol Verify($sid$,tag,$msg, \pi$)
1: Call $ProcessRO(tag)$ and receive $\vec{V}_{tag}$
2: Parse $\pi = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C, \pi_{NIZK})$
3: Set $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg, C, \vec{V}_{tag})$
4: Check that $NIZK.Verify(crs, x, \pi_{NIZK}) =?1$
5: If yes, output 1; else output 0
```

The first modification necessary to implement our scheme is the addition of the *params* parameter as input to both the *Verify()* procedure and zero-knowledge proof statement x on line 3, as seen in Algorithm 5.10. Note that even though no modifications are visible, *NIZK.Verify()*, on line 4, now verifies the statements in the modified zero-knowledge proof in Subsection 5.1.3.4.

Algorithm 5.10 Verify()^1

```
Protocol Verify($sid, tag, msg, \pi, params$)
1: Call $ProcessRO(tag)$ and receive $\vec{V}_{tag}$
2: Parse $\pi = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C, \pi_{NIZK})$
3: Set $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg, C, \vec{V}_{tag}, params)$
4: Check that $NIZK.Verify(crs, x, \pi_{NIZK}) =?1$
5: If yes, output 1; else output 0
```

5.1.4 Modifications to the Procedures of Algorand

This section details the modifications necessary in Algorand's [29] procedures to incorporate the use of homomorphic encryption and our modified Baldimtsi et al.'s [5] protocols from Subsection 5.1.3. We also consider the implication of these modifications for Algorand's *CommonCoin()* procedure. We refer to Subsection 4.4.1 and the paper [29] for more details on the original procedures.

5.1.4.1 Sortition()

Algorand's original *Sortition()* procedure from [29, p. 56] is shown in Algorithm 5.11.

Algorithm 5.11 Sortition()

```
1: procedure SORTITION($sk, seed, \tau, role, w, W$)
2:     $\langle hash, \pi \rangle \leftarrow VRF_{sk}(seed||role)$
3:     $p \leftarrow \frac{\tau}{W}$
4:     $j \leftarrow 0$
5:     while $2^{\frac{hash}{hashlen}} \notin \left[\sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p)\right]$
6:         do $j \leftarrow j + 1$
7:     return $\langle hash, \pi, j \rangle$
8: end procedure
```

The modified *EligibilityCheck()* protocol, Algorithm 5.6, is implemented in the modified *Sortition()* procedure, Algorithm 5.12, and replaces the calculation of voting power, j_i, in lines 3 through 6 of Algorithm 5.11. Note that the *Eligible()* protocol, Algorithm 5.5, does this calculation within *EligibilityCheck()*. The calculation in *Eligible()* adapted from [5] is similar to the original calculation in Algorand's *Sortition()* procedure except for replacing *hash* with v_i and *hashlen* with $len(vi)$. The modified *Sortition()* does not return *hash* (or its replacement v_i), as it is no longer gossiped in the *CommitteVote()* procedure, as detailed in Subsection 5.1.4.2.

In our modified *Sortition()* procedure, Algorithm 5.12, the modified *CreateProof()* protocol, Algorithm 5.8, replaces the VRF to create a zero-knowledge proof of selection. See line 2 in the original procedure and line 8 in the modified procedure, Algorithms 5.11 and 5.12, respectively. In contrast to the original *Sortition()* procedure, a (zero-knowledge) proof of selection is only created if the party was selected, i.e., if the party has voting power $j_i > 0$, as seen in lines 5-8 of Algorithm 5.12. In the case where $j_i = 0$, j_i' and the proof remain 0 and *null*, respectively.

Notably, the modified version performs homomorphic encryption on j_i, as seen in line 6 in Algorithm 5.12. Then, the homomorphically encrypted voting power, j_i', is provided as input to the *CreateProof()* protocol, Algorithm 5.8, and returned by the modified *Sortition()* procedure, both as part of msg_i.

Algorithm 5.12 Sortition()[1]

```
1: procedure SORTITION($value, tag, params$)
2:     $\left\langle j_i, v_i, \vec{V}_{tag} \right\rangle \leftarrow EligibilityCheck(tag)$
3:     $\pi_i \leftarrow null$
4:     $j'_i \leftarrow 0$
5:     if $j_i > 0$ then
6:         $j'_i = FHE.Enc(params, FHE.pk, j_i)$
7:         $msg_i = (H(ctx.last\_block), value, j'_i)$
8:         $\pi_i \leftarrow CreateProof(msg_i, tag, v_i, \vec{V}_{tag}, j_i, params)$
9:     return $\langle \pi_i, msg_i \rangle$
10: end procedure
```

Additionally, the original input variable W is renamed $totalStake$ as in Baldimtsi et al.'s proposal [5]. totalStake and the input variable τ are global variables and are thus no longer required as input. The *Eligible()* protocol, Algorithm 5.5, uses them to calculate j_i. Moreover, the input variable *role* is no longer needed as the VRF is removed, and the original input variable *seed* is part of the new input variable *tag* instead. The input variables, $tag = (round, step, seed)$ and *value*, as part of msg_i, are needed as input to the *CreateProof()* procedure in line 8 in Algorithm 5.12, while *params* is needed for the homomorphic encryption on line 6.

5.1.4.2 CommitteeVote()

The original *CommitteVote()* procedure, Algorithm 5.13, calls *Sortition()*, Algorithm 5.11, to evaluate whether the party is selected to participate in the committee. If chosen, if $j_i > 0$, the party gossips "a signed message containing the value passed to CommitteeVote(), which is typically the hash of some block" [29], i.e., their vote for some *value*.

Algorithm 5.13 CommitteeVote()

```
1: procedure COMMITTEEVOTE($ctx, round, step, \tau, value$)
2:     // check if user is in committee using Sortition
3:     $role \leftarrow \langle "committee", round, step \rangle$
4:     $\langle sorthash, \pi, j \rangle \leftarrow Sortition(user.sk, ctx, seed, \tau, role, ctx.weight[user.pk], ctx.W)$
5:     // only committee members originate a message
6:     if $j > 0$ then
7:         $Gossip(\langle user.pk, Signed_{user.sk}(round, step, sorthash, \pi, H(ctx.last\_block), value) \rangle)$
8: end procedure
```

In our modified version, Algorithm 5.14, the encrypted voting power, j'_i, acquired from the modified *Sortition()* procedure, Algorithm 5.12, as part of msg_i, is gossiped along with the proof, π_i, and *tag*. In contrast, in Algorand, originally, j_i is not gossiped but calculated by the receiver using *VerifySort()*, Algorithm 5.15. However, this is not possible in our privacy-preserving proposal as it reveals the voting power, j_i, and consequently the stake of the selected to the receiver. Also, in Baldimtsi et al.'s [5] calculation of j_i, the variable v_i is used. They state that "we are required to hide the value v_i so that the identity of the party is not revealed" [5]. Thus, the receiver cannot use this variable.

Also, note that the proof and message are gossiped only if the selected has voting power in both the original and modified procedure. Originally, in Algorithm 5.13, this is done with the check "if $j > 0$" on line 6. In the modified version, Algorithm 5.14, the check "if $msg_i[2] \neq 0$", corresponds to the check"if $j_i' \neq 0$".

Additionally, the message, msg_i, is no longer signed before gossip, as seen on lines 7 and 6 in Algorithms 5.13 and 5.14, respectively. Instead, it is part of the zero-knowledge proof, as noted in Subsection 5.1.3.4. Specifically, the proof, π_i, tag, and $msg_i = (H(ctx.last\_block), value, j_i')$ are gossiped. The signature in the zero-knowledge proof, $SIG.Sign(SIG.ski, msg_i||tag)$, is a signature on the message, msg_i, and tag. Thus, the integrity of these two variables, and the variables within msg_i, are verified when verifying the zero-knowledge proof. The proof is not valid if any of these variables are tampered with during gossip.

Algorithm 5.14 CommitteeVote()[1]

```
1: procedure CommitteeVote($ctx, tag, value, params$)
2:     // check if user is in committee using Sortition
3:     $\langle \pi_i, msg_i \rangle \leftarrow Sortition(value, tag, params)$
4:     // only committee members originate a message
5:     if $msg_i[2] \neq 0$ then
6:         $Gossip(\langle tag, \pi_i, msg_i \rangle)$
7: end procedure
```

Other changes include using the variable $tag = (round, step, seed)$ instead of the two variables $step$ and $round$, removing the input variable τ, and adding $params$. Also, when the modified *Sortition()* procedure, Algorithm 5.12, is called, the input to this procedure is changed accordingly. Finally, $role$ is removed since it is no longer needed in *Sortition()*.

5.1.4.3 VerifySort()

The receivers call Algorand's *VerifySort()* procedure to get the voting power, j_i, of the received proofs and corresponding messages. As seen in Algorithm 5.15, this effectively reveals the voting power of the sender. As discussed in Section 5.1.4.2, adapting Baldimtsi et al.'s [5] privacy-preserving scheme, j_i cannot be calculated in *VerifySort()* by the receiver since j_i is to be kept secret as it is proportional to the sender's stake. In addition, the $hash$ variable's replacement v_i reveals the identity of the prover. Thus, the encrypted voting power, j_i', is instead gossiped to the receiver as shown in the modified *CommitteVote()* procedure, Algorithm 5.14. As such, we remove the calculation of j_i and are left with only lines 1, 2, and 8 of Algorithm 5.15.

The *VerifySort()* procedure now only consists of the $VerifyVRF_{pk}()$ procedure, which we replace with the modified *Verify()* procedure, Algorithm 5.10, to verify the zero-knowledge proof. This modification accounts for the change in proof creation in Subsection 5.1.4.1. We choose to remove *VerifySort()* entirely and replace it with *Verify()* as this is the only functionality left. The result of this change will be made more apparent in the next section, Subsection 5.1.4.4, which covers the modifications done to Algorand's *ProcessMsg()* procedure.

Algorithm 5.15 VerifySort()

```
1: procedure VerifySort($pk, hash, \pi, seed, \tau, role, w, W$)
2:     if $\neg VerifyVRF_{pk}(hash, \pi, seed||role)$ then return 0
3:     $p \leftarrow \frac{\tau}{W}$
4:     $j \leftarrow 0$
5:     while $2^{\frac{hash}{hashlen}} \notin \left[\sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p)\right]$
6:         do $j \leftarrow j + 1$
7:     return $j$
8: end procedure
```

5.1.4.4 ProcessMsg()

The *ProcessMsg()* procedure, Algorithm 5.16, is run for each received message when counting votes. Originally, the procedure returns, i.a. the voting power, j_i, named *votes*, and the hash of the block the sender votes for or proposes, *value*.

Algorithm 5.16 ProcessMsg()

```
1: procedure ProcessMsg($ctx, \tau, m$)
2:     $\langle pk, signed\_m \rangle \leftarrow m$
3:     if $VerifySignature(pk, signed\_m) \neq$ OK then
4:         return $\langle 0, \perp, \perp \rangle$
5:     $\langle round, step, sorthash, \pi, hprev, value \rangle \leftarrow signed\_m$
6:     // discard messages that do not extend this chain
7:     if $hprev \neq H(ctx.last\_block)$ then return $\langle 0, \perp, \perp \rangle$
8:     $votes \leftarrow$ VerifySort($pk, sorthash, \pi, ctx.seed, \tau,$
9:                 $\langle$committee$, round, step \rangle, ctx.weight[pk], ctx.W$)
10:    return $\langle votes, value \rangle$
11: end procedure
```

As the gossiped variables in m_i have changed in the modified version of *CommitteeVote()*, Algorithm 5.14, the variables retrieved from m_i in the modified version of *ProcessMsg()* change. Thus, lines 2 and 5 in Algorithm 5.16 are replaced with lines 2 and 3 in Algorithm 5.17. Also, as the message sent does not contain a signed message, lines 3 and 4 verifying the signature in Algorithm 5.16 are removed. Replacing the signature of the gossiped message with a signature in the zero-knowledge proof as discussed in Section 5.1.4.2.

Furthermore, as previously mentioned in Section 5.1.4.3, the *VerifySort()* procedure has been replaced by Baldimtsi et al.'s [5] *Verify()* protocol. Lines 8 and 9 in Algorithm 5.16 are replaced by line 6 in Algorithm 5.17. *Verify()* returns the encrypted voting power, j'_i, named $votes'$. If the poof is not valid, $votes'$ is set to 0. The encrypted voting power $votes'$ is returned instead of $votes$ in plaintext.

Algorithm 5.17 ProcessMsg()[1]

```
1: procedure ProcessMsg($ctx, m_i$)
2:     $\langle tag, \pi_i, msg_i \rangle \leftarrow m_i$
3:     $\left\langle hprev, value, j'_i \right\rangle \leftarrow msg_i$
4:     // discard messages that do not extend this chain
5:     if $hprev \neq H(ctx.last\_block)$ then return $\langle 0, \perp, \perp \rangle$
6:     if $Verify(tag, msg_i, \pi_i)$ then $votes' \leftarrow j'_i$
7:     else $votes' \leftarrow 0$
8:     return $\langle votes', value \rangle$
9: end procedure
```

Also, the variable τ is no longer needed and thus not provided as input. Lastly, the variable *sorthash* is not returned because Baldimtsi et al.'s [5] replacement of *sorthash*, v_i, must be kept secret, as mentioned in Subsections 5.1.4.2 and 5.1.4.3. This replacement also has consequences for the *CommonCoin()* procedure, which originally uses this output from *ProcessMsg()*, discussed in Section 5.1.4.6.

5.1.4.5 CountVotes()

The *CountVotes()* procedure is called to count all votes for the block hashes in the received messages. The procedure calls *ProcessMsg()* for each received message to get the block hash and associated votes for each message.

Algorithm 5.18 CountVotes()

```
1: procedure CountVotes($ctx, round, step, T, \tau, \lambda$)
2:     $start \leftarrow Time()$
3:     $counts \leftarrow \{\}$                         ▷ hash table, new keys mapped to 0
4:     $voters \leftarrow \{\}$
5:     $msgs \leftarrow incomingMsgs[round, step].iterator()$
6:     while TRUE do
7:         $m \leftarrow msgs.next()$
8:         if $m = \perp$ then
9:             if $Time() > start + \lambda$ then return $TIMEOUT$
10:        else
11:            $\langle votes, value, sorthash \rangle \leftarrow ProcessMsg(ctx, \tau, m)$
12:            if $pk \in voters$ or $votes = 0$ then continue;
13:            $voters \cup = \{pk\}$
14:            $counts[value] += votes$
15:            // if we got enough votes, then output this value
16:            if $counts[value] > T \cdot \tau$
17:                return $value$
18: end procedure
```

The modified version of *ProcessMsg()* returns the encrypted voting power, j_i', named $votes'$, on line 11 in Algorithm 5.19. Moreover, the *counts* table, renamed *counts*, now stores the current total votes for each block hash in the encrypted form. The addition of new votes, *votes*, to the *counts* table for the relevant block hash is modified to carry out the addition of $votes'$ with the relevant homomorphically encrypted votes in the *counts* table, see lines 14 and 13 in Algorithms 5.18 and 5.19, respectively.

Regarding homomorphic addition, recall that we hypothesize it is possible to define the level A as the maximum amount of additions necessary for counting votes to achieve consensus in Algorand in Subsection 3.1.2.1. With this hypothesis, and because of a shared public encryption key as mentioned in Subsection 5.1.2.1, we completely remove the need for *FHE.Refresh()* and its negative effects on performance.

Further modifications include replacing the greater-than comparison with a call to our ideal functionality that carries out a homomorphic greater-than comparison, Algorithm 5.2, as seen in lines 16 and 15-16 in Algorithms 5.18 and 5.19, respectively. The ideal functionality returns true or false (1 or 0).

Algorithm 5.19 CountVotes()[1]

```
1: procedure CountVotes($ctx, tag, T, \lambda$)
2:     $start \leftarrow Time()$
3:     $counts' \leftarrow \{\}$            ▷ hash table, new keys mapped to $FHE.Enc(params, FHE.pk_i, 0)$
4:     $voters \leftarrow \{\}$
5:     $msgs \leftarrow incomingMsgs[tag].iterator()$
6:     while TRUE do
7:         $m \leftarrow msgs.next()$
8:         if $m = \perp$ then
9:             if $Time() > start + \lambda$ then return $TIMEOUT$
10:        else
11:            $\langle votes', value \rangle \leftarrow ProcessMsg(ctx, m)$
12:            if $votes' = 0$ then continue;
13:            $counts'[value] = FHE.Eval(FHE.pk, f, counts'[value], votes')$
14:            // if we got enough votes, then output $value$
15:            Query $(counts'[value])$ to the ideal functionality $\mathcal{F}_{Greater-Than}^{Homomorphic}$
16:            Receive $b$ from the ideal functionality $\mathcal{F}_{Greater-Than}^{Homomorphic}$
17:            if $b$ return $value$
18: end procedure
```

Other changes include removing the now global τ from lines 1 and 11 and removing *sorthash* on line 11 as this is no longer returned from *ProcessMsg()*. Additionally, the variable *tag* replaces the variables *round* and *step* on lines 1 and 5. Finally, a part of the if-check on line 12, $pk \in voters$, and line 13, $voters \cup = \{pk\}$, is removed as the public key of the sender is secret.

5.1.4.6 CommonCoin()

The *CommonCoin()* procedure from [29, p. 61] is executed in the event that the *CountVotes()* procedure times out. *CommonCoin()* ensures that a certain type of attack is not possible through the use of a “common coin”, as explained thoroughly in [29, p. 59-61]. For the purpose of this thesis, we will not delve into details regarding such an attack; however, we will note the modifications necessary for *CommonCoin()* to comply with our proposal. Algorand’s *CommonCoin()* makes use of the variable *sorthash*, originally returned from the *ProcessMsg()* procedure, Algorithm 5.16, but removed in the modified procedure, Algorithm 5.17, in Section 5.1.4.4 as its replacement, v_i, reveals the identity of the selected.

One possible solution to replace *sorthash* is for the selected to accompany a random hash in msg_i before gossiping. This hash should be new for each *tag* if selected. Thus, the hash is random, unique for each (selected) party, and updated for each new *tag*, similar to *sorthash*. Additionally, it does not identify the selected. This solution is not implemented in the above modified protocols as it complicates the presentation of our scheme.

5.1.5 Outline

This section provides a supplementary explanation of how our modified protocols and procedures from Sections 5.1.3 and 5.1.4, respectively, fit together in an instantiation of Algorand. For clarity, the instantiation assumes an honest party, P_i, who possesses voting power for *tag*, and an honest party, P_h, who counts the votes in the message received from P_i. The following sequence of events takes place after setup.

1. Party, P_i, calls $ComitteeVote()$, Algorithm 5.14, with inputs tag and $value$.
2. $CommitteeVote()$ calls $Sortition()$, Algorithm 5.12.
3. $Sortition()$ calls $EligibilityCheck()$, Algorithm 5.6.
4. $EligibilityCheck()$ receives the trapdoor permutation, $\vec{V}_{tag}$, from $ProcessRO()$, computes its inverse, v_i, and calls $Eligible()$, Algorithm 5.5.
5. $Eligible()$ calculates the voting power, j_i, and returns it to $EligibilityCheck()$.
6. $EligibilityCheck()$ receives j_i and returns $(j_i, v_i, \vec{V}_{tag})$ to $Sortition()$.
7. $Sortition()$ receives $(j_i, v_i, \vec{V}_{tag})$, applies homomorphic encryption to j_i yielding j_i', and composes the message, $msg_i = (H(ctx.last\_block), value, j_i')$.
8. $Sortition()$ calls $CreateProof()$, Algorithm 5.8, with inputs msg_i, tag, $v_i, \vec{V}_{tag}$ and j_i.

9. $CreateProof()$ creates a zero-knowledge proof, π_i, on msg_i and tag, and returns π_i to $Sortition()$

10. $Sortition()$ receives π_i and returns (π_i, msg_i) to $CommitteeVote()$.

11. $CommitteeVote()$ gossips the message, $m_i = (tag, \pi_i, msg_i)$.

12. Upon receiving the gossiped message m_i, party P_h calls $CountVotes()$, Algorithm 5.19, for tag.

13. $CountVotes()$ calls $ProcessMsg()$, Algorithm 5.17, for m_i.

14. $ProcessMsg()$ acquires tag, π_i, and msg_i from m_i.

15. $ProcessMsg()$ calls $Verify()$, Algorithm 5.9, on (tag, π_i, msg_i).

16. $Verify()$ checks that π_i is a valid proof for msg_i and tag and returns 1.

17. $ProcessMsg()$, upon receiving 1, sets $votes'$ equal to j_i' and returns (*votes'*, *value*).

18. $CountVotes()$ receives $(votes', value)$ and adds $votes'$ to $counts'[value]$ using homomorphic addition.

19. $CountVotes()$ checks if $counts'[value]$ is larger than the threshold using homomorphic greater-than comparison, and if it is, returns $value$.

5.2 Proposal 2

The analysis of Baldimtsi et al.'s PPoS proposal [5] in Section 4.2.2 identifies performance as a central issue based on the need to send one (large) proof for each unit of the participant's total voting power (weight) in the multi-stake setting. With this in mind, this section presents a proposal to tackle the performance sacrifice currently needed to achieve complete privacy in [5] by providing a trade-off between privacy and performance. Note that this proposal does not aim to provide a fully private PoS proposal w.r.t. stake and identity, but rather a PoS scheme with a trade-off between the privacy of the stake and the performance.

The idea is to mitigate the performance issue of sending one proof with one voting power for each unit of the total voting power by sending a varying combination of proofs with voting powers of, e.g., 1, 2, and 3, that together add up to a participant's total voting power. While this does reveal some lower ranges of voting power for each participant, it does not (necessarily) reveal the total voting power. The possible combinations of voting powers and the number of proofs sent yield a trade-off between privacy and performance. While many proofs with low voting powers provide better privacy than performance, fewer proofs with higher voting powers provide better performance than privacy.

We start by introducing the equations and function used to formalize the connection between the privacy preserved and the number of proofs and voting powers thereof. Additionally, we explain how the number of proofs and corresponding voting powers are chosen based on the participant's preferences. Next, the modifications necessary in [5]'s protocols for the inclusion of this trade-off scheme are presented. Finally, the modifications necessary to Algorand's procedures [29] to accommodate this trade-off scheme and to apply the modified protocols are presented.

We suggest following the outline, located in Subsection 5.2.4, detailing the order in which the modified protocols and procedures run while reading these sections. Note that, as with Proposal 1, the color red marks the modifications done within Algorithms.

5.2.1 Scheme

This section presents the two equations, Equation 5.3 and 5.4, and the function, *CombinationFunc()*, Algorithm 5.20, necessary for achieving our desired scheme. To summarize, the idea is for the party to decide what level of privacy they want and provide it, along with their total voting power, as input to *CombinationFunc()*. *CombinationFunc()* returns the number of proofs to create for each voting power. The first equation, 5.3, restricts the voting powers in the multiple proofs to be equal to the total voting power of the participant. Meanwhile, the second equation, 5.4, calculates a level of privacy based on the number of proofs sent with specific voting powers.

5.2.1.1 Instantiation of the General Equations

This subsection presents the idea behind the general equations, Equations 5.3 and 5.4, through an instantiation of the equations with some example values, illustrated by Equations 5.1 and 5.2. The example values are the voting powers 1, 2, and 3 and their corresponding *privacy numbers* 1, -2, and -12.

$$j_i = x_1(1) + x_2(2) + x_3(3) \tag{5.1}$$

$$privacy\_level = x_1(1) + x_2(-2) + x_3(-12) \tag{5.2}$$

Equation 5.1 provides the constraint that the sum of the voting powers, 1, 2, and 3, in the multiple proofs for a participant, equals the participant's total voting power, j_i. Moreover, the variables x_1, x_2, and x_3 denote the number of proofs with voting power 1, 2, and 3, respectively. Equation 5.2 gives an estimation of the privacy level based on the number of proofs with voting power 1, 2, and 3. The numbers 1, -2, and -12 in Equation 5.2 are called *privacy numbers* and correspond to the voting powers 1, 2, and 3, respectively. The privacy numbers are chosen based on the notion that one proof with voting power 2 is worse than many proofs with voting power 1, and one proof with voting power 3 is worse than many proofs with voting power 2. Following this notion, the privacy numbers must be adjusted according to the range of possible j_i values.

Thus, having a higher x_1 value than x_2 and x_3 values, i.e., more proofs with voting power 1, yields a higher *privacy_level* variable. The downside to a high x_1 value is the need to generate a higher number of proofs in total. On the other hand, higher x_2 and x_3 values, i.e., more proofs with voting power 2 and 3, contribute to a lower *privacy_level* variable, effectively resulting in fewer proofs in total. The latter case requires less processing power and bandwidth but has worse privacy than the former case. Thus, the scheme presents participants with a choice of *privacy_level*, where a high *privacy_level* variable offers better privacy and worse performance, while a low *privacy_level* variable offers worse privacy and better performance.

Generally, privacy numbers should be chosen based on the corresponding voting power. For example, x_1 has a positive privacy number since proofs with a voting power of 1 preserve privacy as they do not reveal anything about the prover's stake. Meanwhile, x_2 and x_3 have negative privacy numbers as both a voting power of 2 and 3 reveal some information about the participant's stake. These negative privacy numbers effectively decrease the *privacy_level* variable in accordance with the decreased privacy a participant experiences when sending proofs with voting powers higher than 1. Consequently, x_2 has a higher privacy number than x_3 because a voting power of 3 reveals more about the provers stake than a voting power of 2. Note that only a voting power of 1 can correspond to a positive privacy number.

5.2.1.2 Privacy Levels

Table 5.1 shows a varying *privacy_level* variable for different combinations of x_1, x_2, and x_3 with a total voting power $j_i = 6$. Note that the prover achieves the best privacy yet worst performance with six proofs, each with voting power 1. On the other hand, they achieve the worst privacy and best performance with two proofs, each with voting power 3. The number(s) in brackets in the *Privacy description* column shows the stake revealed. I.e., (2 + 3) means that the prover sends one proof with a voting power of 2 and one proof with a voting power of 3. Also, note that proofs with a voting power of 1 are not included in the brackets as they do not reveal any stake.

privacy level	x_1	x_2	x_3	Privacy description	Performance description
6	6	0	0	The highest possible privacy	Worst performance, sends 6 proofs
2	4	1	0	Good privacy, reveal some stake (2)	Sends 5 proofs
-2	2	2	0	Reveal some more stake (2 + 2)	Sends 4 proofs
-6	0	3	0	Reveal some more stake (2 + 2 + 2)	Sends 3 proofs
-9	3	0	1	Reveal some more stake (3)	Sends 4 proofs
-13	1	1	1	Reveals even more stake (2 + 3)	Sends 3 proofs
-24	0	0	2	The lowest possible privacy (3 + 3)	Best performance, sends 2 proofs

Table 5.1: Privacy_Level (Example) Possible combinations of x_1, x_2, and x_3 for $j_i = 6$ with corresponding *privacy_level* and a short description of both the privacy and performance.

Also note that since $(x_1, x_2, x_3) = (0, 3, 0)$ has the same and better performance than $(x_1, x_2, x_3) = (1, 1, 1)$ and $(x_1, x_2, x_3) = (3, 0, 1)$, respectively, and better privacy than both, the two latter combination of proofs should never be chosen. Thus, for $j_i = 6$, if the previously defined privacy numbers were chosen, these options would not be included in the table for *CombinationFunc()*, described in the next subsection.

5.2.1.3 Combination Function

To make the choice of *privacy_level* more intuitive for a party, we introduce the idea of using a function, *CombinationFunc()*, Algorithm 5.20, with the input variables *privacy* $\in [1, 10]$ and j_i. In our instantiation, this function returns the variables x_1, x_2, and x_3 based on the input.

The possible *privacy_level* values varies for different j_i value. For example, for $j_i = 6$, it varies from 6 to -24, as seen in Table 5.1. For another j_i, the possible *privacy_level* values are different. Thus, instead of having parties choose a specific *privacy_level* value, they choose a *privacy* variable $\in [1, 10]$. The party chooses *privacy* $= 10$ if privacy is of utmost importance, and the highest possible *privacy_level* for j_i is chosen by *CombinationFunc()*. If performance is vital, *privacy* $= 1$ is provided as input and the lowest possible *privacy_level* is chosen. If a *privacy* value between 1 and 10 is provided, the *privacy_level* is chosen proportionally to this value and the number of possible *privacy_level* variables (rows). Furthermore, as the voting powers and corresponding *privacy numbers* are predetermined, tables similar to Table 5.1 can be pre-computed for all possible j_i values. Thus, when the *CombinationFunc()* function is called with specific j_i and *privacy* variables, it first looks up the table for j_i. Based on the *privacy* variable, it chooses a *privacy_level* and returns its corresponding x_1, x_2, and x_3 values.

Algorithm 5.20 CombinationFunc()

```
1: function CombinationFunc($privacy, j_i$)
2:     $table = tables[j_i - 1]$
3:     $chosen\_index = int(((len(table)/10) * privacy - 1) + random.random())$ ▷ rounding up or
   down to nearest integer w/ some randomization
4:     $x_1 = table[chosen\_index][1]$
5:     $x_2 = table[chosen\_index][2]$
6:     $x_3 = table[chosen\_index][3]$
7:     return $x_1, x_2, x_3$
8: end function
```

Algorithm 5.20 shows an implementation of the function *CombinationFunc()*. We assume pre-computed tables for each j_i in *tables* and that each table is a list consisting of lists starting at index 0. The lists in each table consist of the *privacy_level* and corresponding x_1, x_2, and x_3 variables, $[privacy\_level, x_1, x_2, x_3]$. First, the table corresponding to the input variable j_i is retrieved as $table = tables[j_i - 1]$ on line 2. Then, given the length of *table*, i.e., the number of *privacy_level* variables and corresponding x_1, x_2, and x_3 combinations possible for j_i, the index corresponding to the chosen *privacy* input variable is retrieved in line 3. Note that the lists in the table are sorted on *privacy_level* where the "best privacy" (highest *privacy_level*) has the highest index. For instance, if there are 15 possible *privacy_level* variables and corresponding combinations and the input variable $privacy = 10$, then the *chosen_index* is 14 (note that the indexes start at 0). If $privacy = 1$, then $chosen\_index = 0$, and if $privacy = 5$, then $chosen\_index = 6$ or $chosen\_index = 7$. Finally, the variables x_1, x_2, and x_3 are retrieved for the *chosen_index* in lines 4 to 6 before being returned in line 7.

Note that for the example above, with 15 possible *privacy_level* variables, for $privacy = 5$, either index 6 or 7 can be chosen. Both indexes are equally likely due to the addition of a random number, *random.random*(), between 0 and 1, before rounding down to the nearest integer. Without this, if we always choose to round up or always choose to round down, some indexes (*privacy_levels*) will never be chosen. The randomization works as follows. If the number $(len(table)/10)*privacy-1)$ is 5.2 there is a 80% chance that $chosen\_index = 5$ and a 20% chance that $chosen\_index = 6$. If this number is 5.5 there is a 50% chance of both $chosen\_index = 5$ and $chosen\_index = 6$. Also note that if the number $(len(table)/10) * privacy - 1)$ is an "integer", e.g. 5.0, it is not changed, i.e., the result is $chosen\_index = 5$.

5.2.1.4 General Equations

This subsection provides generalizations of Equations 5.1 and 5.2. Equation 5.3 gives a general version of Equation 5.1. The x_p variables denote the number of proofs associated with voting powers $p = (1, 2, 3, ..., n-1, n)$. Equation 5.4 gives a general version of Equation 5.2. The a_p variables denotes the privacy numbers associated with the voting powers $p = (1, 2, 3, ..., n-1, n)$.

$$j_i = \sum_{p=1}^{n} (x_p)(p) = (x_1)(1) + (x_2)(2) + ... + (x_n)(n) \tag{5.3}$$

$$\begin{aligned} privacy\_level &= \sum_{p=1}^{n} (x_p)(a_p) \\ &= (x_1)(a_1) + (x_2)(a_3) + ... + (x_{n-1})(a_{n-1}) + (x_n)(a_n) \end{aligned} \tag{5.4}$$

A higher choice of n effectively gives a lower possible *privacy_level* variable because a higher n gives a higher possible voting power to associate with a single proof. On the other side, a higher n gives the choice of better performance.

In this general setting, *CombinationFunc()* returns the variables x_1, ..., x_n. The variable n, i.e., the possible voting powers up to n, and their corresponding privacy numbers a_1, ..., a_n are predetermined, and the tables of the possible x_1, x_2, ..., x_n combinations for different j_i values are pre-computed. When *CombinationFunc()* receives the input variables *privacy* and j_i, it looks up the table for j_i and chooses a *privacy_level* variable based on *privacy* and returns the corresponding x_1, ..., x_n similarly to the description in Subsecion 5.2.1.3 above.

5.2.2 Modifications to the Protocols of Baldimtsi et al.

This section covers the modifications needed in the protocols and the zero-knowledge proof statements from the paper *Anonymous Lottery In the Proof-of-Stake setting* [5] to implement the scheme of Proposal 2. Note that the *Setup()*, *Initialization()*, and *Verify()* protocols are not covered as no modifications are necessary. We refer to Section 2.2 and paper [5] for more details regarding the original protocols.

5.2.2.1 Anonymous Selection Protocol

The modifications done to the overall Anonymous Selection Protocol are the same as with Proposal 1, Algorithm 5.4, in Subsection 5.1.3.1. Thus, the changes consist of replacing b_{tag} with j_i, replacing the condition in the if-check on line 4, $b_{tag} = 1$, with $j_i > 0$ and providing j_i as input to the *CreateProof()* protocol. This Anonymous Selection Protocol is not directly used by our modified Algorand procedures but serves to show the overall order in which Baldimtsi et al.'s protocols [5] run. For an outline of the order in which the modified Algorand procedures and modified Baldimtsi et al.'s protocols [5] run, see 5.2.4.

5.2.2.2 EligibilityCheck and Eligible

The modifications to the *Eligible()* and *EligibilityCheck()* protocols are the same as with Proposal 1, Algorithms 5.5 and 5.6, respectively. The minimal changes consist of renaming wt_i to j_i in *Eligible()* and thus, also renaming the variable returned from *Eligible()*, wt_i, to j_i in *EligibilityCheck()*.

5.2.2.3 Zero-knowledge proof statements

This subsection details the modification to the original zero-knowledge proof statements seen in Subsection 5.1.3.4. First of all, note that wt_i has been renamed j_i. Next, the total voting power, j_i, and the list, $votingpowers$, are provided in the witness, w. As an example, the votingpowers list, $votingpowers = [1, 1, 2, 3, 3]$, signifies that the winning party with $j_i = 10$ sends five proofs, two proofs with voting power 1, one proof with voting power 2, and two proofs with voting power 3.

The *index* is still provided in the witness, w, but has a new definition as it denotes which voting power in *votingpowers* is associated with the particular proof. Line 2 shows this modification. Note that even though there are no apparent changes to the statements in lines 10 and 11, msg_i now contains the voting power, 1, 2, or 3, and thus gives integrity to the voting power gossiped as part of msg_i.

$$\pi \leftarrow NIZK.Prove(crs, x, w)$$

- **Statement** $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C^v_{i,index}, \vec{V}_{tag})$
- **Witness** $w = (i, j_i, votingpowers, stake_i, index, PRF_{sk_i}, v_i, \sigma_i, s_{prf}, pk_i, path_{pk}, path_{\vec{V}_{tag}}, path_{cm}, cm_i)$ where $pk_i = (TRP_{pk_i}, SIG.vk_i, C_{prf})$

  R(x,w)=1 if and only if:

  1. $C^v_{i,index} = F(PRF.sk_i, v_i || tag || index)$
  2. $index \in [1, len(votingpowers)]$
  3. $Sum(votingpowers) = j_i$
  4. $votingpowers[index] = msg_i[2]$
  5. $C^{prf}_i = Com(PRF.sk_i; s_{prf})$
  6. $cm_i = Com(stake_i)$
  7. $V_i = f_{TRP.pk_i}(v_i)$
  8. $V_i = \vec{V}_{tag}[i]$
  9. $Eligible(v_i, stake_i, tag) = j_i$
  10. $\sigma = SIG.Sign(SIG.sk_i, msg_i || tag)$
  11. $SIG.Ver(SIG.vk_i, \sigma_i, msg_i || tag) = 1$
  12. $validPath_h(path_{pk}, rt_{pk}, pk_i) = 1$
  13. $validPath_h(path_{\vec{V}_{tag}}, rt_{\vec{V}_{tag}}, \vec{V}_{tag}[i]) = 1$
  14. $validPath_h(path_{cm}, rt_{cm}, cm_i) = 1$

Lines 3 and 4 provide two new statements to account for unhonest parties who may attempt to alter their voting power(s) in the *votingpowers* list before creating the proof and before gossiping. Line 3 checks that the voting powers in the *votingpowers* list add up to j_i. Note that the list can still be altered, but the proof will not be valid if the voting powers in the *votingpowers* list do not add up to j_i. However, as long as the distribution of the voting powers adds up to j_i, and as long as the *votingpowers* list is the same in all of a party's proofs, we do not concern ourselves with the distribution of the voting powers. Note that while the former is ensured by the addition of line 3, the latter is ensured by an ideal functionality in the *CreateProof()* protocol, see Subsection 5.2.2.4. In turn, line 4, $votingpowers[index] = msg[2]$, retrieves the voting power from the received message as $msg[2]$ and checks that this voting power is part of the *votingpowers* list at the correct *index* and thus not altered by an unhonest party prior to gossiping the message.

5.2.2.4 CreateProof()

As mentioned earlier, the "original" *CreateProof()* protocol, seen in Algorithm 5.7, is based on the single-stake *CreateProof()* protocol by Baldimtsi et al. [5] and adapted to the multi-stake setting following their descriptions.

The first modification is the addition of the function, *CombinationFunc()*, explained in Section 5.2.1, in line 1 of Algorithm 5.21. We use our example from above with the possible voting powers 1, 2, and 3, and let x_1, x_2, and x_3 denote the number of proofs created with the voting power 1, 2 and 3, respectively. As such, *CombinationFunc()* returns the variables x_1, x_2, and x_3 as detailed in Section 5.2.1.

Algorithm 5.21 CreateProof()^2 Version 1

```
Protocol CreateProof($msg_i, tag, v_i, \vec{V}_{tag}, j_i$)
 1: $\langle x_1, x_2, x_3 \rangle \leftarrow CombinationFunc(privacy, j_i)$
 2: $votingpowers = [1] * x_1 + [2] * x_2 + [3] * x_3$        ▷ Create $votingpowers$ list w.r.t. $x$ variables
 3: Let $rt_{\vec{V}_{tag}}$ be the root of $MTree(\vec{V}_{tag})$
 4: Let $path_{\vec{V}_{tag}[i]}$ be the path to $\vec{V}_{tag}[i]$ in $MTree(\vec{V}_{tag})$
 5: Let $rt_{pk}$ be the root of $MTree(pk)$
 6: Let $path_{pk_i}$ be the path to $pk_i$ in $MTree(pk)$
 7: Let $rt_{cm}$ be the root of $MTree(cm)$
 8: Let $path_{cm}$ be the path to $cm_i$ in $MTree(cm)$
 9: Compute $\sigma_i = SIG.Sign(SIG.sk_i, msg_i||tag)$
10: $\pi_i = []$
11: for each $index \in [0, len(votingpowers)]$
12:     Compute $C^v_{i,index} = F(PRF.sk_i, v_i||tag||index)$
13:     $msg_{i,index} = msg_i$        ▷ Create a copy of $msg_i$
14:     Add $votingpowers[index]$ to $msg_{i,index}$
15:     Let $x_{index} = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_{i,index}, C^v_{i,index}, \vec{V}_{tag})$
16:     Let $w_{index} = (i, j_i, votingpowers, stake_i, index, PRF.sk_i, v_i, \sigma_i, pk_i, path_{pk_i}, path_{\vec{V}_{tag}[i]},$
            $path_{cm}, cm_i)$
17:     Compute $\pi_{NIZK,index} := NIZK.Prove(crs, x_{index}, w_{index})$
18:     Set $\pi_{i,index} := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C^v_{i,index}, \pi_{NIZK,index})$
19:     Add $\pi_{i,index}$ to $\pi_i$
20: Output $\langle \pi_i, votingpowers \rangle$
```

The next modification is the addition of line 2, which creates the *votingpowers* list in accordance with the variables x_1, x_2, and x_3. For example, if $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, this creates the list $votingpowers = [1, 1, 2, 3, 3]$ and results in the creation of two proofs with voting power 1, one proof with voting power 2, and two proof with voting power 3.

Another modification is the name change of the input variable wt_i to j_i, provided to the witness in line 16, and used for the zero-knowledge proof statements in lines 3 and 9 as shown above. Also, the *index* in the for-loop in line 11 denotes the indexes in the list *votingpowers*. Inside the for-loop, a copy of msg_i is created and stored as $msg_{i,index}$ in line 13. The voting power (1, 2, or 3), corresponding to the current *index*, is added to the $msg_{i,index}$ in line 14 and thus provided as part of $msg_{i,index}$ to the statement, x_{index}, in line 15. The *votingpowers* list is a new addition to the witness, w_{index}, in line 16. Finally, the *votingpowers* list is returned along with the list of proofs, π_i, in line 20, as each of the voting powers in this list are gossiped as part of msg_i, and thus needed by the *CommitteeVote()* procedure, see Section 5.2.3.2.

Introduction of an Ideal CreateProof() Functionality With the above modified *CreateProof()* protocol, Algorithm 5.21, there exists a possibility for a selected adversary to cheat to get more voting power. An adversary can achieve this by changing the *votingpowers* list in w_{index} (line 16) for each round in the for-loop and adding a high voting power to $msg_{i,index}$ in each round (line 14).

Algorithm 5.22 CreateProof()^2 Version 2

```
Protocol CreateProof($P_i, sid, msg_i, tag, v_i, \vec{V}_{tag}, j_i$)
 1: $\langle x_1, x_2, x_3 \rangle \leftarrow CombinationFunc(privacy, j_i)$
 2: $votingpowers = [1] * x_1 + [2] * x_2 + [3] * x_3$      ▷ Create $votingpowers$ list w.r.t. $x$ variables
 3: Let $rt_{\vec{V}_{tag}}$ be the root of $MTree(\vec{V}_{tag})$
 4: Let $path_{\vec{V}_{tag}[i]}$ be the path to $\vec{V}_{tag}[i]$ in $MTree(\vec{V}_{tag})$
 5: Let $rt_{pk}$ be the root of $MTree(pk)$
 6: Let $path_{pk_i}$ be the path to $pk_i$ in $MTree(pk)$
 7: Let $rt_{cm}$ be the root of $MTree(cm)$
 8: Let $path_{cm}$ be the path to $cm_i$ in $MTree(cm)$
 9: Compute $\sigma_i = SIG.Sign(SIG.sk_i, msg_i||tag)$
10: Query ($CreateProof(msg_i, votingpowers, v_i, j_i, \sigma_i, rt_{\vec{V}_{tag}}, path_{\vec{V}_{tag}[i]}, rt_{pk}, path_{pk_i}, rt_{cm},$
        $path_{cm})$) to the ideal functionality $\mathcal{F}_{CreateProof}$
11: Receive $\pi_i$ from the ideal functionality $\mathcal{F}_{CreateProof}$
12: Output $\langle \pi_i, votingpowers \rangle$
```

For instance, if $j_i = 6$ and $votingpowers = [1, 2, 3]$ (originally), an adversary adds 3 to the $msg_{i,index}$ for all three loops and gives the lists $[3, 2, 1]$ in the first loop, $[1, 3, 2]$ in the second, and $[1, 2, 3]$ in the third. Thus satisfying lines 2, 3 and 4 (and all others) of the zero-knowledge proof statements and resulting in total voting power of $3+3+3=9>j_i$. Therefore, we need to make sure the selected party gives the same *votingpowers* list in w_{index} in each round of the for-loop. For our thesis, we enforce this restriction with the ideal functionality $\mathcal{F}_{CreateProof}$, Algorithm 5.23, replacing the for-loop in Algorithm 5.21. See the new modified *CreateProof()* in Algorithm 5.22.

Algorithm 5.23 $\mathcal{F}_{CreateProof}$

```
The ideal functionality is parameterized by $tag$ and $\vec{V}_{tag}$.
  Upon receiving $(CreateProof(msg_i, votingpowers, v_i, j_i, \sigma_i, rt_{\vec{V}_{tag}}, path_{\vec{V}_{tag}[i]}, rt_{pk}, path_{pk_i},$
      $rt_{cm}, path_{cm}))$ from party $P_i$
  1: $\pi_i = []$
  2: for each $index \in [0, len(votingpowers)]$
  3:     Compute $C^v_{i,index} = F(PRF.sk_i, v_i||tag||index)$
  4:     $msg_{i,index} = msg_i$                                  ▷ Create a copy of $msg_i$
  5:     Add $votingpowers[index]$ to $msg_{i,index}$
  6:     Let $x_{index} = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_{i,index}, C^v_{i,index}, \vec{V}_{tag})$
  7:     Let $w_{index} = (i, j_i, votingpowers, stake_i, index, PRF.sk_i, v_i, \sigma_i, pk_i, path_{pk_i}, path_{\vec{V}_{tag}[i]},$
             $path_{cm}, cm_i)$
  8:     Compute $\pi_{NIZK,index} := NIZK.Prove(crs, x_{index}, w_{index})$
  9:     Set $\pi_{i,index} := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C^v_{i,index}, \pi_{NIZK,index})$
 10:     Add $\pi_{i,index}$ to $\pi_i$
 11: Output $\langle \pi_i, votingpowers \rangle$
```

5.2.3 Modifications to the Procedures of Algorand

We modify Algorand's [29] procedures to implement the combination trade-off scheme and the modified Baldimtsi et al. [5] protocols. The implication of these modifications for Algorand's *CommonCoin()* procedure is also considered. We refer to Section 4.4.1 and the paper [29] for more details into the original procedures.

5.2.3.1 Sortition()

First, the $VRF_{sk}()$ procedure, on line 2 of the original *Sortition()* procedure, Algorithm 5.11, has been replaced with our modified *CreateProof()* protocol, Algorithm 5.22, as seen in line 6 of the modified *Sortition()* procedure, Algorithm 5.24. Specifically, our modified version of CreateProof(), Algorithm 5.22, returns a list of proofs π_i and the list of voting powers, *votingpowers*. This *votingpowers* list is also returned in line 7 of the modified *Sortition()* procedure. We also replace the calculations in lines 3 through 6 of Algorithm 5.11 with the modified *EligibilityCheck()*, Algorithm 5.6, which calls the modified *Eligible()* protocol, Algorithm 5.5, performing the actual calculation of j_i. Continuing, as *CreateProof()* requires j_i, v_i, and $\vec{V}_{tag}$, returned by *EligibilityCheck()*, as input, *EligibilityCheck()* is put first on line 2 in the modified procedure, Algorithm 5.24. Also, in contrast to the original procedure, we only create a proof if $j_i > 0$ and set $\pi_i = null$ if $j_i = 0$.

Further changes include removing *seed*, W, and τ as input variables and including the new input variable *tag*. Note that W, renamed *totalStake*, and τ are global variables used by the *Eligible()* protocol when calculating j_i.

Algorithm 5.24 Sortition()[2]

```
1: procedure SORTITION($value, tag$)
2:     $\left\langle j_i, v_i, \vec{V}_{tag} \right\rangle \leftarrow EligibilityCheck(tag)$
3:     $\pi_i \leftarrow null$
4:     if $j_i > 0$ then
5:         $msg_i = (H(ctx.last\_block), value)$
6:         $\langle \pi_i, votingpowers \rangle \leftarrow CreateProof(msg_i, tag, v_i, \vec{V}_{tag}, j_i)$
7:     return $\langle \pi_i, j_i, votinpowers \rangle$
8: end procedure
```

5.2.3.2 CommitteeVote()

The original *CommitteeVote()* procedure, shown in Algorithm 5.13, is modified in Algorithm 5.25 to use the modified *Sortition()* procedure, Algorithm 5.24. Thus, the input to the *Sortition()* procedure is changed accordingly as seen on line 3 in Algorithm 5.25. The modified *Sortition()* procedure returns the new variable *votingpowers* along with π_i and j_i. Note that π_i is a list of proofs, each of which is associated with the voting power in the same position of the *votingpowers* list. Thus, instead of only gossiping one proof associated with the total voting power j_i as in the original *CommitteeVote()*, we add a for-loop to gossip all the proofs and their associated voting power in lines 6 and 7 of Algorithm 5.25. Note that the length of the lists *votingpowers* and π_i are equal; therefore, it is arbitrary whether we use $len(votingpowers)$ or $len(\pi)$ in line 6. Also, note that the voting power is provided as part of the message msg_i. If the voting power is tampered with, the proof will not be verified as the zero-knowledge proof statements include a signature on this message, see Subsection 5.2.2.3.

Algorithm 5.25 CommitteeVote()[2]

```
1: procedure COMMITTEEVOTE($ctx, tag, value$)
2:     // check if user is in committee using Sortition
3:     $\langle \pi_i, j_i, votingpowers \rangle \leftarrow Sortition(value, tag)$
4:     // only committee members originate a message
5:     if $j > 0$ then
6:         for each $index \in [0, len(votingpowers)]$
7:             $Gossip(\langle tag, \pi_i[index], msg_i = (H(ctx.last\_block), value, votingpowers[index])\rangle)$
8: end procedure
```

5.2.3.3 VerifySort()

As with Proposal 1, the computation of j_i in lines 3 to 6 is removed from Algorand's *VerifySort()* procedure, Algorithm 5.15. For this proposal, the calculation of j_i is not needed as a proof is not associated with the total voting power of the prover, but rather with a fraction of this gossiped in msg_i and thus available to the receiver when counting votes. The $VerifyVRF_{pk}$ procedure is replaced with Baldimtsi et al.'s *Verify()* protocol, Algorithm 5.9. Thus, as with Proposal 1, *VerifySort()* only consists of *Verify()* and is consequently replaced by this protocol.

5.2.3.4 ProcessMsg()

The original *ProcessMsg()* procedure shown in Algorithm 5.16 and modified to fit Proposal 2 in Algorithm 5.26 is similar to Proposal 1's modified *ProcessMsg()* procedure, Algorithm 5.17. However, there are some differences as the plaintext voting power, *votingpower*, is retrieved from msg_i in line 3 instead of the encrypted total voting power, j_i', as in Proposal 1. Otherwise, lines 2 and 3 are equal in both the modified versions. In this proposal, *VerifySort()* is replaced with Baldimtsi et al.'s *Verify()* protocol, Algorithm 5.9. The votes are set equal to *votingpower* if the proof is verified. Thus, the votes are returned to *CountVotes()* in plaintext as in the original *ProcessMsg()* procedure, Algorithm 5.16. Also, note that m_i does not contain a signed message, so lines 3 and 4 of the original *ProcessMsg()* are removed as in Proposal 1. Furthermore, *sorthash* is not returned for the same reasons as discussed in Section 5.1.4.4 of Proposal 1.

Algorithm 5.26 ProcessMsg()^2

```
1: procedure ProcessMsg($ctx, m_i$)
2:     $\langle tag, \pi_i, msg_i \rangle \leftarrow m_i$
3:     $\langle hprev, value, votingpower \rangle \leftarrow msg_i$
4:     // discard messages that do not extend this chain
5:     if $hprev \neq H(ctx.last\_block)$ then return $\langle 0, \perp, \perp \rangle$
6:     if $Verify(tag, msg_i, \pi_i)$ then $votes \leftarrow votingpower$
7:     else $votes \leftarrow 0$
8:     return $\langle votes, value \rangle$
9: end procedure
```

5.2.3.5 CountVotes()

Algorithm 5.18 shows the original *CountVotes()* procedure, and Algorithm 5.27 shows the modified procedure, which only consists of a few changes. The modifications shared with Proposal 1, Algorithm 5.19, are as follows; (1) *tag* replaces *round* and *step*, (2) *ProcessMsg()* no longer returns *sorthash*, and (3) *CountVotes()* and *ProcessMsg()* do not take τ as input as it is a global variable in both proposals. However, in contrast to Proposal 1, Proposal 2 deals with voting powers in plaintext in the same manner as the original *CountVotes()* procedure; thus, no further changes are needed.

Algorithm 5.27 CountVotes()[2]

```
procedure COUNTVOTES($ctx, tag, T, \lambda$)
    $start \leftarrow Time()$
    $counts \leftarrow \{\}$                                    ▷ hash table, new keys mapped to 0
    $voters \leftarrow \{\}$
    $msgs \leftarrow incomingMsgs[tag].iterator()$
    while TRUE do
        $m \leftarrow msgs.next()$
        if $m = \perp$ then
            if $Time() > start + \lambda$ then return $TIMEOUT$
        else
            $\langle votes, value \rangle \leftarrow ProcessMsg(ctx, m_i)$
            if $pk \in voters$ or $votes = 0$ then continue;
            $voters \cup = \{pk\}$
            $counts[value] + = votes$
            // if we got enough votes for $value$, then output $value$
            if $counts[value] > T \cdot \tau$
                return $value$
end procedure
```

5.2.3.6 CommonCoin()

As explained in Subsection 5.1.4.6, the original *CommonCoin()* procedure in Algorand relies on the *sorthash* variable returned from *ProcessMsg()*. As with Proposal 1, the modified *ProcessMsg()* procedure, Algorithm 5.26, do not return this variable's replacement v_i, as this would reveal the identity of the selected. A similar solution as the one in Proposal 1 can be applied, see Section 5.1.4.6. A possible solution is for the prover to accompany a random hash in msg_i before gossiping. Since each selected party may gossip several messages for the same *tag*, a different random hash should be attached to each message to maintain the unlinkability of the messages. New hashes should be created for each message when selected for a *tag*. Note that we do not implement the modifications required to realize this solution in the modified protocols and procedures of Proposal 2, as doing so unnecessarily complicates the presentation of our scheme.

5.2.4 Outline

This section outlines how our modified protocols and procedures from Sections 5.2.2 and 5.2.3 fit together in an instantiation of Algorand. The instantiation assumes an honest party, P_i, with voting power for tag, and an honest party, P_h, who counts votes for the messages received from P_i. Note that the following takes place after setup.

1. Party, P_i, calls $ComitteeVote()$, Algorithm 5.25, with inputs tag and $value.$
2. $CommitteeVote()$ calls $Sortition()$, Algorithm 5.24.
3. $Sortition()$ calls $EligibilityCheck()$, Algorithm 5.6.
4. $EligibilityCheck()$ receives the trapdoor permutation, $\vec{V}_{tag}$, from $ProcessRO()$, computes it's inverse, v_i, and calls $Eligible()$, Algorithm 5.5.
5. $Eligible()$ calculates voting power, j_i, and returns it to $EligibilityCheck()$.
6. $EligibilityCheck()$ receives j_i and returns $(j_i, v_i, \vec{V}_{tag})$ to $Sortition()$.
7. $Sortition()$ receives $(j_i, v_i, \vec{V}_{tag})$, and composes the message, $msg_i = (H(ctx.last\_block), value)$.
8. $Sortition()$ calls $CreateProof()$, Algorithm 5.22, with inputs msg_i, tag, v_i, $\vec{V}_{tag}$, and j_i.
9. $CreateProof()$ creates the $votingpowers$ list with the distribution of voting powers for each proof.
10. For each voting power in $votingpowers$, $CreateProof()$ adds the voting power to $msg_{i,index}$ (i.e., a copy of msg_i), creates a zero-knowledge proof, $\pi_{i,index}$, on $msg_{i,index}$ and tag, and adds $\pi_{i,index}$ to the list π_i. After the loop, the lists π_i and $votingpowers$ are returned to $Sortition()$.
11. $Sortition()$ receives the lists π_i and $votingpowers$ and returns $(\pi_i, j_i,$ $votingpowers)$ to $CommitteeVote()$.
12. For each proof and voting power in the lists π_i and $votingpowers$, respectively, $CommitteeVote()$ gossips the message $m_{i,index} = (tag, \pi_{i,index}, msg_{i,index} = (H(ctx.last\_block), value, votingpower[index]))$ since P_i's voting power, j_i, is greater than 0.
13. After receiving the messages, the party, P_h, calls $CountVotes()$, Algorithm 5.27 for tag.

14. $CountVotes()$ calls $ProcessMsg()$, Algorithm 5.26, for each message, $m_{i,index}$, received.

15. $ProcessMsg()$ acquires tag, $\pi_{i,index}$, and $msg_{i,index}$ from $m_{i,index}$.

16. $ProcessMsg()$ calls $Verify()$, Algorithm 5.9, on (tag, $msg_{i,index}$,π_i).

17. $Verify()$ checks if $\pi_{i,index}$ is a valid proof for $msg_{i,index}$ and tag, and returns 1 if $\pi_{i,index}$ is valid, 0 otherwise. $Verify()$ determines that $\pi_{i,index}$ is a valid proof for $msg_{i,index}$ and tag, and returns 1.

18. $ProcessMsg()$, upon receiving 1, sets $votes$ equal to the voting power in $msg_{i,index}$, and returns $(votes, value)$.

19. Upon receiving $votes$ and $value$, $CountVotes()$ adds $votes$ to the current total votes for $value$ in the table $counts$.

20. Note that steps 15-19 repeats for each message $m_{i,index}$.

21. Also note that if a $value$ receives more votes than the threshold after step 19, the $CountVotes()$ procedure returns $value$.

Note that for the receiver, the index in $m_{i,index}$, $msg_{i,index}$, and $\pi_{i,index}$ are neither known nor needed to verify and count votes. Thus, for the receivers, the procedures *CountVotes()*, *ProcessMsg()*, and *Verify()*, Algorithms 5.27, 5.26 and 5.9, respectfully, this variable is not used. The *index* variable is therefore only included in context of these three procedures in the explanation above in order to yield a better understanding of the connection between the proofs sent by party P_i and received by party P_h.

5.3 Evaluation of the Proposals

We evaluate Proposals 1 and 2 with regards to performance, privacy and security, and compare the proposals to the built upon proposal by Baldimtsi et al. [5].

Note that currently there is no publicly available information on:

1. the probabilities of a stakeholder being selected with different j_i values in Algorand
2. the frequency in which different stakeholders would choose specific *privacy* variables.

While the average case performance improvement in Proposal 1 compared to [5] relies on j_i, Proposal 2 relies on j_i and the *privacy* variable chosen by the stakeholder. Thus, we cannot say anything for certain about the average cases. As a solution, we choose to assess the complexity in order of j_i in Proposal 1. We choose to assess complexity in order of n in Proposal 2, where n is the number of proofs sent determined by j_i and the *privacy* variable. Also, note that while we look at the best and worst case in Proposal 2, this is redundant and thus, not looked at in Proposal 1 as the performance is constant. In terms of privacy, the evaluations differ with regards to the constant privacy for Proposal 1, while, similar to performance, privacy depends upon the choice of privacy variable in Proposal 2.

5.3.1 Proposal 1: Performance Estimate

Proposal 1 improves performance in Baldimtsi et al.'s proposal [5] by removing the need for the creation, gossip, verification, and counting of j_i proofs, resulting in only one proof for each selected party. In this performance estimate, we give complexity in order of j_i.

In Baldimtsi et al.'s proposal, j_i number of (unlinkable) proofs are sent for each selected party. Thus, complexity increase linearly with j_i in [5]. This complexity encompasses (1) proof creation time, (2) bandwidth regarding proof size, (3) everything in $ProcessMsg()$ including proof-verification time, and (4) everything in the while loop in $CountVotes$, including the addition and greater-than comparison. Note that while the complexity in proof creation affects the selected parties, the complexity in $ProcessMsg()$ and $CountVotes()$ affects the receivers. For Proposal 1, one proof is sent regardless of the value of j_i. This reduction improves performance by providing constant complexity in order of j_i as only one proof is created, gossiped, verified, and counted for each selected party.

For the specific case of $j = 1$, the added complexity of incorporating homomorphic encryption must be considered. The zero-knowledge proof statement $index \in [1, wt_i]$ is replaced with $FHE.Equality(msg_i[2], FHE.Enc(params, FHE.pk_a, j_i))$, a more complex statement, as seen in Subsection 5.1.3.4. In addition, the complexity is increased by replacing the addition and greater-than comparison in CountVotes() with homomorphic versions in Subsection 5.1.4.5. Thus, to conclude, Baldimtsi et al.'s proposal [5] is more efficient than Proposal 1 for the specific case of $j_i = 1$. However, Proposal 1 improves performance in [5] for all $j_i > 1$ as the complexity remains constant in Proposal 1 while it increases linearly in Baldimtsi et al.'s proposal.

5.3.2 Proposal 1: Privacy and Security

Proposal 1 aims to achieve complete privacy with regard to stake and identity. We refer to Baldimtsi et al.'s proposal [5] for the privacy and security of the identity and stake achieved with the functionality adapted from their paper. This section focuses on the privacy preserved and security achieved with our new functionality for Proposal 1.

The privacy of the voting power and, thus, also stake depends on the security provided by the homomorphic encryption used. Ideally, an adversary should learn nothing about the plaintext (voting power j_i) from the ciphertext (encrypted voting power j_i'). Proposal 1 makes use of the BGV homomorphic encryption scheme by Brakerski et al. [11]; see their paper for the security provided.

The privacy of the stake also depends upon the ideal functionalities defined in Subsection 5.1.2. Through the use of an ideal greater-than functionality, Algorithm 5.2, the confidentiality of the voting powers is based on the fact that parties do not have access to $FHE.GreaterThan()$. The ideal functionality restricts parties from comparing two chosen encrypted values and thus, restricts them from deducing the plaintext values of the encrypted voting powers. In addition, the confidentiality of the encrypted voting powers depends on the ideal functionality for homomorphic setup, Algorithm 5.1. Proposal 1 relies on this functionality to delete and not reveal the secret key, i.e., the homomorphic decryption key, to the parties.

Since the reliance on ideal functionalities to achieve the desired privacy is not ideal, we include the creation of protocols to realize these ideal functionalities in future work, as detailed in Subsection 6.3.1.

5.3.3 Proposal 2: Performance Estimate

This proposal provides a trade-off between privacy and performance. To a degree, the parties can choose their own performance. For instance, they can choose to send only one proof with their total voting power, resulting in the best possible performance but the worst privacy. Alternatively, they can choose to send multiple proofs, each with voting power one, resulting in the worst possible performance but best privacy. Or they can choose something in-between. Thus, for the performance estimate of this proposal, it is fitting to look at performance in the worst case, best case, and average case. However, as mentioned in Section 5.3, we do not know the probabilities for different j_i values or the frequency in which different stakeholders would choose specific *privacy* variables. Thus, as we cannot say anything for sure about the average case, we look at performance for different j_i values and privacy levels not assessed in the best and worse cases. In this performance estimate, we will give complexity in order of n, where n is the number of proof sent.

Note that because of the different purposes of Proposal 2 and Baldimtsi et al.'s proposal [5], a privacy and performance trade-off versus constant complete privacy, respectively, a direct comparison may seem inexpedient. However, as this proposal is built upon [5], we use this as a reference point to highlight the advantages and disadvantages of Proposal 2. Also, note that while Proposal 2 allows a party to determine the computation power required for proof creation by selecting the *privacy* variable, a party has no control over the number of proofs it needs to verify. If the selected parties value high performance, the receivers get better performance, but the receivers have no say in this decision. Thus, the scheme fails to provide a choice of performance in terms of verification time, as this is decided by the selected parties. Hence, for the remainder of this Subsection, we evaluate Proposal 2 in terms of the best, worst and average case for proof creation.

The best case for performance is when only one proof is sent, i.e., when the lowest privacy level is chosen by Algorithm 5.20 based on the *privacy* variable and j_i of the stakeholder, or when $j_i = 1$. Note that for $j_i = 1$ and inevitably $n = 1$, [5] performs better because of the two extra statements added to [5]'s zero-knowledge proof for Proposal 2 in Subsection 5.2.2.3. For the best-case scenario with the lowest privacy level, Proposal 2 has constant complexity for all $j_i > 1$ as only one proof is sent; $n = 1$. In contrast, the complexity of [5]'s proposal increases linearly with j_i as j_i proofs are sent; $n = j_i$. For this case, note that Proposal 2 reveals the total voting power of the selected party, while [5] does not.

The worst-case in terms of performance for this proposal is when the highest privacy level is chosen, i.e., when j_i proofs each with voting power one are sent. For this case, the selected sends the maximum number of proofs, $n = j_i$, and the complexity increases linearly with j_i as in [5]. Thus, for this worse-case scenario, because of the two added zero-knowledge statements to [5] in Proposal 2, and because the same number of proofs are sent in [5] and Proposal 2, Proposal 2 has worse performance than [5].

As mentioned in Section 5.3, we do not know the probability for different j_i values or the frequency in which stakeholders would choose specific privacy variables. Thus, we cannot say anything for sure about the average case. However, for all other privacy levels (not the highest or the lowest possible privacy level) with $j_i > 2$, this proposal leads to fewer proofs sent by each party compared to [5] without sending the total voting power in one proof; $j_i > n > 1$. Thus, outside of the best and worse case scenarios, Proposal 2 improves performance when looking at complexity in order of n compared to [5], as $n = j_i$ for [5] and $j_i > n > 1$ for Proposal 2.

Hence, the performance improvement compared to [5] lies between the best and worst cases. Hypothesizing that the average case lies somewhere between the worst and best case; that a stakeholder on average does not send one proof with total voting power or only proofs with voting power one, Proposal 2 improves the performance of [5] in order of n.

Generally, besides the possible average case improvement, an advantage unique to this proposal compared to [5] is the performance and privacy trade-off. This trade-off gives a party the possibility to interact with other parties who prioritize performance differently. The party can also change their mind and switch to a better or worse performance dependent on their available computing power or preference.

5.3.4 Proposal 2: Privacy and Security

This proposal does not focus on achieving complete privacy at all times but provides a trade-off between (stake) privacy and performance chosen by the parties themselves while preserving the privacy of the identity. As in Section 5.3.2, we will not focus on the privacy and security provided by the functionality adapted from Baldimtsi et al.'s proposal [5] but on the privacy and security unique for this proposal.

Proposal 2 tackles the practical problem of the computational complexity needed to achieve complete privacy in a PoS protocol and provides an option to achieve some chosen level of privacy while accounting for the fact that computing power is limited and varying for the different parties. Thus, some fixed privacy is not provided by Proposal 2 but rather a choice of privacy. If parties value privacy, they can choose a high privacy level. If the parties value high performance or have limited computing power, they can choose a lower privacy level.

Proposal 2 relies on an ideal functionality, Algorithm 5.23, for security in the creation of proofs. The ideal functionality is in place to avoid a party cheating to get more voting power by adjusting the $votingpowers$ list as detailed in Subsection 5.2.2.4. The transition from an ideal functionality to an implementable protocol is included in future work in Subsection 6.3.2.

Finally, note that when choosing the privacy numbers for the network, one must consider what should determine the best (and worst) privacy. For instance, one has to decide whether the privacy notion suggested in Subsection 5.2.1.1 is applicable; one proof with voting power 3 is worse than many (unlinkable) proofs with voting power 2. The privacy numbers should be adjusted according to such decisions.

Chapter 6

Conclusion

This chapter outlines the results, conclusions, and recommendations for future work. First, a summary of our results is presented. Next, we discuss said results while looking back at the research question presented in Chapter 1. Finally, we outline the future work for Proposal 1 and 2 and two other ideas for future work that came to mind over the course of this thesis.

6.1 Summary of Results

Our thesis started off with a literature study of the state of the art on the topic of privacy-preservation in PoS. As a result, the available formal studies on PPoS, with a record of scientific publication, were presented in Chapter 2. We provided a thorough analysis of these formal studies in Chapter 4, with particular regard to privacy and performance. While the PPoS proposal [12] requires formalization, issues such as performance in [5] and the reliance on an ABC in [25] and [36] were identified.

Of the issues identified in the analysis, we chose to focus our thesis on the performance issue concerning the need for multiple unlinkable proofs in the multi-stake setting of Baldimsti et al.'s PPoS proposal [5]. As such, we studied [5]'s underlying PoS protocol and provided an analysis identifying what parts of Algorand [29] reveal the identity or stake.

Chapter 5 contains two proposals for the mitigation of [5]'s performance issue. The first proposal removes the need for multiple unlinkable proofs in the multi-stake setting of [5] through the use of homomorphic encryption. The second proposal provides an alternative way of tackling the privacy of identity and stake within the consensus mechanism, namely a trade-off scheme between stake privacy and performance. It is worth noting that while the first proposal is for a PPoS, the second proposal is a PoS in the respect that the participants can choose performance over privacy. Finally, we performed an evaluation of the two proposals with regards to performance, privacy, and security. Proposal 1 provides constant performance compared to linear performance in [5] while preserving full privacy. Proposals 2 lets parties interact with other parties with different choices of performance and privacy, accounting for different preferences and computational limitations.

In addition, the paper *Efficient Novel Privacy Preserving PoS Protocol* presenting Proposal 1, was accepted for publication to the Blockchain and Internet of Things Conference (BIOTC). The submitted paper is attached in Appendix A.1 and the final version will be published in the International Conference Proceedings by ACM.

6.2 Discussion

The problem description and introduction present a research question central to the thesis: *Is it possible to design consensus mechanisms that are as energy-efficient as PoS and as privacy-preserving as PoW?* Our approach to answering this question was to complete a literature study and analysis of the currently available PPoS proposals with regards to privacy and performance. The literature study and analysis identified performance issues in Baldimtsi et al.'s proposal [5]. Meanwhile, Ganesh et al.'s proposal [25] and Kerber et al.'s proposal [36] do not preserve the privacy promised due to the reliance on ABC. Following a thorough deliberation, we decided to make improvements to one of the aforementioned PPoS proposals.

To make [5] more efficient, Proposal 1 removes the need for multiple proofs in the multi-stake setting. Our modifications make [5] more efficient in the multi-stake setting when voting power $j_i > 1$; however, we slightly decrease the efficiency of [5] when voting power $j_i = 1$, as detailed in Subsection 5.3.1. The complexity for $j_i > 0$ remains constant in Proposal 1 and increases linearly in [5]. Thus, Proposal 1 performs better than [5] in the multi-stake setting for all j_i except $j_i = 1$.

In contrast to [5]'s proposed protocol, Proposal 1 relies on ideal functionalities for the setup necessary for homomorphic encryption and for the greater-than comparison, as discussed in Subsection 5.1.2.1 and 5.1.2.2, respectively. Thus, to implement Proposal 1, a transition from ideal functionalities to implementable protocols is required. This transition is left for future work as presented in Subsection 6.3.1. Proposal 1 is also reliant upon homomorphic encryption, which is still complex and, thus, difficult to implement in practice. However, research in this field is ongoing, and many papers on FHE [57, 11] and homomorphic comparison [54, 15, 10, 16] exist. Since this is a developing field, we believe that homomorphic encryption, and thus, also Proposal 1, will be practically implementable in a few years.

In response to our research question, despite our improvements to [5], there still does not exist a PPoS proposal as energy-efficient as PoS that is simultaneous as privacy-preserving as PoW. While Proposal 1 succeeds at the latter by providing privacy of stake and identity to the underlying PoS protocol, it fails to be as energy-efficient as the underlying PoS protocol. However, our research suggests that the gap between the efficiency of PoS and PPoS proposals can be reduced. To what extent such reduction is possible remains unclear; however, we strongly believe that the efficiency of PPoS proposals can and will be improved in the future.

Since the complexity of Proposal 1 is constant, we may claim that Proposal 1 provides a competitive edge in terms of performance to PoW protocols, whose complexity and efficiency are directly tied to the difficulty of and computation power needed to solve hard puzzles. This leaves us questioning if maybe our research question poses too strict constraints on how a PPoS protocol has to perform in terms of efficiency. That is to say, the notion that a PPoS must be as energy efficient as a PoS may be too strict since we hypothesize that a PPoS that is much more energy-efficient than some PoW would be a valid alternative to that PoW.

In our attempt to answer the research question, we were left wondering whether it is necessary for all parties to have the same privacy on a Blockchain. Thus, our second proposal provides a flexible trade-off between privacy and performance, i.e., a scheme in which each individual party decides to what extent they want to keep their stake private. Such a scheme sets the stage for an application to a blockchain in which parties with desires for varying degrees of privacy can interact. For example, a party that cares greatly about keeping their stake private can interact on the same blockchain with a party that has no care for the privacy of their stake. Further motivation for Proposal 2 can be found in the paper *On the Anonymity Guarantees of Anonymous Proof-of-Stake Protocols* [39], which claims that "it is impossible to devise a PoS blockchain protocol where both liveness and anonymity are guaranteed", making it reasonable to propose a trade-off scheme in order to achieve some privacy. However, it is worth noting that the practicality of [39]'s attack is conducted on the PoW protocol, ZCash [50], because no practical PPoS implementation exist. Thus, it may be appropriate to question their claim.

Proposal 1 and 2 are not directly comparable in terms of efficiency, complexity, privacy, and security since they are completely different approaches to answering the research question. Whereas Proposal 1 focuses on creating a PPoS, Proposal 2 yields a flexible scheme within which parties can participate to the best of their ability, i.e., with privacy and performance reflective of their computation power and individual preferences. Proposal 2 cannot offer concrete performance guarantees to its parties as the computation power required to verify received proofs is dependent on how many proofs other parties choose to send. In this respect, Proposal 1 offers near constant verification time as the number of proofs received will be consistent. With regards to privacy, Proposal 1 guarantees privacy while Proposal 2 provides the parties with a chosen privacy. When comparing Proposal 1 and Proposal 2, Proposal

1 looks to have better privacy and performance (dependent on the performance of the homomorphic encryption scheme). Although, homomorphic encryption is complex, with ongoing research, we see more potential in Proposal 1 for the future. Thus, while proposal 2 offers an interesting outlook on the problem tackled in this thesis, we endorse Proposal 1 due to the guarantees it offers.

6.3 Future work

This section presents our recommendations for future work on Proposals 1 and 2, followed by two ideas for improvements to Baldimtsi et al.'s proposal [5]. These ideas are not investigated in the thesis and are, thus, presented as possible future directions for research.

6.3.1 Proposal 1

Proposal 1 defines an ideal functionality, Algorithm 5.1, to generate and distribute the homomorphic public key. The ideal functionality specifies that the homomorphic secret key must be deleted immediately following generation to ensure that decryption is impossible. We theorize that the functionality can be realized through a distributed key generation of the homomorphic encryption keys. In such a homomorphic distributed key generation, a threshold of parties contributes to the calculation of homomorphic public and secret keys. Note that in distributed key generation, while the public key is output to the system, no single party has access to the secret key or is able to reconstruct the secret key without compromising the entire threshold of parties [27]. All parties participating in the distributed key generation will be instructed to delete their “individual secret keys” to reduce the likelihood of a single party reconstructing the secret key. Thus, our theorized homomorphic distributed key generation scheme ensures that decryption by a single party remains impossible. Regarding the instantiation with Algorand, the parties chosen to participate in the distributed homomorphic key generation could, for example, be those participating in the governance of the system outlined in [42].

Another ideal functionality specified by Proposal 1, Algorithm 5.2, pertains to the homomorphic greater-than comparison. The homomorphic greater-than comparison is called within an ideal functionality to ensure that parties can only compare a received homomorphically encrypted voting power to the homomorphically encrypted threshold. This restriction is necessary because a comparison of a received homomorphically encrypted voting power with some chosen homomorphically encrypted value would let any party deduct the plaintext values of the received voting powers. We theorize that creating a protocol that realizes this ideal functionality is possible and leave the design and creation of such a protocol to future work.

Once protocols that emulate the ideal functionalities, Algorithms 5.1 and 5.2, have been created, a natural next step will be to test their security by creating simulators. Further work to confirm the feasibility of Proposal 1 in practice is necessary. A practical implementation of Proposal 1 would provide a more qualitative and quantitative basis for evaluating the proposal with regards to performance, privacy, and security. While we restrict the scope of our thesis to not cover the anonymization of transactions, a path for future work would be to combine Proposal 1 with a scheme for anonymous transactions.

Similarly, venturing outside the scope of this thesis to look at the dynamic-stake setting is an area left to future work. Introducing recursive zk-SNARKs for the blockchain proofs similar to the blockchain construction Coda [9] could yield advantages in this setting. By using recursive zk-SNARK proofs for the blocks added to the blockchain, new participants would only need to download the current state of the blockchain along with one recursive zk-SNARK proof of the correctness of the history of the blockchain. This application could make it possible to efficiently verify the entire blockchain while only downloading a very small amount of data. With the block proof equal to the proof of selection for the block proposer, this idea could be applied to Proposal 1, as this proof is already zero knowledge.

6.3.2 Proposal 2

As with Proposal 1, we consider the transition from an ideal functionality to an implementable protocol to be a natural next step in the further development of Proposal 2. The ideal functionality, Algorithm 5.23, serves to place a constraint on the generation of proofs based on the *votingpowers* list, such that proofs are generated based on the same *votingpowers* list. We leave the creation of such a protocol to future work, hypothesizing that designing a protocol that realizes this ideal functionality is possible. Once the transition from ideal functionalities to concrete protocols is complete, a practical implementation of the proposal is in order. Such an implementation would greatly aid in further evaluation of the proposal's feasibility, performance, privacy, and security.

We want to note that we expect the equations and algorithms related to creating the *votingpowers* list to be subject to scrutiny and modification. They are now serving as an example of one way to achieve the scheme laid out by the proposal. Since we have not looked into a multitude of PoS protocols, we cannot with any certainty claim that the most optimal equations and algorithms for Algorand will be the most optimal for other PoSs. Therefore, future work on optimizing the equations and algorithms will also encompass a study of whether or not the same equations and algorithms are optimal for all PoSs.

As with Proposal 1, an investigation into the combination of Proposal 2 with a scheme for anonymous transactions would be a reasonable path for future work. In addition, looking at a transition to the dynamic stake setting would be a natural part of the future work for this proposal. Similar to Proposal 1, using recursive zk-SNARKs for the blockchain proofs could yield advantages in this setting.

6.3.3 Further Ideas for Improvement

Methods with the potential to improve upon the performance issue identified in the analysis of *Anonymous Lottery In the Proof-of-Stake setting* in Section 4.2.2 were investigated through this thesis's work. Two ideas for improving Baldimtsi et al.'s [5] multi-stake setting protocol for Algorand resulted in Proposals 1 and 2. Two other ideas considered over the course of this thesis are introduced below.

6.3.3.1 Recursive zero-knowledge SNARKs in the Gossip Protocol

Recursive zk-SNARKs are the recursive composition of SNARKs, whereas verifying one recursive zk-SNARK proof verifies all the combined zk-SNARK proofs. The complexity of recursive zk-SNARKs has made it costly to implement in practice. However, in 2017, Ben-Sasson et al. claim that they "achieve the first zk-SNARK implementation that practically achieves recursive proof composition" [6]. Several applications for recursive zk-SNARKs have recently emerged, such as Bonneau et al.'s paper *Coda: Decentralized Cryptocurrency at Scale* [9] applying [6]'s technique and Garoffolo et al.'s paper *Zendoo: a zk-SNARK Verifiable Cross-Chain Transfer Protocol Enabling Decoupled and Decentralized Sidechains* [26], both from 2020.

The utilization of recursive zk-SNARKs can improve communication cost and verification time when relaying messages in Baldimtsi et al.'s proposal [5] with Algorand [29] as the underlying PoS. In Algorand, all parties relay every received message during gossip, except when receiving the same message twice to avoid loops. Thus, the parties also verify and relay every belonging zk-SNARK proof they receive. As a solution, the participants can instead relay one recursive zk-SNARK proof together with all the messages received within a time limit. When the participants receive messages during each time interval, they verify the associated recursive zk-SNARK proof(s). At the end of each time interval, the parties create a recursive zk-SNARK proof of all the verified zk-SNARK proofs and relay this one proof along with the messages. Because of the large proof size of the zk-SNARK-proofs in [5], this would improve the communication cost. The transition to recursive zk-SNARK proofs could also decrease the number of proofs each receiver has to verify, thus decreasing the verification time when counting votes.

Practical issues such as merging two zk-SNARK proofs, not including the same proof multiple times in the resulting recursive zk-SNARK proof, and how to avoid loops must be investigated. Also, if the idea is deemed theoretically possible, further work to confirm the practical feasibility is necessary.

6.3.3.2 Aggregated Subvector Commitments

Baldimtsi et al.'s proposal [5] makes use of Pedersen commitment schemes and Merkle trees. Specifically, they use commitments to stake and store the commitments in a Merkle tree. Commitments are also used for the Pseudorandom Function (PRF) secret key stored in a Merkle tree along with the trapdoor public key and the signature verification key. However, other commitment schemes may yield better results. As such, we want to mention a promising commitment scheme to replace the Pedersen commitments and Merkle trees, namely the Aggregatable Subvector Commitment (aSVC) presented in the paper *Aggregatable Subvector Commitments for Stateless Cryptocurrencies* by Tomescu et al. [55], and the improvements this may yield.

While the optimal traversal time, and thus the optimal verification time, for a Merkle tree is logarithmic [53], the proposed aSVC scheme verifies the proofs of the (committed) value, e.g., stake, in constant time. As verifying the proof of the (committed) stake and verifying the proof of the (committed) PRF secret key is part of verifying the zero-knowledge proof of selection, this might improve the verification time of the proof of selection, identified as an issue in the analysis in Section 4.2.2.

While Merkle trees have logarithmic proof size, the aSVC scheme supports aggregation of proofs into a constant-sized proof. Although the proof size of the (committed) stake was not identified as an issue in the analysis, it may be of importance since party P_i attaches his proof for each transaction from party P_i to party P_j. Additionally, while Merkle trees only support updates with dynamic update hints, the aSVC scheme supports static update keys. With static update keys, party P_i only needs to include the proof of his (committed) stake. Meanwhile, with dynamic update hints, party P_i must additionally include the proof for party P_j's stake, requiring interaction. Thus, the aSVC scheme can reduce interaction and improve the size of transactions since the proof size is constant and only one proof is attached with a transaction rather than two. Because of the eventually large number of transactions and associated proofs, a smaller proof size and aggregation of the proofs may prove beneficial in a dynamic stake setting.

Future work includes deducting whether this could improve upon Baldimtsi et al.'s proposal [5] or other PPoS proposals while being theoretically and practically feasible.

References

[1] Proof of stake instead of proof of work. Bitcoin Forum. [Online]. Available: https://bitcointalk.org/index.php?topic=27787.0, July 2011.

[2] alaaradwan. Vigcoin. GitHub repository. [Online]. Available: https://github.com/alaaradwan/VIGCoin, 2019.

[3] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* O'Reilly Media, Inc., 2014.

[4] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930, 2018.

[5] Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou. Anonymous lottery in the proof-of-stake setting. *IACR Cryptol. ePrint Arch.*, 2020:533, 2020.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[7] Christopher Bendiksen and Samuel Gibbons. The bitcoin mining network: Trends, composition, average creation cost, electricity consumption & sources, 2019.

[8] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *International conference on financial cryptography and data security*, pages 142–157. Springer, 2016.

[9] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. *IACR Cryptol. ePrint Arch.*, 2020:352, 2020.

[10] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 391–416. Springer, 2020.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[12] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.

[13] Cardano. [Online]. Available: https://cardano.org/.

[14] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

[15] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 221–256. Springer, 2020.

[16] Heewon Chung, Myungsun Kim, Ahmad Al Badawi, Khin Mi Mi Aung, and Bharadwaj Veeravalli. Homomorphic comparison for point numbers with user-controllable precision and its applications. *Symmetry*, 12(5):788, 2020.

[17] CoinMarketCap. Today's cryptocurrency prices by market cap. [Online]. Available: https://coinmarketcap.com/], 2021. (last visited: <2021-03-06>).

[18] Lin William Cong, Zhiguo He, and Jiasun Li. Decentralized mining in centralized pools. *The Review of Financial Studies*, 2019.

[19] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct nizk arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 532–550. Springer, 2014.

[20] Henry de Valence. Penumbra. GitHub repository. [Online]. Available: https://github.com/penumbra-zone/penumbra, 2021.

[21] Alex De Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018.

[22] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[23] Nabil El Ioini and Claus Pahl. A review of distributed ledger technologies. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 277–288. Springer, 2018.

[24] Peter Fairley. Ethereum will cut back its absurd energy use. *IEEE spectrum*, 56(1):29–32, 2018.

[25] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 690–719. Springer, 2019.

[26] Alberto Garoffolo, Dmytro Kaidalov, and Roman Oliynykov. Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. *arXiv preprint arXiv:2002.01847*, 2020.

[27] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[28] Yossi Gilad. Algorand rewards - a technical overview. [Online]. Available: https://www.algorand.com/resources/blog/rewards-technical-overview, May 24, 2019. (last visited: <2021-19-03>).

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[30] Omkar Godbole. Bitcoin eyes second-biggest monthly drop on record. [Online]. Available: https://www.coindesk.com/bitcoin-eyes-second-biggest-monthly-drop-on-record, May 31, 2021.

[31] LM Goodman. Tezos: A self-amending crypto-ledger position paper. *Aug*, 3:2014, 2014.

[32] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[33] Saad Imran. The positive externalities of bitcoin mining. 2018.

[34] Felix Irresberger, Kose John, and Fahad Saleh. The public blockchain ecosystem: An empirical analysis. *Available at SSRN*, 2020.

[35] Rishi Iyengar. Bitcoin plunges 12% after elon musk tweets that tesla will not accept it as payment. [Online]. Available: https://edition.cnn.com/2021/05/12/tech/elon-musk-tesla-bitcoin/index.html, May 13, 2021.

[36] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 157–174. IEEE, 2019.

[37] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[38] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19:1, 2012.

[39] Markulf Kohlweiss, Varun Madathil, Kartik Nayak, and Alessandra Scafuro. On the anonymity guarantees of anonymous proof-of-stake protocols. *IACR Cryptol. ePrint Arch.*, 2021:409, 2021.

[40] The Algorand Foundation Ltd. Long term algo dynamics. [Online]. Available: https://algorand.foundation/the-algo/algo-dynamics, December 9, 2020. (last visited: <2021-19-03>).

[41] Stefan Mehlhorn. Demystifying algorand rewards distribution: A look at how & when algorand token rewards are calculated. [Online]. Available: https://www.purestake.com/blog/algorand-rewards-distribution-explained/, October 29, 2019. (last visited: <2021-19-03>).

[42] Silvio Micali. A proposal for decentralizing algorand governance. [Online]. Available: https://www.algorand.com/resources/blog/rewards-technical-overview, June 24, 2020. (last visited: <2021-19-03>).

[43] Satoshi Nakamoto. A peer-to-peer electronic cash system. *Bitcoin*, 4, 2008. Available: https://bitcoin.org/bitcoin.pdf.

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[45] Maksym Petkus. Why and how zk-snark works. *arXiv preprint arXiv:1906.07221*, 2019.

[46] The Monero Project. Monero. [Online]. Available: https://web.getmonero.org.

[47] Mayank Raikwar, Danilo Gligoroski, and Katina Kralevska. SoK of used cryptography in blockchain. *IEEE Access*, 7:148550–148575, 2019.

[48] Mayank Raikwar, Danilo Gligoroski, and Goran Velinov. Trends in Development of Databases and Blockchain. In *2020 Seventh International Conference on Software Defined Systems (SDS)*, pages 177–182, 2020. `doi:10.1109/SDS49854.2020.9143893`.

[49] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 2018.

[50] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[51] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A protocol for scalable anonymous communication. *Journal of Computer Security*, 13(6):839–876, 2005.

[52] Oda Skoglund. An investigation into private proof of stake. Project report in TTM4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, Dec. 2020.

[53] Michael Szydlo. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 541–554. Springer, 2004.

[54] Mihai Togan and Cezar Pleşca. Comparison-based computations over fully homomorphic encrypted data. In *2014 10th international conference on communications (COMM)*, pages 1–6. IEEE, 2014.

[55] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.

[56] Andrew Urquhart. The inefficiency of bitcoin. *Economics Letters*, 148:80–82, 2016.

[57] Vinod Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 5–16. IEEE, 2011.

Appendix A
Appendix

A.1 BIOTC paper

The paper *Efficient Novel Privacy Preserving PoS Protocol*, presenting Proposal 1 of our thesis, was accepted for publication to the Blockchain and Internet of Things Conference (BIOTC) after a review process. The process includes quality assurance and a peer review process by 2-3 experts of related research fields. The paper is attached on the following page. Note that the attached paper is the submitted version and not the final version that will be published in the International Conference Proceedings by ACM.

Efficient Novel Privacy Preserving PoS Protocol

Proof-of-concept with Algorand

KAMILLA STEVENSON[*], Norwegian University of Science and Technology, Norway
ODA SKOGLUND[*], Norwegian University of Science and Technology, Norway
MAYANK RAIKWAR, Norwegian University of Science and Technology, Norway
DANILO GLIGOROSKI, Norwegian University of Science and Technology, Norway

Proof of Stake (PoS) emerged to replace and tackle the problem of vast energy consumption in Proof of Work (PoW) consensus. PoS is based on the assumption that the majority of the stake is owned by honest participants. Consequently, instead of solving a computationally hard puzzle to propose the next block in the blockchain, PoS selects a participant with probability proportional to its stake in the network. In contrast to the solution to the puzzle, the proof of selection in PoS has inherent privacy issues. The identity of the selected participant is revealed to other participants to verify the proof, and the stake of the selected can be deducted by frequency analysis. Therefore, Private Proof of Stake (PPoS) emerged to provide a valid alternative to PoW, aiming to tackle the energy consumption in PoW while preserving the privacy of the selected participant in a consensus round. Recent PPoS protocols by Baldimtsi et al. and Ganesh et al., rely on an anonymous broadcast channel and have a large proof size that hinders the practical implementation of the protocols.

In this paper, we identify issues and areas of improvement within the current PPoS protocols. We built our privacy-preserving PoS scheme upon the anonymous lottery by Baldimtsi et al. with an instantiation of Algorand as the underlying PoS protocol. We apply fully homomorphic encryption along with zero-knowledge proof techniques to reduce the proof size and to achieve privacy of selected participant's stake and identity. In comparison with the original anonymous lottery scheme, our scheme achieves better efficiency and complexity.

CCS Concepts: • **Security and privacy → Cryptography**; **Security protocols**.

Additional Key Words and Phrases: Blockchain, Consensus, Zero-knowledge proof, Homomorphic encryption

ACM Reference Format:
Kamilla Stevenson, Oda Skoglund, Mayank Raikwar, and Danilo Gligoroski. 2021. Efficient Novel Privacy Preserving PoS Protocol Proof-of-concept with Algorand . In *BIOTC '21: 2021 3rd Blockchain and Internet of Things Conference, July 8-10, 2021, Ho Chi Minh City, Vietnam.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the past decade, the growth and adoption of cryptocurrencies and, consequently, blockchain technology have been expeditious. From a nonexisting one in 2008, the cryptocurrency market nowadays consists of 5424 cryptocurrencies that all together built a financial market worth around $1.71 trillion (as of 26 May 2021) [8]. Most of these cryptocurrencies use PoW as their consensus mechanism, resulting in huge energy wastage [15]. Therefore, as a solution to the energy wastage issue of PoW, PoS was introduced informally in an online Bitcoin forum in 2011 [1]. Since the informal introduction of PoS, there has been a rapid development in new PoS protocol proposals in the subsequent years [11, 14]. PoS implements a selection (lottery) based on the stake, i.e., the proportion of stake a participant holds, to determine who can extend the blockchain. Furthermore, PoS based blockchain protocol, Ouroboros [12] and cryptocurrency

[*]These authors contributed equally to this research

Algorand [10] have emerged as the prominent PoS based consensus. PoS mechanism can be of two types a) slot-based, b) committee-based. In a slot-based mechanism (Ouroboros Praos), winning the lottery means being able to create a new block for a slot (consensus round). However, in contrast, winning the lottery in a committee-based mechanism (Algorand) can encompass different roles as e.g. proposing a new block or voting on a proposed block.

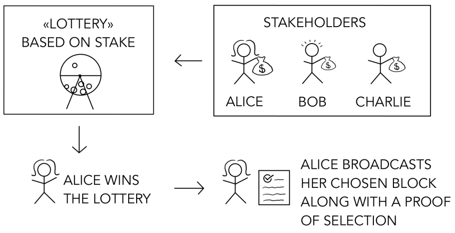

Figure 1. **A Simplified Illustration of Selection in PoS**

Figure 1 depicts the PoS selection process. A lottery is used to illustrate the selection where the stakeholders win with a probability proportional to their stake. Alice wins and thus, she gets to decide the next block added to the blockchain. The selection process illustrated in the Figure 1 must be fair, the participants in the network must be able to evaluate whether they have been chosen as a proposer (who proposes a new block) or a validator (who validates the new block), and all participants must then be able to confirm the validity of the validated/proposed block after being published. To assess the validity of the proposed block or a message, the identity of the selected participant (lottery winner in Figure 1), must be revealed to all participants, as this is a part of the proof of selection. As the identity of the selected participant in PoS is an integral part of the proof of selection, a privacy issue arises as the proof reveals the identity. An adversary can also deduce the stake of a participant by frequency analysis of how often a participant is selected to extend the blockchain. Hence, preserving the privacy of participants is requisite to make PoS equally competitive to PoW, and for its greater adoption to the cryptocurrencies and other blockchain systems while additionally impeding the energy wastage problem of PoW. As a result, a few research studies have been carried out to design privacy-preserving proof of stake protocols, informally known as PPoS.

Note: Throughout the paper, we use participant and stakeholder interchangeably, and the same applies for the lottery and the selection function.

1.1 Related Work

Ganesh et al. [9] proposed *Proof of Stake Protocols for Privacy-aware Blockchains*. In their proposal, the privacy of the winning participant of a lottery is achieved by utilizing a zero-knowledge proof which separates the participant's stake from validating the proof of selection. However, the identity of the selected participant is kept hidden by utilizing an anonymous broadcast channel. Concurrent and independent of Ganesh et al., Thomas et al. proposed *Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake*, a formal model for a Privacy-Preserving PoS distributed ledger. In simple terms, Ouroborous Crypsinous runs variants of Ouroborous Genesis and Zerocash together, creating its own, unique distributed ledger but it also utilizes an anonymous broadcast channel.

The reliance on an anonymous broadcast channel to hide the identity and, thereby, the stake of a participant poses potential drawbacks, especially at the network level. This problem was first pointed out by Baldimtsi et al. [2]. Recently, Kohlweiss et al. [13] renders that even ideal anonymous broadcast channels are insufficient to protect the identity of the selected participant in previously proposed PPoS. Not only protecting the identity of participants but also implementing and maintaining an anonymous broadcast channel is an expensive task. Consequently, the previous proposals mentioned above failed to achieve complete privacy as they stated in their proposals.

Apart from identifying the issues in previous PPoS proposals, Baldimtsi et al. [2] proposed a new method to achieve privacy by separating the identity and stake of the participant from validating the proof of selection. They utilized trapdoor permutation functionality to achieve the privacy of identity. The authors presented their protocol for the single-stake setting and gave ab introduction for the multi-stake setting. The authors proposed the implementation of an Anonymous Selection Functionality based on the underlying selection function of the Algorand protocol. Nevertheless, the major drawback in their protocol occurs during the event of a multi-stake setting where each participant owns multiple stakes. In their multi-stake protocol, each selected participant has to send multiple unlinkable zero-knowledge proofs corresponding to his/her number of winning stakes. Therefore, the total number of proofs arises concerning the number of winning stakes a participant holds. Hence, their proposal suffers from high communication complexity and large proof size; making it impractical to implement in a real-world multi-stake setting of Algorand.

Another recent work by Bünz et al. [6] present Zether: "a fully-decentralized payment mechanism" described as a smart contract. An application of Zether is found to construct a private proof of stake by constructing a lottery protocol to select a winning participant while preserving the identity and stake of the participant. Zether uses ElGamal encryption, hence stakeholders participating in the "lottery", encrypt a lottery ticket and their stake under their public key using Elgamal. Further, the statements about encrypted values can be proved using their zero-knowledge proof mechanism Σ-Bullets. However, formal analysis and investigation are needed for the applicability of Zether in PPoS.

1.2 Our Contribution

- We improve upon the scheme of Baldimtsi et al. [2] [5], and effectively remove the need for multiple unlinkable proofs in the multi-stake setting.
- We thoroughly analyse Algorand protocol and proposed an improved scheme on Algorand with the modifications made to the Original Algorand.
- We present a Proof-of-concept instantiation for privacy preserving Algorand and evaluate our instantiation.

2 PRELIMINARIES

2.1 Non-Interactive Zero Knowledge Proof (NIZK)

NIZK system for a relation R is a set of probabilistic polynomial time algorithms. Given a security parameter λ, a statement $stmt$ and witness w, $NIZK$ is defined as follows:
$NIZK = (NIZK.Setup, NIZK.Prove, NIZK.Verify)$.

- $NIZK.Setup(1^\lambda) \longrightarrow crs$: Produces a common reference string crs.
- $NIZK.Prove(crs, stmt, w) \longrightarrow \pi$: Generates a proof π .
- $NIZK.Verify(crs, stmt, \pi) \longrightarrow 0/1$: Verifies the proof π. Outputs 1 if the proof verifies, else 0.

2.2 Fully Homomorphic Encryption (FHE)

A fully homomorphic encryption scheme [18] is a tuple of probabilistic polynomial time algorithms defined as follows:
$FHE = (FHE.Setup, FHE.KeyGen, FHE.Enc, FHE.Dec, FHE.Eval)$.

- $FHE.Setup(1^\lambda) \rightarrow params$: Outputs global parameters $params$.
- $FHE.KeyGen(params) \rightarrow (pk, sk)$: Outputs a public-private key-pair.
- $FHE.Enc(params, pk, \mu) \rightarrow c$: Given a message $\mu \in R_{\mathcal{M}}$, outputs a ciphertext c.
- $FHE.Dec(params, sk, c) \rightarrow \mu^*$: Given a ciphertext c, outputs a message $\mu^* \in R_{\mathcal{M}}$.
- $FHE.Eval(pk, f, c_1, ..., c_l) \rightarrow c_f$: Given the inputs as public key pk, a function $f : R^l_{\mathcal{M}} \rightarrow R_{\mathcal{M}}$ which is an arithmetic circuit over $R_{\mathcal{M}}$, and a set of l ciphertexts $c_1, ..., c_l$, outputs a ciphertext c_f.

In the above scheme, the message space $\mathcal{M}$ of the encryption schemes is a ring $R_{\mathcal{M}}$, and the functions to be evaluated will be represented as arithmetic circuits over this ring, composed of addition and multiplication gates. We follow BGV FHE scheme of Brakerski et al. [5]. whose security is based on the hardness of the ring-LWE (learning with error) problem. BGV scheme invokes two evaluation functions $FHE.Add()$ and $FHE.Mul()$ in place of $FHE.Eval()$. We assume modifications to $FHE.Eval()$ such that $FHE.Mul()$ is removed and the circuit f is composed solely of addition gates as we are only interested in $FHE.Add()$. At this point, $FHE.Eval(FHE.pk, f, c_1, ..., c_n)$ only encompasses $FHE.Add(FHE.pk, c_1, c_2)$ which returns the homomorphically encrypted sum of c_1 and c_2.

As comparison-based computations can also be performed over fully homomorphic encrypted data [16], recent works [4, 7] show a very efficient way of performing comparisons with optimal complexity. We will apply these astonishing comparison properties over fully homomorphic data. Our work is only interested in greater-than and equality comparison protocols over homomorphically encrypted data. Therefore, following we define these two protocols where x', y' are fully homomorphic encryption of x, y.

(1) $FHE.GreaterThan(x', y')$: Evaluate Greater-Than circuit $GT_{\mathbb{R}}(x', y')$, return 1 if x is greater than y, and 0 if not.
(2) $FHE.Equality(x', y')$: Evaluate Equality circuit $EQ_{\mathbb{R}}(x', y')$, return 1 if x is equal to y, and 0 if not.

For our construction, $FHE.KeyGen()$ is performed using multi-party computation such that each participant has a common shared public key. The secret key is destroyed so that no one posses it as our scheme does not use $FHE.Dec()$.

3 PPOS ANALYSIS

In this section, we give analyses of Algorand consensus [10] and Baldimtsi et al.'s scheme [2] of privacy in Algorand. We enlighten the problem of multiple unlinkable proofs in Baldimtsi et al.'s scheme mentioned in 1.1. Later, we give an overview of our improved scheme built upon Baldimtsi et al.'s scheme to make efficient private Algorand.

3.1 Algorand

In a consensus round of Algorand [10], a fraction of the participants are chosen as committee members using a cryptographic sortition protocol. A committee member can have two roles *role*: a potential block leader or a verifier. The block leaders are chosen to propose a new block in a given round. The participants in Algorand communicate through a gossip protocol. After the proposal of a new block, a set of verifiers, chosen using sortition protocol, vote on the proposed blocks in steps by following a byzantine consensus protocol $BA\star$. This byzantine consensus protocol $BA\star$ does not rely on a fixed number of participants. Moreover, it scales very well. For more details, refer [10].

- Sortition is performed by a participant in a private and non-interactive way using a Verifiable random function. In sortition, a participant computes $\langle hash, \pi \rangle \leftarrow VRF_{sk}(seed||role)$, where sk is participant's secret key and $seed$ is a public random seed. The $hash$ variable determines the number of sub-users j selected for a participant based on the stake of the participant. For example, if a participant is chosen as a verifier and the returned $j = 2$ from the sortition, then the participant acts as two different sub-users and has two votes.

- To reach a consensus, each participant initializes its $BA\star$ protocol with the highest priority block they received. In each step of $BA\star$, a group of participants (a committee of verifiers) is chosen using sortition. The committee members gossip a message, i.e. "their vote" on a block hash ($value$). This is repeated, new committee members are chosen and they vote on a block until enough users in the committee reach a consensus on a block. The committee members vote for the block hash, $value$, in which they received at least some threshold of votes in the previous step. If final consensus is reached, i.e., if some threshold of the committee members votes on the same block hash $value$, then the corresponding block is added to the blockchain.

Protocol $BA\star$ has several procedures $Sortition()$, $VerifySort()$, $CommitteeVote()$, $ProcessMsg()$, $CountVote()$ to perform committee member selection, selection verification, voting on a $value$ (block hash), processing the number of votes for a $value$, til counting all the votes, for a step in a consensus round respectively. Figure 2 depicts the general idea of Algorand consensus. The figure shows how Alice, a selected potential block leader or a verifier, gossips her public key pk, a signed message with her proof of selection π, and her chosen block hash $value$. First, the receivers verify Alice's signed message and then her proof using the received pk. Following, they calculate her voting power j, count j votes for the block hash $value$ and compare the current total votes for $value$ with some threshold.

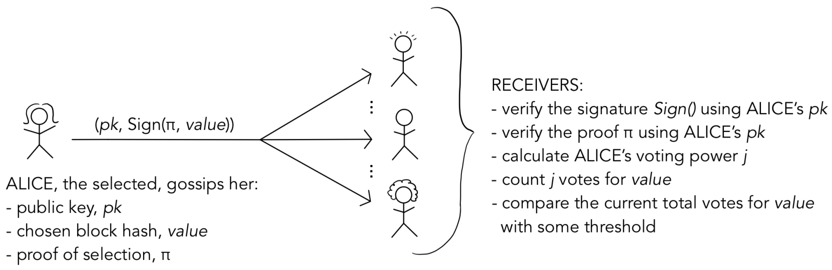
(pk, Sign(π, value))
ALICE, the selected, gossips her:
- public key, pk
- chosen block hash, value
- proof of selection, π
RECEIVERS:
- verify the signature Sign() using ALICE's pk
- verify the proof π using ALICE's pk
- calculate ALICE's voting power j
- count j votes for value
- compare the current total votes for value with some threshold

Figure 2. **A Simplified Illustration of Consensus in Algorand**

In each step of $BA\star$, the identity of the committee members is revealed as the public keys of the committee members are needed to verify the proof of selection (sortition). In the consensus, weight (stake) is provided in the $Sortition()$ and $VerifySort()$ procedures, resulting in making the weight of the participants public. In the $VerifySort()$ procedure, the receiving participants calculate the voting power j of the sending participants. As j is related to the stake of the participants, this should not be revealed to the receiving participant to obtain the privacy of the participants' stake.

3.2 Anonymous Lottery In the Proof-of-Stake setting

In this scheme, the authors presented a nice and flexible anonymous selection functionality to address the problem of identity leaks during the verification of lottery winners. They showed an instantiation of this functionality on the selection (sortition) function of Algorand. As their scheme works very well in the case of single-stake setting, it fails badly in the multi-stake setting. Specifically, in the instantiation of the multi-stake setting of Algorand, the approach to achieve privacy suffers from several large zero-knowledge proofs in each step of a single consensus round in Algorand. A single participant having voting power j from the sortition procedure has to sent j zero-knowledge unlinkable proofs for the same message, where multiple statements in these j zero-knowledge proofs are common.

The following Figure 3 shows how Alice, a selected potential block leader or verifier, gossips her zero-knowledge proof of selection, π, together with a message, *msg*, containing her chosen block hash, *value*. First, the receivers verify Alice's zero-knowledge proof of selection, π. Following, they count 1 vote for the block hash *value* and compare the current total votes for *value* with some threshold. This repeats j times, for each of Alice's j gossiped messages.

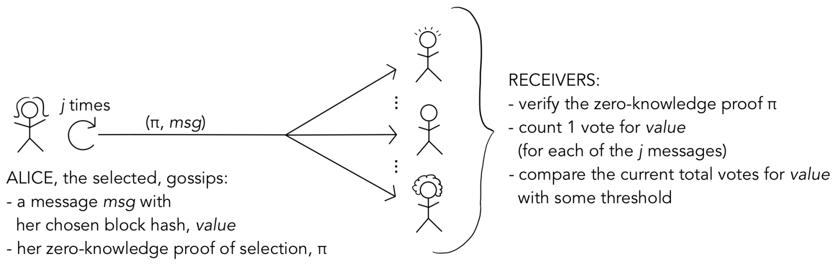
j times
(π, msg)
ALICE, the selected, gossips:
- a message msg with her chosen block hash, value
- her zero-knowledge proof of selection, π
RECEIVERS:
- verify the zero-knowledge proof π
- count 1 vote for value (for each of the j messages)
- compare the current total votes for value with some threshold

Figure 3. **A Simplified Illustration of Baldimtsi et al.'s scheme with Algorand in Multi-stake setting**

Assuming a single round r of Algorand consensus consisting of s steps, if in each step on average n committee members out of P participants send l expected number of zero-knowledge proofs (from their voting power). Then the total complexity for a single round r of Algorand would be $O(snl)$, which can be huge given the number of participants P and their voting power. Henceforth, we conclude that the scheme suffers major performance issues in the multi-stake setting of Algorand. To address this issue, there should be a mechanism to send a single proof for each participant without revealing his stake and voting power, which we construct in the following section.

4 OUR SCHEME

4.1 Overview of our scheme

To remove the need for multiple unlinkable proofs in the multi-stake setting of Baldimtsi et al. proposal of Algorand instantiation, we apply homomorphic encryption to encrypt the voting power, j, such that it can be associated with a single message, *msg*, and proof, π, providing both privacy and accountability. While the former is provided simply by the fact that the j is encrypted, the latter is provided through a property of homomorphic encryption, namely that it is possible to perform basic arithmetic operations on its ciphertexts. Our proposal makes use of BGV homomorphic encryption scheme and two homomorphic comparison tests described in section 2. Hence with the applied changes to Baldimtsi et al. scheme and further modification in the procedures of Algorand, it is possible to 1) send the total voting power along with a single proof, and 2) keep count of votes without revealing the unencrypted voting power. Additionally, through the use of a homomorphic greater-than comparison, participants can tell when a threshold of votes is reached, and henceforth, reach the consensus.

The following Figure 4 depicts how Alice, a selected potential block leader or verifier, gossips her encrypted voting power, j', her zero-knowledge proof of selection, π, and a message, *msg*, containing her chosen block hash, *block*. First, the receivers verify Alice's zero-knowledge proof. Following, they count j votes for the block hash *block* using homomorphic addition and compare the current encrypted total votes for *block* with some threshold using homomorphic greater-than comparison. **Note:** We named block hash *value* of Algorand to *block* in our scheme and use it as further.

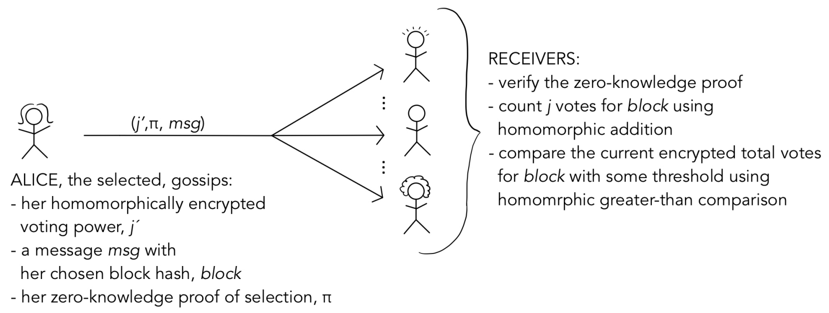
(j′,π, msg)
ALICE, the selected, gossips:
- her homomorphically encrypted voting power, j′
- a message msg with her chosen block hash, block
- her zero-knowledge proof of selection, π
RECEIVERS:
- verify the zero-knowledge proof
- count j votes for block using homomorphic addition
- compare the current encrypted total votes for block with some threshold using homomrphic greater-than comparison

Figure 4. **A Simplified Illustration of our scheme with Algorand in Multi-stake setting**

4.2 Modifications to Baldimtsi et al.'s Protocols

Baldimtsi et al.'s proposed *Anonymized Selection Protocol*, $\Pi^{Eligible}_{Anon-Selection}$, for the single-stake setting. The overall protocol consists of four protocols $Initialization()$, $EligibilityCheck()$, $CreateProof()$, and $Verify()$. This overall protocol shows the order in which a participant, P_i, calls to these protocols. First, it calls the $Initialization()$ protocol, seen on page 12 [2], to generate the public keys, pk_i, and private keys, sk_i, for each participant, P_i. Next, $EligibilityCheck()$ is called to determine whether or not P_i is eligible to "speak" for a tag $(round, step, seed)$, tag . i.e., to determine if P_i is selected to vote on or to propose a block. If eligible, $CreateProof()$ is called to generate a zero-knowledge proof of selection π on a message (that P_i wants to send), msg, and the tag. The proof π, msg, and tag are compiled into a single message and gossiped to the network. Finally, $Verify()$ is called for any message received to confirm in zero-knowledge that the sender of the message is eligible to speak for tag. Note that the underlying PoS protocol is responsible for functionalities such as gossiping and counting of votes.

In this anonymous selection, each participant P_i, has trapdoor permutation keypair $(TRP.pki, TRP.ski)$ for a trapdoor permutation f and a secret key $PRF.ski$ for a pseudorandom function F. Each participant P_i computes a vector $\vec{V}_{tag} = (\vec{V}_{tag}[1], \vec{V}_{tag}[2], \ldots, \vec{V}_{tag}[n])$ consisting the public value $\vec{V}_{tag}[i]$ of P_i. This vector is served as trapdoor permutation. A Merkle tree $MTree(\vec{V}_{tag})$ stores $\vec{V}_{tag}$ with root $rt_{\vec{V}_{tag}}$. The main idea is that for each key, P_i have in $MTree(pk)$, there exist a corresponding $\vec{V}_{tag}[i]$ in the same position in $MTree(\vec{V}_{tag})$. Furthermore, to check for the eligibility, a participant P_i computes $v_i = f^{-1}_{TRP.ski}(\vec{V}_{tag}[i])$. Further, v_i is used to check for the eligibility using $Eligible(v_i, stake_i, tag)$ that returns "j_i" (voting power). To prove the eligibility without disclosing the public identity (public key), P_i computes $NIZK$ argument which proves that P_i knows a pre-image of one of the $\vec{V}_{tag}[i]$ in the vector $\vec{V}_{tag}$. To create a proof about its eligibility for a tag tag, a participant P_i computes a commitment to its winning ticket v_i as $C^v_i = F(PRF.ski, v_i||tag)$ that hides the value v_i. We refer our readers to [2] for full details of the scheme.

Following we present modified $CreateProof()$ for multi-stake setting where only one proof per j_i is needed for each prover, not one proof per $index \in [1, j_i]$ for the same message msg_i, participant P_i wants to send about its selection in original Baldimtsi et al. scheme [2]. **Note:** msg_i contains homomorphically encrypted voting power j'_i.

We construct a modified zero-knowledge proof π_{NIZK} for the original proof of [2] (Section 7.2). Line 7, colored blue in the modified statements below, is a new statement added to account for a dishonest selected participant, an adversary, changing j' before gossiping. The integrity of j from the witness, w, is preserved because of the check on line 6 below, as with Baldimtsi et al.'s original proposal. Thus, j is encrypted and compared with j', retrieved from the received message msg_i (as $msg_i[2]$). The comparison is made using $FHE.Equality()$ to confirm that j' was unaltered before gossiping. Also, note that even though lines 8 and 9 looks unaltered, the signature provides integrity of the encrypted voting power, j', during gossip as this is part of the message, msg, included in the signature.

```
**Protocol** CreateProof($msg_i, tag, v_i, \vec{V}_{tag}, j_i$)
1: Compute $C_i^v = F(PRF.sk_i, v_i||tag)$
2: Let $rt_{\vec{V}_{tag}}$ be the root of $MTree(\vec{V}_{tag})$
3: Let $path_{\vec{V}_{tag}[i]}$ be the path to $\vec{V}_{tag}[i]$ in $MTree(\vec{V}_{tag})$
4: Let $rt_{pk}$ be the root of $MTree(pk)$
5: Let $path_{pk_i}$ be the path to $pk_i$ in $MTree(pk)$
6: Let $rt_{cm}$ be the root of $MTree(cm)$
7: Let $path_{cm}$ be the path to $cm_i$ in $MTree(cm)$
8: Compute $\sigma_i = SIG.Sign(SIG.sk_i, msg_i||tag)$
9: Let $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C_i^v, \vec{V}_{tag})$
10: Let $w = (i, j_i, stake_i, PRF.sk_i, v_i, \sigma_i, pk_i,$
        $path_{pk_i}, path_{\vec{V}_{tag}[i]}, path_{cm}, cm_i)$
11: Compute $\pi_{NIZK} := NIZK.Prove(crs, x, w)$
12: Set $\pi_i := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C_i^v, \pi_{NIZK})$
13: Output $\pi_i$
```

- $\pi \leftarrow NIZK.Prove(crs, x, w)$
- **Statement** $x = (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, tag, msg_i, C_i^v, \vec{V}_{tag})$
- **Witness** $w = (i, j_i, stake_i, PRF_{sk_i}, v_i, \sigma, s_{prf}, pk_i, path_{pk}, path_{\vec{V}_{tag}}, path_{cm}, cm_i)$,
  where $pk_i = (TRP_{pk_i}, SIG.vk_i, C_{prf})$

R(x,w)=1 if and only if:

(1) $C_i^v = F(PRF.sk_i, v_i||tag)$
(2) $C_i^{prf} = Com(PRF.sk_i; s_{prf})$
(3) $cm_i = Com(stake_i)$
(4) $V_i = f_{TRP.pk_i}(v_i)$
(5) $V_i = \vec{V}_{tag}[i]$
(6) $Eligible(v_i, stake_i, tag) = j_i$
(7) $FHE.Equality(msg_i[2], FHE.Enc(params, FHE.pk_a, j_i)) = 1$
(8) $\sigma = SIG.Sign(SIG.sk_i, msg_i||tag)$
(9) $SIG.Ver(SIG.vk_i, \sigma, msg_i||tag) = 1$
(10) $validPath_h(path_{pk}, rt_{pk}, pk_i) = 1$
(11) $validPath_h(path_{\vec{V}_{tag}}, rt_{\vec{V}_{tag}}, \vec{V}_{tag}[i]) = 1$
(12) $validPath_h(path_{cm}, rt_{cm}, cm_i) = 1$

Other participants receiving the proof π_i alongwith msg_i (consisting of j_i') first parse the proof π_i as $\pi_i := (rt_{\vec{V}_{tag}}, rt_{pk}, rt_{cm}, C_i^v, \pi_{NIZK})$. further they run the verification protocol $Verify(msg_i, tag, \pi_i)$ where they verify the proof using $NIZK.Verify(crs, x, \pi_i)$ algorithm.

4.3 Modifications to Algorand

Sortition procedure of Algorand is called in *Eligible* procedure of line 6 in the above NIZK proof construction. Hence, the *Sortition* procedure should be modified to output the homomorphically encrypted voting power of the participant if it is selected as a committee member. In contrast to the original $Sortition()$ procedure, a (zero-knowledge) proof of selection is created only if a participant P_i is selected, i.e., if the participant has voting power $j_i > 0$. Notably, the modified version performs homomorphic encryption on j_i and outputs j_i'. msg_i is provided as input to the $CreateProof()$ protocol to construct NIZK proof as described above. Following, we describe the modified Sortition procedure of Algorand's original Sortition procedure [2](Figure 11). The *block* represents the block hash the prover/selected wants to vote for or propose corresponds to Algorand's *value*. The current state of the ledger is represented by context *ctx*.

After getting selected through the $Sortition()$ procedure, a selected participant gossips a message through the modified $CommitteVote()$ procedure of Algorand. In contrast to Algorand, the message, msg_i, is no longer signed before gossip to preserve the voting power and not to disclose the signing key of the prover. Henceforth, the signature is included in zero-knowledge proof (line 8 of NIZK proof) $\sigma = SIG.Sign(SIG.sk_i, msg_i||tag)$.

Furthermore, after receiving the gossiped message $m = \langle tag, \pi_i, msg_i \rangle$, the receiving participants call $VerifySort()$ procedure of Algorand. In our case, this procedure can simply replaced by the $Verify()$ protocol where receiving participant checks the zero-knowledge proof received in the gossip message m. After checking the correctness of the proof, the participants process each message m by executing $ProcessMsg()$ protocol of Algorand. Following are the modified version of the Algorand procedures in our scheme.

```
procedure SORTITION($block, tag$)
    $\left\langle j_i, v_i, \vec{V}_{tag} \right\rangle \leftarrow EligibilityCheck(tag)$
    $\pi_i \leftarrow null$
    $j'_i \leftarrow 0$
    if $j_i > 0$ then
        $j'_i = FHE.Enc(params, FHE.pk_a, j_i)$
        $msg_i = (H(ctx.last\_block), block, j'_i)$
        $\pi_i \leftarrow CreateProof(msg_i, tag, v_i, \vec{V}_{tag}, j_i)$
    return $\langle \pi_i, msg_i \rangle$
end procedure
```

```
procedure COMMITTEEVOTE($ctx, tag, block$)
    // check if user is in committee using Sortition
    $\langle \pi_i, msg_i \rangle \leftarrow Sortition(block, tag)$
    // only committee members originate a message
    if $msg_i[2]! = 0$ then
        $Gossip(\langle tag, \pi_i, msg_i \rangle)$
end procedure
```

Further, these receiving participants call the $CountVotes()$ procedure to count all the received votes for the block hashes $block$ in the received messages $\{m\}$. The procedure calls ProcessMsg() for each received message to get the block hash $block$ and associated votes for each message. $CountVotes()$ procedure computes the threshold value similar to Algorand using a fraction of the expected committee size T, and the expected number of users selected for the committee τ in a round, and homomorphically encrypt it. Regarding the homomorphic addition, we hypothesize it is possible to define a level A as the maximum amount of additions necessary for counting votes to achieve consensus in our privacy-preserving Algorand. Note that in step 3 of the $CountVotes()$ procedure for the $counts'$ data structure we use a hash table, where the new keys will be mapped to 0 or 1 depending on the used homomorphic encryption.

```
procedure PROCESSMSG($ctx, m$)
    $\langle tag, \pi_i, msg_i \rangle \leftarrow m$
    $\langle hprev, block, j'_i \rangle \leftarrow msg_i$
    // discard messages that do not extend this chain
    if $hprev \neq H(ctx.last\_block)$ then return $\langle 0, \perp, \perp \rangle$
    if $Verify(tag, msg_i, \pi_i)$ then $votes' \leftarrow j'_i$
    else $votes' \leftarrow 0$
    return $\langle votes', block \rangle$
end procedure
```

```
procedure COUNTVOTES($ctx, tag, T, \tau, \lambda$)
    $start \leftarrow Time()$
    $counts' \leftarrow \{\}$
    $msgs \leftarrow incomingMsgs[tag].iterator()$
    while TRUE do
        $m \leftarrow msgs.next()$
        if $m = \perp$ then
            if $Time() > start + \lambda$ then return $TIMEOUT$
        else
            $\langle votes', block \rangle \leftarrow ProcessMsg(ctx, m)$
            if $votes' = 0$ then continue;
            $counts'[block] = FHE.ADD(FHE.pk_a, f, counts'[block], votes')$
            $x' \leftarrow counts'[block]$
            $y' \leftarrow FHE.Enc(params, T \cdot \tau + 1)$
            $b \leftarrow FHE.GreaterThan(x', y')$
            if $b$ return $block$
end procedure
```

The complete details of the modified procedures and the protocols of Algorand and Baldimtsi et al.'s scheme will be provided along with the security proofs in the full version of the paper.

4.4 Evaluation of Our Scheme

Our scheme aims to achieve full privacy concerning stake and identity. We refer to Baldimtsi et al.'s scheme [2] for the privacy and security of identity and stake achieved with the functionality adapted from their paper. The privacy of the voting power and thus, also stake, depends on the security properties provided by the underlying homomorphic encryption. Ideally, an adversary should learn nothing about the plaintext (voting power j) from the ciphertext (encrypted voting power j'). As our scheme makes use of the BGV homomorphic encryption, therefore, the privacy of voting power in our scheme directly follows from the security properties of the BGV scheme.

5 CONCLUSION

In this paper, we thoroughly analyze the privacy implications in Algorand and performance issues of the multi-stake setting of Baldimtsi et al.'s scheme [2]. We remove the need for multiple unlinkable proofs in the multi-stake setting of [2] through the use of BGV homomorphic encryption scheme with the secure comparison properties. Our scheme performs better and is more efficient than [2].

5.1 Future Directions of work

In our attempt to achieve privacy in PoS and improve upon the existing schemes, we found potential improvement in PPoS. An interesting direction of work can be to provide a flexible trade-off between privacy and performance,i.e., a scheme in which each participant decides to what extent they want to keep their stake private. The utilization of recursive zk-SNARK [3] can improve the communication cost and verification time when relaying messages in Algorand's gossip protocol with Baldimtsi et al.'s scheme [2]. Recent advances in aggregatable subvector commitment schemes [17] can be applied to aggregate the commitments and to make proof-size shorter.

REFERENCES

[1] 2011. Proof of stake instead of proof of work. Bitcoin Forum. https://bitcointalk.org/index.php?topic=27787.0

[2] Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou. 2020. Anonymous Lottery in the Proof-of-Stake Setting. *IACR Cryptol. ePrint Arch.* 2020 (2020), 533.

[3] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 111–120.

[4] Florian Bourse, Olivier Sanders, and Jacques Traoré. 2020. Improved Secure Integer Comparison via Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing, Cham, 391–416.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[6] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*. Springer, 423–443.

[7] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 221–256.

[8] CoinMarketCap. 2021. Total Market Capitalization. https://coinmarketcap.com. [Online; accessed 26-May-2021].

[9] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. 2019. Proof-of-stake protocols for privacy-aware blockchains. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 690–719.

[10] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.

[11] Felix Irresberger, Kose John, and Fahad Saleh. 2020. The Public Blockchain Ecosystem: An Empirical Analysis. *Available at SSRN* (2020).

[12] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

[13] Markulf Kohlweiss, Varun Madathil, Kartik Nayak, and Alessandra Scafuro. 2021. On the Anonymity Guarantees of Anonymous Proof-of-Stake Protocols. (2021).

[14] Cong T Nguyen, Dinh Thai Hoang, Diep N Nguyen, Dusit Niyato, Huynh Tuong Nguyen, and Eryk Dutkiewicz. 2019. Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities. *IEEE Access* 7 (2019), 85727–85745.

[15] Mayank Raikwar, Danilo Gligoroski, and Katina Kralevska. 2019. SoK of used cryptography in blockchain. *IEEE Access* 7 (2019), 148550–148575.

[16] Mihai Togan and Cezar Pleşca. 2014. Comparison-based computations over fully homomorphic encrypted data. In *2014 10th international conference on communications (COMM)*. IEEE, 1–6.

[17] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. 2020. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*. Springer, 45–64.

[18] Vinod Vaikuntanathan. 2011. Computing blindfolded: New developments in fully homomorphic encryption. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, 5–16.