BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE EVALUATION OF DOS RESISTANT PROOF-OF-STAKE PROTOCOLS

VYHODNOTENIE VÝKONU PROOF-OF-STAKE PROTOKOLOV ODOLNÝCH VOČI DOS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. TIMOTEJ PONEK**
AUTOR PRÁCE

SUPERVISOR — **Ing. MARTIN PEREŠÍNI**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

155973

Institut: Department of Intelligent Systems (DITS)
Student: **Ponek Timotej, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Computer Networks
Title: **Performance Evaluation of DoS Resistant Proof-of-Stake Protocols**
Category: Security
Academic year: 2023/24

Assignment:

1. Get familiar with existing Proof-of-Stake protocols, particularly COPOR and LaKSA.
2. Make a theoretical comparison of these protocols regarding throughput, scalability, security, liveness, safety, finality, etc. And make a security analysis considering general vulnerabilities.
3. Acquaint yourself with anonymization techniques used in network traffic, particularly onion routing.
4. Analyze the missing functionality of the original COPOR solution with and without the anonymization layer and extend and reimplement missing parts.
5. Implement proof-of-concept LaKSA protocol and add an anonymization layer to it.
6. Evaluate and compare the key performance of the extended COPOR protocol and LaKSA approach.
7. Discuss results, security, and potential improvements of the proposed solution.

Literature:

- Rychlý, škálovatelný, a DoS-rezistentní proof-of-stake konsensuální protokol založen na anonymizační vrstvě - https://www.vut.cz/studenti/zav-prace/detail/136727
- LaKSA: A Probabilistic Proof-of-Stake Protocol
- Algorand
- DarkFi, an anonymous L1 based on zero-knowledge, multi-party computation, and homomorphic encryption.
- Dou, Hanyue, et al. "A probabilistic Proof-of-Stake protocol with fast confirmation." *Journal of Information Security and Applications* 68 (2022): 103268.
- Homoliak, Ivan, et al. "A security reference architecture for blockchains." *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019.
- Demystifying the Dark Web: An Introduction to Tor and Onion Routing

Requirements for the semestral defence:
1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Perešíni Martin, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: 1.11.2023
Submission deadline: 17.5.2024
Approval date: 6.11.2023

Abstract
This diploma thesis focuses on evaluation of the Proof of Stake protocols Algorand, LaKSA and COPOR in terms of properties key to blockchains. The focus is on the theoretical analysis of the protocols in terms of security, performance, and ability to deal with DoS attack on the leaders responsible for block creation in each round. The individual protocols have been implemented and extended with an anonymization layer to thwart DoS attacks on the leader, and their performance has been tested. Last but not least, we pointed out another possible direction for the development of the novel COPOR protocol.

Abstrakt
Táto práca je zameraná na vyhodnotenie Proof of Stake protokolov Algorand, LaKSA a COPOR z pohľadu vlastností kľúčových pre blockchainy. Dôraz je kladený na teoretickú analýzu protokolov z pohľadu bezpečnosti, výkonnosti a schopnosti vysporiadať sa s útokom typu DoS na lídrov, ktorý sú zodpovedný za tvorbu blokov v jednotlivých kolách. Jednotlivé protokoly boli implementované a rozšírené o anonymizačnú vrstvu pre zmarenie útokov typu DoS na lídra, a ich výkonnosť bola otestovaná. V neposlednom rade sme poukázali na ďalší možný smer vývoja nového protokolu COPOR.

Keywords
blockchain, consensus, Proof of Stake protocols, Algorand, LaKSA, COPOR, anonymization, onion routing, tor, dandelion, VRF, PRF

Klíčová slova
blockchain, konsensus, Proof of Stake protokoly, Algorand, LaKSA, COPOR, anonymizácia, cibuľové smerovanie, tor, dandelion, VRF, PRF

Reference
PONEK, Timotej. *Performance Evaluation of DoS Resistant Proof-of-Stake Protocols.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Perešíni

Rozšířený abstrakt

Cieľom tejto práce bolo študovať vybrané konsensuálne protokoly typu Proof of Stake (dôkaz vloženého vkladu, skratka PoS), konkrétne Algorand, COPOR a LaKSA, vykonať ich porovnanie z pohľadu vlastností kľúčových pre blockchain siete,ako je priepustonosť, škálovateľnosť, živosť, konečnosť/finalita, bezpečnosť a zároveň sme porovnali aj ich vlastností týkajúce sa férovosti a odolnosti voči útokom typu Denial of Service (zamedzenie prístupu, skratka DoS).

Algorand je protokol ktorého účinným mechanizmom, ktorý znižuje šancu DoS útoku, je využitie overovateľnej náhodnej funkcie (skratka VRF), ktorá vyberá nepredvídateľne vyberá lídrov (sú zodpovedný za vytváranie blokov) a voličov (hlasujú za bloky s najvyššou prioritou), kde je v každom kole zvolený počet rozdielny. Pre útočníka je ťažké identifikovať lídra v danom kole, keďže jeho identitu sa dozvie až keď zistí ktorý vytvorený blok mal najvyššiu prioritu a obdrží samotný blok (priorita a blok sa posielajú v dvoch samostatných správach).

Protokol Laksa adresuje problémy v Algorande, ako sú vysoko kolísajúci počet lídrov a voličov v jednotlivých kolách. Obmedzením tohoto počtu fixnou hornou hranicou autori argumentujú zvýšenie bezpečnosti protokolu a rýchlosti finalizácie pridávaných blokov.

COPOR bol navrhnutý a prvotne implementovaný v diplomovej práci Mareka Tamaškoviča s cieľom byť lepšou alternatívou voči existujúcim PoS protokolom. Odstránil potrebu pre hlasovanie za bloky, prítomne v Algorande a LaKSe, použitím takého mechanizmu voľby lídra, ktorý zaručuje výber rovnakého lídra každým uzlom. Ďalej tento mechanizmus vyberá aj alternatívnych lídrov, ktorý sú vyprodukujú blok v prípade nedostupnosti hlavného lídra alebo predošlých lídrov, a týmto zabezpečuje, že v každom kole bude eventuálne vytvorený nový blok. Ako súčasť tohto protokolu bolo navrhnuté používanie identifikátorov namapovaných na sieťové adresy, ktorých mapovanie nie je dopredu známe utočníkovi a musí ho zistiť aby bol schopný zamedziť lídrom vo vytváraní blokov útokom typu DoS.

Pre ďalšie znemožnenie utočníkovi získať mapovanie identifikátorov na sieťové adresy bolo navrhnuté použitie anonymizácie posielaných správ. Sem sme sa pozreli na rôzne možnosti anonymizácie ako sú cibuľové smerovanie (onion routing) vysoko využívané v anonymizačnej sieti tor, mixovanie správ (mixing networks) a taktiež na jednoduchý mechanizmus navrhnutý ako spôsob znemožnenia útočníkovi zisťiť pôvodcu danej správy v blockchaine Bitcoin, nazývaný dandelion (slovensky púpava). anonymizačnú vrstvu sme následne implementovali na ideách použitých v anonymizačnej sieti tor, a zakomponovali sme aj ideu dandelionu a skúmali bezpečnosť týchto riešení.

Pre porovnanie COPORu s ostatnými menovanými protokolmi sme ich taktiež implementovali v jazyku `python3.8` s využitím rovnakých stavebných blokov, aby bolo porovnanie čo najférovejšie. Nami implementovaná anonymizačná vrstva je implementovaná tak aby nebola závislá od bežiaceho protokolu, a preto bolo jej zakomponovanie do LaKSy priamočiare. Pre Algorand nebola použitá, pretože sem anonymizácia správ neovplyvňuje útočníkovu schopnosť identifikácie lídra a voličov v danom kole.

Testy ukázali, že pridanie anonymizačnej vrstvy vytvára rozummné spomalenie priepustnosti protokolu v kontraste s dodatočne získanou bezpečnosťou. Priepustnosť protokolu COPOR bola pri využití anonymizácie znižená maximálne o polovicu, kde v niektorých prípadoch šlo iba o 20% spomalenie. V prípade kde sme simulovali útok typu DoS na niektoré uzly v sieti ich neprítomnosťou sme naviac pozorovali podobnú priepustonosť pri využití anonymizácie. Pre protokol LaKSA pridaná anonymizácia nemá veľký vplyv na priepustnosť, pretože sem prebiehajú kolá vo fixných časových intervaloch, a od ich dĺžky závisí priepustnosť. Anonymizácia tu nesmie vytvoriť spomalenie dlhšie ako dĺžka týchto inter-

valov, inak protokol nebude schopný vytvárať rozhodnutia. V našich testoch sme zistili, že pre použité fixné časové intervaly anonymizácia nevytvára takéto spomalenie. Výsledky pre Algorand ukázali jeho schopnosť vysporiadať sa aj s prípadnými offline lídrami, kde tento fakt nemal vplyv na priepustonosť.

Performance Evaluation of DoS Resistant Proof-of-Stake Protocols

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Martin Perešíni. The supplementary information was provided by Ing. Ivan Homoliak, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Timotej Ponek
May 17, 2024

Acknowledgements

I would like to thank my supervisor Ing. Martin Perešíni, for his valuable advice in composing the text of this thesis and guidance thoughtout the whole work. Next, I would also like to thank Ing. Ivan Homoliak Ph.D. for his consultations on the implementation direction. I would also like to thank Ing. Marek Tamaskovic for his help with the creation accunt for metacentrum tests. Computational resources were supplied by the project "e-Infrastruktura CZ". Finally, I would like to thank my family and my girlfriend for standing beside me during my studies for their support during the making of this thesis.

Contents

1 Introduction and motivation **3**

2 Blockchain structure **4**
2.1 Consensus . . . 6
2.2 Proof of Work . . . 7
2.3 Proof of Stake . . . 8
2.4 Practical Byzantine Fault Tolerance (PBFT) . . . 9

3 Algorand **11**
3.1 Block structure . . . 11
3.2 Algorand model . . . 12
3.3 Sortition . . . 13

4 LaKSA **15**
4.1 Block structure . . . 15
4.2 LaKSA model . . . 16
4.3 Leader and voter election . . . 17

5 COPOR **19**
5.1 Block structure . . . 19
5.2 COPOR model . . . 20
5.3 Leader election . . . 21

6 Comparison **22**
6.1 Throughput . . . 22
6.2 Scalability . . . 23
6.3 Liveness . . . 23
6.4 Finality . . . 24
6.5 Safety . . . 24
6.6 Fairness . . . 25
6.7 DoS Resistance . . . 25

7 Network anonymization techniques **27**
7.1 Onion routing . . . 27
7.2 Mixing networks . . . 29
7.3 Dandelion . . . 30

8 Implementation details **31**
8.1 Network layer . . . 32

8.2 Structure of messages 32
8.3 Message processing 34
8.4 Anonymization layer 34
8.4.1 Key exchage mechanism 35
8.4.2 Finding a circuit 35
8.4.3 Different anonymization types 36
8.5 Protocol bootstrapping 39
8.6 Used libraries 40

9 Testing **42**
9.1 Serialization tests 42
9.2 Local tests 43
9.3 Metacentrum tests 46
9.4 Discussion 48

10 Conclusion **49**

Bibliography **50**

A Included CD contents **54**

Chapter 1

Introduction and motivation

The growing interest in blockchain technology is due to its various features, such as decentralization, transparency, and immutability. Moreover, its potential extends beyond financial sector. It is revolutionizing supply chain management, intellectual property rights, and even voting systems, proving its versatility. In this fast-paced evolution, the quest for scalable, secure, and efficient consensus mechanisms has been a driving force behind numerous innovations. These mechanisms assure that all participating nodes have the same view of the blockchain and encourage them to obey the rules of the protocol. One such innovation are the Proof of Stake (PoS) protocols, which has garnered significant attention for its potential to address the scalability and energy consumption issues associated with traditional Proof of Work (PoW) protocols. The primary objective of this thesis is to not only explore the fundamental principles and features of chosen PoS protocols - Algorand, Laksa, and Copor - but also to extend this exploration by adding an anonymity layer to these protocols. This additional layer introduces a novel element, aiming to enhance security and privacy of each consensus participant. The main reason for this addition is to reduce possibility of Denial of Servise (DoS) attacks on users with specific roles in given round by replacing their routing address for an anonymous identifier. With this countermeasure in place, anonymity layer only reveals the identifier of the given user to the attacker when the message gets decrypted. However, the pivotal question that arises is what drawbacks this additional layer of security brings, whether it results in significant throughput decrease and what other downsides it has.

In the chapters that follow, we start with an overview on the blockchain technology (Chapter 2), followed by indepth look at Algorand (Chapter 3), LaKSA (Chapter 4) and Copor (Chapter 5) protocols, explaining the building blocks behind them. After that, we provide comparison of these protocols in Chapter 6. We then embark on journey to find suitable anonymization of routing information in Chapter 7. Next, we outline all important details of the implementation, giving extra explaination on the main contribution of this thesis - the anonymization layer. The question whether addition of anonymization layer results in significant throughput decrease is answered in Chapter 9. Finally, in Chapter 10, we discuss of the results and possible future improvements for both the anonymity layer and proof of concept implementation.

Chapter 2

Blockchain structure

Blockchains can be regarded as a result of continous evolution of distributed ledger technologies (DLT) [3]. Its initial use case was for financial industry (i.e. cryptocurrency), but it has grown into various other spheres. Their data structure can be defined as decentralized, distributed, and immutable ledger that store information in a series of interconnected blocks. Each block contains a set of transactions or data, and is linked to the previous block via hash of the previous block, forming a chain. The transactions are typically created by a set of users to indicate the transfer of tokens from one blockchain user to the other blockchain user.

Blockchain is usually deployed on a peer-to-peer (P2P) network. Blocks on the chain are created by nodes, you can imagine them as user runned computers that are interconnected via affiliation to the same network. Unlike traditional client-server architectures [32], every node in P2P network can be both client (issue transactions) and a server (validate and execute transactions). Other than that, the nodes in the blockchain do not always have a strict specific role or a fixed hierarchy. The role may not exist or may change over time depending on the actual operation in the blockchain. In general, the typical roles of the node in the network (depicted in Figure 2.1) are as follows [15]:

- **Consensus node** - Such nodes maintain a local copy of the entire blockchain in order to participate in the underlying consensus protocol. They read and validate incoming blocks (made by other concensus nodes), create new blocks and can prevent malicious behavior of the adversary, by not appending invalid transactions or blacklisting malicious nodes. If controlled by the adversary, they may support harmful behavior and participate in attacks on the blockchain.

- **Validating node** - These nodes have same privileges as consensus nodes, except that they cannot create new blocks. They are not capable of preventing malicious behavior of the adversaries, but they can detect them (since they posses copies of the entire blockchain).

- **Lightweight node** - These nodes keep only a limited fragments of the blockchain, typically only block headers that concern them, reducing the overall size of the stored data. Therefore, they can detect only a limited set of attacks, pertaining to their transactions.

The nodes may also take other roles specific to the underlaying blockchain, but the ones described above are common for most blockchains.

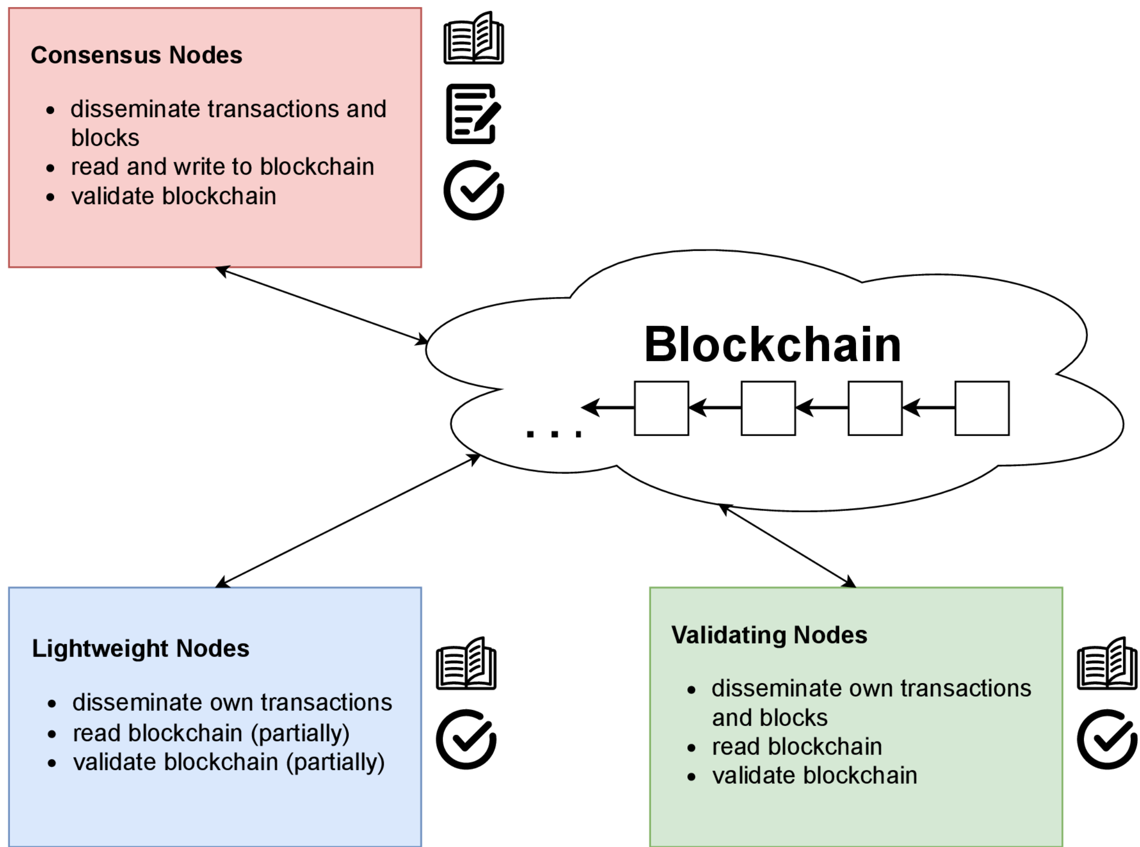

Figure 2.1: Roles that the node can take in blockchain [15]

At the inception of each blockchain there has to be created a genesis block. It states initial coin supply and its distribution to the bootstrapping nodes. As the blockchain starts to work and produces more blocks, another nodes can join the network and validate the transactions. For new nodes (or non-node users possesing a blockchain wallet [8]) to create transactions, they first have to buy crypto coins from the nodes that poss them, exchanging the coins for other assets they own (could be classical currency or crypto coins from other blockchain). They can also obtain coins by other ways, such as by validating or producing blocks or as a reward for honest behavior. Each blockchain protocol specifies different rewarding mechanism (also refered to as incentives mechanisms when they encourage users to behave honesty and possibly punish users behaving adversarially) and possibilities to obtain coins.

Based on how a new node enters a consensus protocol, blockchain can be of a following type [15]:

- **Permissionless** - allow anyone to join the consensus protocol without permission.
- **Permissioned** - require a consensus node to obtain permission to join the consensus protocol from a centralized or federated authority(ies), while nodes usually have equal consensus power (i.e., one vote per node).
- **Semi-Permissionless** - require a consensus node to obtain some form of permission (i.e., stake) before joining the protocol; however, such permission can be given by any

consensus node. The consensus power of a node is proportional to the stake that it has.

2.1 Consensus

Consensus mechanism is a critical component that enables nodes to agree on the state of the chain and validate transactions. These mechanisms replace the need for human verification and auditing, automating the process of achieving distributed agreement and ensuring the integrity and security of the blockchain. Consensus mechanisms ensure that all participating nodes have a consistent view of the ledger, which is essential for the trustless environment of blockchains. The choice of consensus mechanism directly impacts the network's parameters like transaction processing speed, throughput and security.

Standard design goal for concensus mechanisms is to secure properties of CAP theorem (consistency, availability and partition-tolerance). In terms of blockchains (as outlined by Hud [16]), we refer to consistency as **safety**, and it ensures that if an honest node accepts (or rejects) a transaction, then all other honest nodes make the same decision. **Liveness** stands for availability and it ensures that all valid transactions are eventually processed and included in the blocks. Finaly, **finality** stands for partition-tolerance and it represents a sequence of blocks from genesis block to block B which can be assumed to be infeasible to overturn.

Xiao at al. [37] define goals that blockchain consensus seeks to achieve a little differently. They state these four goals:

- **Termination** - At every honest node, an incoming new transaction is either discarded or accepted into the blockchain (first inserted into the mempool and over time picked and added into the new block).
- **Agreement** - Every new transaction including the block it is contained within should be either accepted or discarded by all honest nodes. An accepted block should be assigned the same sequential block number by every honest node.
- **Validity** - A valid transaction/block should be accepted accepted into the blockchain by all honest nodes.
- **Integrity** - All accepted transactions at all honest nodes should be consistent (no double spending). All accepted blocks must be generated correctly and hash chained in chronological order.

The *termination* and *validity* goals are simply a more detailed descriptions of the *liveness* from our previous definition. The *agreement* goal intuitively corresponds to the *safety*. The descriptions of the left of terms *integrity* and *finality* do not necessarly match, but we can see the common thought behind them.

As we strive to find a mechanism that would provide us with better and better properties, many consensus mechanisms have emerged. We further present recent classification by Singh et al. [33] (you may find other not enterily matching classifications, as new consensus protocols are still emerging), which is as follows:

- **Consensus based on Proof of X** (PoX) - these protocols are based on the proof of something, which they eleviate to achieve consistency and guarantee safety and other desired blockchain properties. Examples are Proof of Work (PoW) [24], Proof

of Stake (PoS) [17], Proof of Burn (PoB) [26], etc., many others were not mentioned and new ones are still emerging.

- **Consensus based on PAXOS** - it is a family of protocols intended for solving consensus in a network of unreliable nodes, where partial network or node failures can happen. Created and first described in paper "The Part-Time Parliament" [18] in 1998 by Leslie Lamport, Paxos works in three phases: *1. prepare* - establish the latest Generation Clock and gather any already accepted values, *2. accept* - propose a value for this generation for replicas to accept and *3. commit* - let all replicas know the chosen value and the fact it was chosen [27, 19]. Here are also many subflavors of the main Paxos idea, like Multi-Paxos, Fast-Paxos, Byzantine-Paxos, etc..

- **RAFT algorithm** - is a consensus algorithm that is an effective equivalent to PAXOS. The consensus process in RAFT is broken into three parts [25]: *leader election*, *log replication* and *safety*. The basic idea is that a leader is elected in the cluster who accepts client request and replicates the log to other servers. The server is at any time in one of the three states: leader, follower, or candidate. As RAFT is not a family of protocols, we mention no subflavors.

- **Consensus based on byzantine faults** - family of protocols that are based on Byzantine General Problem. Different generals in different locations with their troop need to attack a mutual enemy. It was proven that to solve this problem more than 2/3 of the generals have to be honest [20]. This effectively means that there is no solution in case of $3m + 1$ generals if more than m of them are adversaries. Examples of this family members are Practical Byzantine Fault Tolerance (PBFT), Byzantine agreement-based consensus and XRP Ledger consensus protocol [7].

- **DAG-based consensus** - These protocols utilize total ordering of recieved blocks to rielieve network consumption [22]. Examples are Dagbase, Jointgraph and BlockDAG.

Further we provide closer look at those protocols that are somehow related to our work.

2.2 Proof of Work

In this protocol, the block creator has to prove that it has done some amount of computational work in a certain interval of time. PoW was first proposed to be used in peer-to-peer version of electronic cash system (Bitcoin), where the online payments occur between two parties directly without any intermediary in between. It is believed to solve double spending problem that was caused due to reversible nature of online transactions.

The addition of the block in this system is called as the mining process and the nodes performing this operation are called as miners. In order to mine (produce) new block, the miner must choose a random nonce value and calculate the hash value. If this hash value is less than a certain pre-defined target treshold value, then the block gets added to the blockchain (the whole described process is illustrated in Figure 2.2). This is also confirmed by the other miners in the network. In bitcoin, SHA256 hash function is used. The difficulty for calculating a valid hash is maintained by setting the target T value for every 2016 block It is feasible for a malicious attacker to overthrow one block in a chain, but as the valid blocks in the chain increase, the workload is also accumulated, therefore overthrowing a long chain requires a huge amount of computational power. PoW belongs to the probabilistic-finality consensus protocols since it guarantees eventual consistency. At

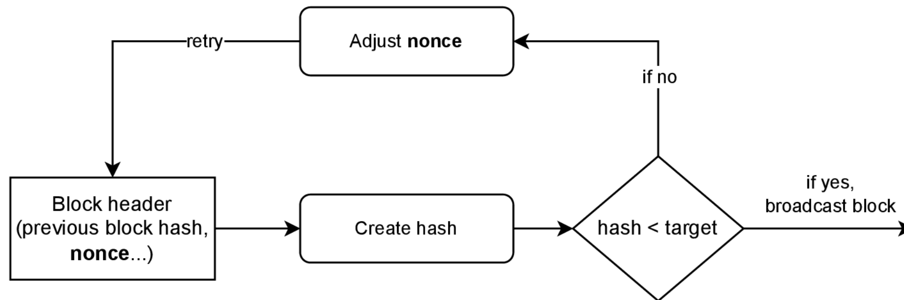

Figure 2.2: Flowchart of PoW [39]

times it is possible that two miners create the block at the same time. When this happens, the block that is the most synchronized in the network gets the finalization of agreement by all the nodes in that network.

Analysis of PoW shown that it takes 10 min to generate a block and it takes 1 h (60 min) to confirm one block; the time one block takes for confirmation, 6 blocks are generated. The PoW consensus protocol is used by cryptocurrencies Bitcoin, Ethereum (which recently switched to PoS in Ethereum 2.0), etc.. The Ethereum also provides platform for building smart contracts and other applications over it.

2.3 Proof of Stake

In PoS, the node which creates a new block is selected with depending on the held stake rather than the computational power. Although nodes still need to solve a SHA256 puzzle - *SHA256(previous block hash, ...)* $< target \times coin$, in contract to PoW, the key to solve this puzzle is the amount of held stake - *coin* (shown in Figure 2.3). Hence, PoS is far more energy-saving protocol compared to PoW. Like PoW, PoS is also a probabilistic-finality consensus protocol. PPcoin [17] was the first cryptocurrency to apply PoS to the blockchain. In PPcoin, in addition to the size of the stake, the coin age is also introduced in solving a PoS puzzle. For instance, if you hold 10 coins for a total of 20 days, then your coin age is 200. As the coinage grows, the difficulty for selecting correct nonce lowers. Once a node creates a new block, her coin age will be reset to 0.

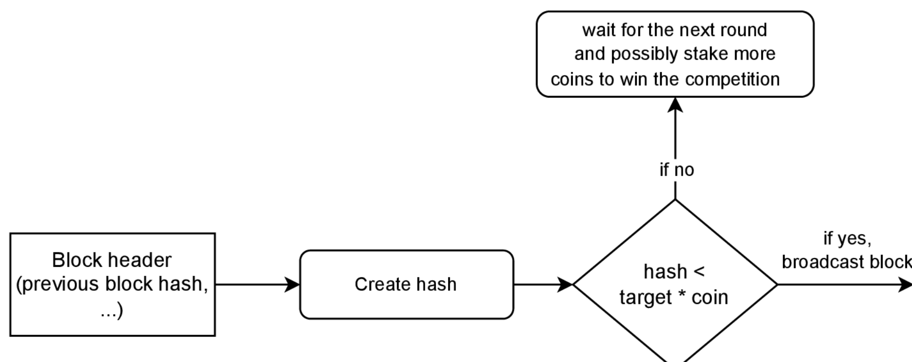

Figure 2.3: Flowchart of PoS [39]

To decrease the power consumption even more, **Delegated Proof of Stake (DPoS)** was introduced. It's principle is to let the nodes who hold stake vote to elect the block creators. This way, the stakeholders give the right to create the blocks to the delegates they chose instead of creating blocks themselves, thus reducing their energy consumption to 0. If the delegates are unable to generate blocks in their turns, they will be dismissed and the stakeholders will select new nodes to replace them. Cryptocurrencies that adopted this mechanism are BitShares and EOS, where EOS has turned DPoS to BFT-DPoS (Byzantine Fault Tolerance-DPoS) in its newest version.

In this work, we will focus on protocols that are a subtype of PoS protocols, where block producers (also called leaders) are selected based on the signature from the last block in a function that selects them proportionally based on their actual stake (this process is called election). Other values like randomness created by each round leader and contained within the block can also be incorporated to ensure fair and non-predictable selection of the leaders. If the leaders could be predicted and known before the start of the corresponding round, a malicious user could use this information to his advantage, and e.g. deny them to produce the block by DoS attack (which we adress in this work). This subtypes of PoS protocols are also known as *virtual mining* protocols [36], as the block producers do not consume any resources.

Pure proof of Stake (PPoS) is group of PoS protocols where anyone who staked at least one corresponding coin (in contrast to previous PoS representants, where usually a certain treshold for staked coin is possed and has to be reached for stakeholder to become eligible to participate in the consensus) can be elected as leader. The election process is secret and utilizes verifiable random function (VRF), thanks to which anyone can validate that an incoming block was created by the valid leader, without prior knowledge of him [21]. First protocol that adopted this idea was Algorand, where after election and block creation follows rounds of voting for blocks, depending on the priority for each crated block, where highest priority wins. The nodes again find out secretly, utilizing VRF, whether they are eligible to vote and then cast their votes. The block that first recieves specified treshold for votes is considered as valid and appended to the blockchain.

2.4 Practical Byzantine Fault Tolerance (PBFT)

In this consensus protocol, nodes are of two type - primary and backup. The client (user) will issue a request to the primary node (which is responsible for block creation). It will be decided whether the request can be executed or not after the primary and backup nodes have agreed upon the request. PBFT works in following 5 phases [38] (see Figure 2.4):

- **Request** - client send a request to the primary node
- **Pre-prepare** - primary node assigns a sequence number corresponding to the request. Then a *pre-prepare* message is constructed and broadcast to the backup nodes.
- **Prepare** - after receiving the *pre-prepare* message, each backup node broadcasts a *prepare* message to other backup nodes. All backup nodes broadcast messages to each other.
- **Commit** - All nodes validate the message and broadcast a *commit* message. The request will be executed if verified successfully.

- **Reply** - The client waits for responses from different nodes. If the client receives a correct response from $f+1$ identical *reply* messages (f is the number of Byzantine nodes), it indicates that the nodes in the network have reached a consensus

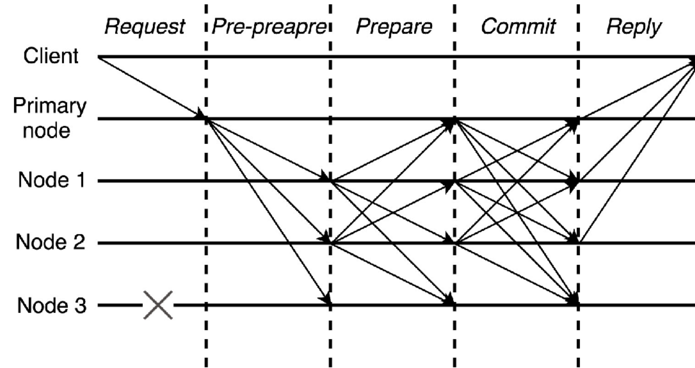

Figure 2.4: Flow of communication in PBFT. From article [39]

The message complexity of *pre-prepare* phase is $O(n)$, upon reception each backup node has to broadcast the *prepare* and *commit* messages, which is of $O(2n)$ complexity, resulting in $O(2n) \times O(n) = O(2n^2)$ that belongs to $O(n^2)$ complexity. The resulting message complexity is one of the drawbacks for PBFT consensus.

Chapter 3

Algorand

Algorand [13] is a Pure Proof of Stake (PPoS) protocol, and also a digital currency and transaction platform, that incorporates an innovative consensus algorithm designed to address scalability issues by eliminating mining inherent to PoW protocols and aims for strong decentralization by basing decisions on calculations rather than subjective influences. Algorand is capable of swiftly confirming transactions, with latencies of approximately one minute, even when it scales to accommodate a large user base. Furthermore, It also ensures almost immediate recovery from any potential network partitions, making the likelihood of forking extremely low, even in asynchronous environments. To achieve strong consistency, algorand employes a protocol known as BA*, a Byzantine Agreement protocol that is based on cryptographic sortition and utilizes Verifiable Random Functions (VRFs) to ensure high user security. The concept of Algorand was originally developed by Silvio Micali, a Turing Award and Godel Prize-winning computer scientist from MIT, and the todays version of Algorand [1] incorporates a variety of these ideas supported by rigorous proofs from Micali's work and the work of others.

3.1 Block structure

For our explanation of Algorand, we will assume following structure of the block

$$B = (i, H(B_{-1}), Txs_root, Txs, seed, pk, \sigma),$$

where:

- $\boldsymbol{i}$ is the round number
- $\boldsymbol{H(B_{-1})}$ is the hash of the previous valid block
- $\boldsymbol{Txs_root}$ a set of transactions included in the block
- $\boldsymbol{Txs}$ a set of transactions included in the block
- $\boldsymbol{pk}$ is a leaders public key
- $\boldsymbol{\sigma}$ is a signature created by the leader over all previous fields except pk

Other than the messages containing blocks, also the messages containing votes and priorities are broadcasted via network. The vote is structure $v = (i, H(B_{-1}), s, pk, \sigma)$ with the folowing fields:

- $\boldsymbol{i}$ is the round number
- $\boldsymbol{H(B_{-1})}$ is the hash of the previous valid block
- $\boldsymbol{phase}$ is the number of the phase the vote was created in
- $\boldsymbol{proof}$ is the proof generated by the `ProveVRF` function
- $\boldsymbol{proof_hash}$ is the the hash of the proof
- $\boldsymbol{pk}$ is the voter's public key
- $\boldsymbol{\sigma}$ is a signature, created by the voter over all previous fields except pk

3.2 Algorand model

Further, we describe how Algorand works on achieving consensus. It works in 5 phases [1, 2] each lasting for different amount of time and consisting of different amount of steps:

- **Phase 0 - Proposal** (lasts for 2λ seconds)

  In this phase, each node first runs `Sortition` function (Algorithm 1) with her public key, number of expected leaders - parameter *expectedLeader* and role *leader*. When the obtained number of $subUsers$ is positive, the node then proceeds to calculates its priority, by consencutively hashing her *proof_hash* concatenated with index value, starting from 0 to obtained number of $subUsers$. After that, the node sends to her peers two different messages - one containing the block together with *proof* and *proof_hash* obtained from sortition and other one with her highest calculated *priority*, *proof* and hash of the block. The message with priority is broadcasted in order to make other nodes aware of the proposed block and be able to vote for obtained block hash if it was the one with the highest priority. Each node stores received priorities and the highest recieved priority is voted for in the next phase. The received blocks are for now only cached and broadcasted further by the nodes, but not validated, as the nodes have yet to decide on the block that will be appended

- **Phase 1 - Filtering** (last for max λ seconds)

  Each node here again first runs `Sortition` function to determine whether she should vote for her highest recieved priority block hash. This time, the `Sortition` is runned with arguments *expectedFilter* and role *voter_filtering*. Nodes that are eligible to vote broadcast a message containing the received hash of the block with highest priority and a number of obtained $subUsers$ from the aforerunned `Sortition` (which are used as a number of votes for given block hash), together with *proof*, *proof_hash*, their public key and the number of the phase (so the other nodes can determine which expectedUsers and role values were used in `Sortition` without transmitting the underlaying values, which reduces the message size). Upon reception of the message with vote, each node verifies it, checking the vote signature and running `Verify-Sortition` function (Algorithm 2) to check if the vote was created by the eligible node. If that's the case, the number of $subUsers$ with the vote are appended to the corresponding counter with votes for the given block hash. With each increase of votes for the given block hash, it is checked if the resulting number of votes exceeded the *threshold_filter*, and in such case the block hash won this round and the node

proceed to the next phase voting for this block hash if eligible. If no block hash recieves enough votes to exceed the *threshold_filter* after λ seconds, it is voted for empty block in the next round.

- **Phase 2 - Certifying** (last for max λ seconds)

  Now when the nodes found out the winner of the filtering phase (or it resulted in the empty block), they vote once again. They run `Sortition` function with arguments *expectedCert* and role *voter_certifying* to find out whether they are eligible to vote, and when yes, they create and broadcast a message containing the winning block hash (or no hash if voting for empty block) and a number of obtained *subUsers* from the aforerunned `Sortition` together with *proof*, *proof_hash*, their public key and the number of the phase. The first block hash with votes exceeding the *threshold_filter* is then appended to the blockchain (empty block is appended in case the empty block hash was he first to exceed *threshold_filter*), the round number is increased and consensus start again at the phase 1. If no such block hash exists after *lambda* seconds, protocol continues with recovery in next phase.

- **Phase 3 - Recovery** (can be repeated 249 times and each time it lasts for λ-2λ seconds)

  Here the nodes periodicaly run the steps from certifying phase, hoping that some blockhash will exceed the specified threshold. Reason for why this phase can find such block hash is because the possibility that some of the nodes not yet recieved the block hash with overall highest priority (which they can still recieve in these rounds) and therefore this block could not recieve enough votes. If such block hash is found, the round number is increased and consensus goes back at the phase 1

- **Phase 4 - Fast recovery** (can be repeated 2 times and each time it lasts for λ seconds)

  This phase is responsible for making the final decision for block commitment. It is highly unlikely that the consensus reaches this phase, but if it happens, the nodes vote in this phase with higher probability for an empty block, so the next round can start.

3.3 Sortition

Algorand uses sortition to determine roles for nodes in given phase of the round. With help of the sortition a small group of nodes is found, which can be easily verified by any other node of their status as verifier or proposer using the proofs outputted by the algorithm. The sortition ensures that only a small amount of nodes are selected based on their weight that depends on their account balance. It is also completely objective, meaning that the entire process is conducted purely through computation, thus adversary is unable to sway the process.

To determine the role in the given round, the node runs the `Sortition` function Algorithm 1 with corresponding inputs. At first, the `ProveVRF` function is called with node's private key, to obtain the *proof* and *proof_hash*. Then, the number of *subUsers* is computed in a way that ensures the node is selected for the role with probability corresponding to her weight (weight is equal to the stake the node currently holds). The *subUsers* are either

used as tries to obtain the highest priority for the proposed block or as number of votes the node casts for some block hash, as was described in Section 3.2

Algorithm 1: Sortition

```
Function
 Sortition(privateKey, seed, expectedUsers, role, weight, totalCurrency):
   < proof, proof_hash > ← ProveVRF(privateKey, seed||role)
   probability ← expectedUsers / totalCurrency
   subUsers ← 0
   while proof_hash / MAX_HASH ∉
     [ Σ_{k=0}^{subUsers} Binom(k, weight, probability), Σ_{k=0}^{subUsers+1} Binom(k, weight, probability) )
    do
     | subUsers ++
   return < proof, proof_hash, subUsers >
```

To verify the truthfullnes of the obtained block or vote broadcasted by some node A, other nodes can run `VerifySortition` function Algorithm 2. This function first runs `VerifySortition` where only public key of the node A is needed to verify that the *proof* and *proof_hash* were created by the holder of the private key corresponding to the given public key. After success in this step, algorithm follows with the same steps as in `Sortition` function, but it instead returns only the number of *subUsers* as that is the only value that was computed during this process.

Algorithm 2: VerifySortition

```
Function VerifySortition(publicKey, proof, proof_hash, seed, expectedUsers,-
 role, weight, totalCurrency):
   if not VerifyVRF(publicKey, proof, proof_hash, seed||role) then
     | return 0
   probability ← expectedUsers / totalCurrency
   subUsers ← 0
   while proof_hash / MAX_HASH ∉
     [ Σ_{k=0}^{subUsers} Binom(k, weight, probability), Σ_{k=0}^{subUsers+1} Binom(k, weight, probability) )
    do
     | subUsers ++
   return subUsers
```

Chapter 4

LaKSA

LaKSA (Large-scale Known-committee Stake-based Agreement protocol) [28] is a PoS protocol that addresses downsides of Algorand [13], mainly the too much varying number of leaders and voters in given rounds. It also promises to bring better security, throughput, higher lightweightness and DOS resistance compared to Algorand.

4.1 Block structure

LaKSA works on a blockchain, such that each block contains a set of transactions, a link to the previous block, and various metadata. The structure of a block is as following

$$B = (i, r_i, H(B_{-1}), F, V, Txs, pk, \sigma),$$

where:

- $\boldsymbol{i}$ is the round number
- $\boldsymbol{r}$ is a random value generated by the leader
- $\boldsymbol{H(B_{-1})}$ is the hash of the previous valid block
- $\boldsymbol{V}$ is a set of *votes* that support the previous block (B_{-1})
- $\boldsymbol{F}$ is a set of known, to the leader, forked blocks that have not been reported in any previous known blocks
- $\boldsymbol{Txs}$ a set of transactions included in the block
- $\boldsymbol{pk}$ is a leaders public key
- $\boldsymbol{\sigma}$ is a signature created by the leader over all previous fields except pk

Every block B supports its predecessor B_{-1} by including votes of nodes who were elected to vote in B's round and who vouched for B_{-1} as the last block on their preffered chain.

LaKSA does not assume any specific structure of transactions. This means that models used in various systems should be easy to implement with LaKSA.

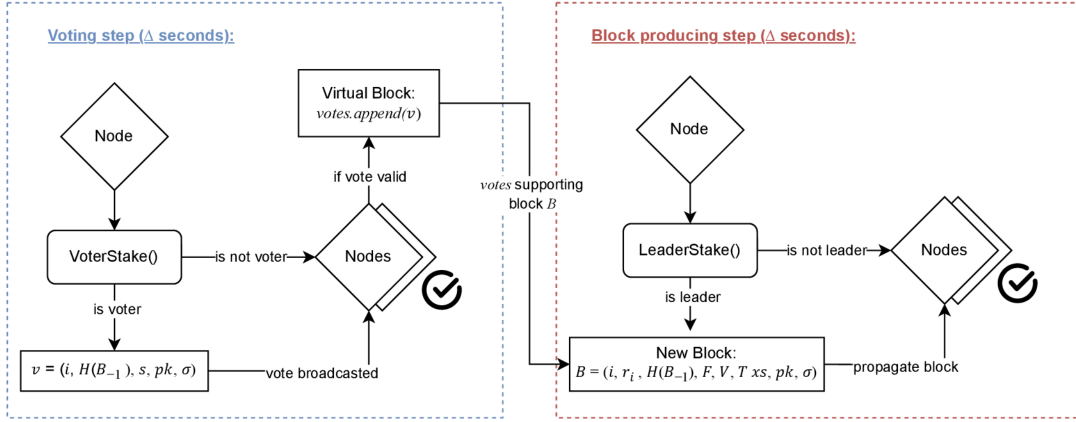

Figure 4.1: Round consisting of two steps in LaKSA

4.2 LaKSA model

Protocol model is based on two-step rounds (Figure 4.1), each lasting Δ seconds. In first step of round i, each node checks whether it can vote in round i by calling *VoterStake()*. This function returns positive number, if a node has some stake that can be used in round i for voting. In such case, the node is called a *voter* in round i and is eligible **to vote for the last block of what it believes to be the main chain**, in order to support this chain. For voting procedure, the node creates a structure

$$v = (i, H(B_{-1}), s, pk, \sigma),$$

representing a vote with folowing fields:

- $\boldsymbol{i}$ is the round number
- $\boldsymbol{H(B_{-1})}$ is the hash of the previous valid block
- $\boldsymbol{s}$ is the stake that the voter was selected with for the round i
- $\boldsymbol{pk}$ is the voter's public key
- $\boldsymbol{\sigma}$ is a signature, created by the voter over all previous fields except pk

The process of voting is as follows: Node creates a vote v and broadcasts the vote immediately to the network. Other nodes validate each recieved vote by checking whether:

a) it is authentic, formatted correctly and not with round number exceeding i (not from the future)

b) it points to a valid previous block

c) voter is legitimate (eligible to vote by *VoterStake()* function)

If verification is successful, the vote is added to the pending list of votes, also called **virtual block**, that directly support its predecessor block.

After Δ seconds of the first step of round i (during which voters vote for their oppinions and votes are validated), nodes proceed to the second step. At first, the node determines whether she is the leader by calling *LeaderStake()*, and if it returns a positive value, the node is a *leader* in that round. Next node determines the main chain, then creates and propagates a new block that has the main chain's last block as its predecessor and which includes all collected votes and the generated random value r_i. Nodes that recieve a new block check whether:

a) it is authentic, formatted correctly and not with round number exceeding i (not from the future)

b) it points to a valid previous block

c) its votes are correct

d) leader is legitimate (eligible to be a leader by *LeaderStake()* function)

After successful block validation, it is appended to its corresponding chain.

4.3 Leader and voter election

Election in LaKSA (Algorithm 3) is based on a cryptographic sampling method (further reffered to as *Sample()*), which creates an array of all stake units and pseudorandomly samples a fraction from it. It uses uniquely generated PRF outputs to sample stake units. The sampling method returns a list of sampled public keys, each corresponding to a stake unit. Thanks to the sampling method, the number of voters and leaders in each round is fixed (instead of highly varying as in Algorand [13]).

Algorithm 3: Leader/voter election via cryptographic sampling

```
Function VoterStake(pk, r):
    tmp ← Sample(q, r, 'vote')
    return tmp.Count(pk)
Function LeaderStake(pk, r):
    tmp ← Sample(l, r, 'lead')
    return tmp.Count(pk)
Function Sample(size, r, role):
    tmp ← []
    res ← []
    foreach pk ∈ stake do
        for s ← 1 to stake[pk] do
            tmp.Append(pk)
    for i ← 1 to size do
        k ← PRF(i, r || role)  mod Len(tmp)
        res.Append(tmp[k])
        tmp.Delete(k)
    return res
```

Nodes then find out thier roles by calling *VoterStake()* and *LeaderStake()*, functions, that run *Sample()* and **return how many times the given public key is present in**

the sampled stake. *Sample()* takes *role* as a parameter/seed, in order to provide different results for leader/voter election, as algorithm remains the same for each election. It also takes parameter *size*, which represents number of stakes to be sampled. If we seek to achieve highest performance, we may set *size* to 1 in call to *LeaderStake()*.

Limitations of this approach are that an adversary may try to launch an adaptive attack (DOS), as elected nodes are known before they broadcasts their messages. LaKSA proposes usage of network anonymization, namely Dandelion [5, 11], to mitigate this issue. Another approach it suggests is to elect nodes using cryptographic primitives with secret inputs (similarly as VRFs in Algorand, or unique signatures).

Pseudorandom sampling function used in *Sample()* is dependant on the value called **random beacon**. Random value *r*, included within each created block, is used in procces of random beacon creation. For security reasons, it is important for these beacons to be difficult to bias by adversaries. If the adversary is able to manipulate random beacon, then she can change outputs of *VoterStake()* and *LeaderStake()*, which can result in selection of nodes controlled by the adversary for voting and block creation. Furthermore, if the beacon is too predictable, such blockchain network is vulnerable to adaptive network-level attacks (e.g., DoS), as voters and leaders can for some round *rd* can be calculated ahead of the *rd* start.

LaKSA in its implementation uses approach by Dain el al. [9] (who used this principle in their Snow White consensus protocol), where beacons are generated purely from the random values aggregated over the main chain blocks. But this approach is not strict and other approaches can be used for implementation of random beacon creation.

Chapter 5

COPOR

COPOR (Consensus Protocol with Native Onion Routing) [35] is also belongs to the family of PoS protocols. It tries to achieve DoS resistace by anonymization of messages containing blocks via onion routing, to prevent the adversary able to intercept network traffic from indentifying the leader in the given round.

5.1 Block structure

COPOR assumes the blockchain, in which each block,

$$B = (BH, Txs),$$

contains header ($\boldsymbol{BH}$), the list of contained transactions ($\boldsymbol{Txs}$) and the signature of the header made by the round leader ($\boldsymbol{\sigma}$, contained in the header). The header,

$$BH = (id, H(BH_{-1}), txsRoot, mptRoot, pk, rand, altIdx, \sigma),$$

consists of fields:

- $\boldsymbol{id}$ - the incrementing counter of all created blocks (starting from 0)
- $\boldsymbol{H(BH_{-1})}$ - the hash of the previous valid block header
- $\boldsymbol{txsRoot}$ - the root of all transactions included in the block
- $\boldsymbol{mptRoot}$ - the root hash of Merkle Patricia trie aggregating all account states
- $\boldsymbol{pk}$ (also reffered to as $\boldsymbol{coinbase}$) - the pk of the node that is the leader of the block (and created the block). It is used for signature verification.
- $\boldsymbol{rand}$ - randomness of the current round. It is created from a signature made by the leader of the current round on the randomness from the previous round (i.e., block).
- $\boldsymbol{altIdx}$ - index of the alternative leader who created the current block in the case higher responsible leader was not available. $altIdx = 0$ for the main leader, $altIdx = 1, 2, 3, \dots$ for alternative leaders.
- $\boldsymbol{\sigma}$ - the signature of all above fields in the header made by the creator of the block.

COPOR assumes following transaction structure for simplicity (it can be easily exteneded):

$$Tx = (src, dst, val, fee, \sigma)$$

where $\boldsymbol{src}$ is the address of the sender of transaction, $\boldsymbol{dst}$ is the address of the recipient of the transaction, $\boldsymbol{val}$ is the value sent from the sender to recipient, $\boldsymbol{fee}$ is the amount that is payed to the leader who creates the block and $\boldsymbol{\sigma}$ is the signature of the transaction made by the sender.

5.2 COPOR model

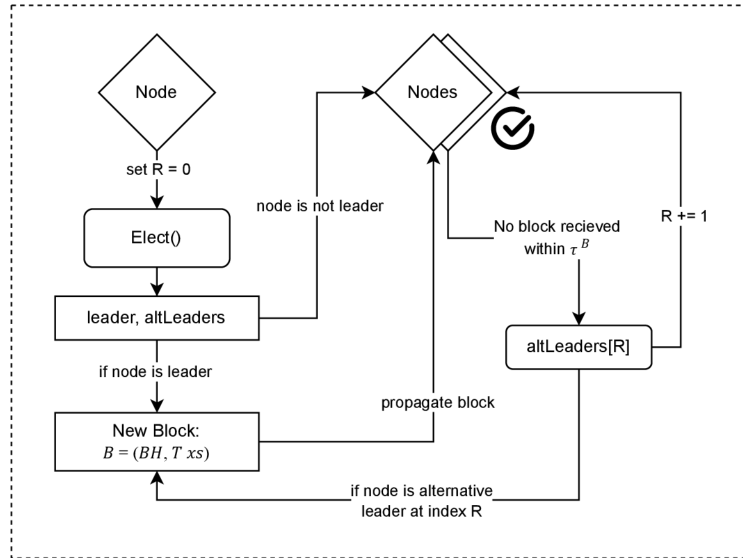

Figure 5.1: Round in COPOR. Each node first sets timeout for block reception R to 0, and if she does not recieve the new block within τ^B seconds, it checks whether she is not the round alternative round leader. If yes, it procceds with block creation, otherwise it increments R and waits for another τ^B seconds for a new block.

Protocol model is single round with single leader. At the round start, node resets the counter R of round-timeout expirations. Then a node checks whether it is the leader of the round based on the randomness from the previous round. This randomness is obtained from the block created in the last round, stored in *rand* field, and passed to function *Elect()*, which determines index of the leader and indexes of alternative leaders. **If node is the leader**, it creates and gossips new block, including a subset of transactions collected in the mempool (subset because we select as many txs as we can, but we cannot degrade difficulty of the block). **In case node is not the leader**, node sets the synchronous timeout τ^B and waits for the reception of the block. When node recieves a block, it checks whether:

a) it points to a valid previous block

b) the signature of header is valid (using the *coinbase* field as public key for signature verification)

c) the node that produced block is truly the elected leader (comparison of *Elect()* function with *coinbase* field)

When all checks pass, the protocol proceeds with rewarding of the leader and all alternative leaders. Afterwards, the block is appended to the last block of main chain.

In contrast to LaKSA, COPOR takes into account a case when the leader might be offline. In such case, a node does not recieve a new block within timeout τ^B, and therefore an alternative leader is selected and the round is restarted (as can be seen in Figure 5.1). LaKSA somewhat tackles this issue by allowance of multiple leaders in a round, but even in such case, it can still happen that all round leaders are offline and no new block is produced.

5.3 Leader election

Leader election in COPOR (Algorithm 4) is done via iteration over ids of all the present nodes. Leader is selected in the first iteration, in accordance to the *rand* field from the previous created block. Alternative leaders are selected in the following iterations, where each time *currentRand* is updated by hashing its current value and other leaders are chosen based on this new value. This procedure ensures that all the nodes select the same leaders in the given round.

Algorithm 4: Elect

```
Declaration of Types and Constants:
MAX: the size of interval used for election
ALT: the desired number of alternative leaders
rand: randomness for the current round
stakesum: sum of all stakes in blockchain

Function Elect(rand, altIdx):
    slider ← 0
    for node in sortByIds(nodes) do
        intervalSize ← (node.stake × MAX) / stakesum
        slider ← slider + intervalSize
        intervalEnds[node.ID] ← slider
    leaders ← []
    leftToPick ← ALT +1
    currentRand ← rand
    while leftToPick > 0 do
        for node, intervalEnd in intervalEnds do
            if currentRand ≤ intervalEnd then
                selectedNode ← node
                break
        if selectedNode ∉ leaders then
            leaders.append(selectedNode)
            leftToPick − −
        currentRand ← h(currentRand)  mod MAX
    return (leaders[0], leaders[1 :])
```

Chapter 6

Comparison

In addition to comprehensive look at the mechanisms behind Algorand Chapter 3, LaKSA Chapter 4 and COPOR Chapter 5 protocols, we also provide their comparison by by parameters that are crucial for blockchains in the following sections.

6.1 Throughput

Algorand commits the new block in the best case in less than 4λ seconds. The exact time cannot be predetermined, as time spent in each phase of the protocol (except the block proposal phase, that exactly last for 2λ seconds) depends on when the threshold for given phase is reached. The sooner the threshold is reached, the more quicker the block get commited and transactions executed. So the throughput is dependant on the λ value and thresholds values (*threshold_filter* and *threshold_cert*), that affect how long it will take to commit the block, where the it will take is between $\langle 2\lambda, 4\lambda \rangle$ seconds. The theoretical throughput of Algorand is 7500 txs per second, with the actual value being somewhere betweeen 10-20 txs per second as of time of the writing [1].

LaKSA throughput is highly dependant on Δ value of round steps. Throughput is likely to be high if the network is synchronous - which can be achieved by choosing a sufficiently high value for Δ. However, transaction throughput can only be high if Δ is low. So a sensible Δ value has to be choosen in order to achieve optimal high throughput. As experiments proved, it is better to choose different Δ values - Δ_1 for waiting for votes and Δ_2 for waiting for blocks, where $\Delta_1 < \Delta_2$.

COPOR throughput should be slowed down by anonymization layer, but as the experiments showed, it has little to no effect on throughput. Another thing that should influence throughput is timeout τ^B. It is analogous as in LaKSA - if it is too high, we might end up waiting too long for an offline leader to produce the block before switching to the next alternative leader, and if it is too low, we might belittle network lattencies and end up switching between round leaders too much before finally recieving the block.

So from my deduction, throughput of all protocols should be the closest to each other if $\Delta_1 + \Delta_2 = \tau^B = 4\lambda$. If we want one protocol to outperform the other in terms of throughput, we need to break relationship defined above. However, we have to take in account that COPOR does not involve voting and is more lightweight in this sense, which can result in significant throughput increase compared to other mentioned protocols.

[1] https://chainspect.app/dashboard?sort=name&order=asc&range=30d

6.2 Scalability

The higher the number of participating nodes in the consensus, the higher the probability for some of them being chosen to produce block or vote by `Sortition` function. This helps Algorand to make decision in every phase of the round. Furthermore, authors of the Algorand argue that with increasing number of users, the time it takes for message to be gossiped to every present node does not increase significantly, as the peers are chosen randomly and it is advised to replace them frequently to prevent the problem of some of them becoming offline.

Thanks to probabilistic definition of safety, LaKSA can scale to systems with thousands of active nodes. For running of its protocol, it just needs q nodes from n present nodes to achieve consensus in voting and then l leaders (also from n present nodes) to produce a new block. This means that the protocol needs to pass messages only betweeen $q + l$ nodes, which is only a subset of all participating/present nodes. Thanks to this, the protocol can scale to large quantity of nodes without significant impact on throughput.

COPOR consensus protocol removes need for voting, and leader is determined with usage of probabilistic function. Thanks to this, there is no need for any message passing except for propagation of newly created block by the leader. Because of this message lightweightness, protocol can scale without any limitations.

6.3 Liveness

Algorand relies on the assumption of strongly synchronous network, which if fulfilled esures that all users will eventually reach consensus on a block and proceed to a next round. The issue is that, if the network is not strongly synchronous, forks can be created and the nodes may split into smaller groups on different forks. If the groups are too small, they may have problems with the new blocks commitments, as they will not have enough participants to cross the predefined thresholds. To tackle this problem, Algorand can issue special recovery phase, where the nodes will vote for one of the existing forks as the fork that everyone should continue with.

LaKSA prooves that it achieves liveness after $t = 2t'$ rounds, because after those rounds at least one honest node appends a block to the main chain and the block is committed by every honest node. They argue, that for adversary to overthrown the chain, she has to be elected for leader m' times, where $m' < t'$. The adversary is elected as round leader with probability proportional to her stake possession $\alpha < \frac{1}{3}$. Therefore, the probability that at least one honest node adds a block in those m' rounds is $1 - \alpha^{m'}$, which they prove that with increasing t goes to 1. Furthermore, they say that for some t', at which p-value (representing threshold probability for transaction rejection) is below user specified $p*$, user can commit her transaction, which proves liveness. Protocol also rewards nodes not only for creating blocks (leading), but also for voting for blocks, which also supports liveness of blockchain.

COPOR supports liveness by rewarding the alternative leaders with partial reward and the main leader with leader reward and transaction fees. Partial rewarding incentives the alternative leaders to spread newly created blocks across the network even if they do not created them, because they recieve some reward for being active and honest.

6.4 Finality

The low possibility of forks results in strong finality in Algorand protocol. When no forks are observed, we can assume that the block will never be reverted. If there are some forks, we have to wait for consensus on the fork to continue on to see if block was reverted or not. However, mechanism behind Algorand highly prevent such situation from heppening.

From my understanding, LaKSA assumes finality with some probability. User can be pretty sure that her block won't be overturned with meaningful $p*$ value after about a hundred rounds, which is for the protocol that assumes high throughput reasonably short time (for 1MB blocks, it can around 2 minutes). In conclusion, LaKSA assumes chain is final and won't be overturned with some probability, believing that the adversary is not able to control enough nodes to create chain fork stronger than the main chain.

COPOR incorporates finality with usage of concepts from CASPER[6] protocol. It uses concept of check-pointing after each C blocks. Thanks to this, the main chain cannot be overturned after a checkpoint ck by the other stronger chain, because this stronger chain would include blocks created before checkpoint ck, in order to be stronger than the main chain. The above described case could happen when a node that was offline during its leader role returns online and tries to create a block, which could then accomodate other blocks and create chain that is stronger (has higher quality, because it is made by more responsible leaders).

6.5 Safety

With overwhelming probability, all nodes agree on the same transactions in Algorand protocol. Authors of the Algorand also argue, that safety is assured even in times of a „weak synchrony“, meaning that the network can be asynchronous for a variably long period of time after which it becomes again synchronous for a reasonably long period of time. With this assumption holding true, the network can start the recovery phase and bounce back from possible forks. To put it more formal, the weak synchrony assumption is that in every period of length b (a day or a week), there must be a strongly synchronous period of length $s < b$ (an s of a few hours suffices).

LaKSA seeks probabilistic safety. It assumes probability for the block to be overturned in the future, and if it is low enough (lower than user defined $p*$), voters might vote for this block. In other case, voter does not vote for the newly produced block B_i, and with high probability, if there comes to live a new fork of the main chain with more supporting votes that does not include transaction in block B_i, transactions in this block will not be considered and node will lose it's reward for producing of this block.

To achieve safety, COPOR again uses concept of check-pointing from CASPER[6] protocol. There is one case in COPOR when safety can be violated, and that is when the main leader of the round is offline. She might come online and try to produce block, that would produce stronger chain than the main chain accepted so far. But with check-pointing, she is only able to overturn the main chain if newly produced block is created after last made checkpoint ck. Blocks that are to produce stronger chain than the main chain created before checkpoint ck won't be considered. Therefore safety is assured for the main chain after each checkpoint within C rounds.

6.6 Fairness

The fairness of Algorand's consensus protocol is highlighted by its usage of VRF function, that enables any user holding even a small number of stake to participate in the consensus protocol. Furthermore, nodes are incentivized to not forge votes, as such vote will be disclosed after verification by ProveVRF function, and vote honesty, as dishonest voting only slows down the process of block commitment. Apart from this, there is no collateral incetivization for voters and leaders mentioned,

LaKSA minimizes the reward variance by rewarding nodes for voting as well as for leading every round. Rewards are given uniformly at random and proportional to stake possesion. Authors of the protocol claim that even for a node with small stake possesion $\beta = 1$, in a setting where $n = 10000$, $q = 200$ and $l = 1$, she would recieve voter reward every 50 rounds and a leader reward every 10^4. With frequent rewards for active nodes that are honest about their views, LaKSA achieves fairness.

Fairness in COPOR is achieved by usage of a weighted pseudo-random function for leader selection, where the weight is proportional to the node's stake. So in other words, a node is selected for a leader x times, where x is proportional to portion of node's stake in the sum of all stakes. COPOR also rewards alternative leaders with partial reward, which diminishes variance in rewarding of the participating nodes. All nodes are rewarded for being honest and spreading new blocks across the network which supports fairness of protocol.

6.7 DoS Resistance

The decision of Algorand to use VRF function to randomly and unpredictably determine leaders and voter is the main driving force behind its DoS resistance. Because it is impossible for the attacker to identify the leader with the highest priority or the voter with the most significant number of votes, he is unable to disrupt the concensus, as even if he manages to DoS some portion of the network, there will be with high probability select leaders and voters amongst not DoS-ed nodes who will continue and reach a consensus.

LaKSA authors claim, that DoS resistance can be achieved with usage of network anonymization techniques such as onion routing[10] or Dandelion[5, 11]. These techniques would hide leader identity in the current round and make it more difficult for the adversary to prevent the leader from creating a new block. However protocol in default setting assumes no anonymization of messages and therefore does not provide DoS resistace from scratch.

DoS resistace in COPOR is achieved by native node anonymization. Id of the leader of the round r is known at the end of the round $r-1$, but it is not possible to obtain network address of the leader from her id. Similarly, it is unfeasible for the adversary to DoS all the peering nodes of the leader since the adversary does not know the role of peering nodes in the anonymization layer. Therefore COPOR provides in its concept robust mechanisms for assurance of DoS resistance.

In the Table 6.1 is the summary of the comparison provided in sections above.

	Algorand	LaKSA	Copor
Throughput (theoretical)	7500 tx/s	$450 - 1300$ tx/s	760 tx/s
Scalability	scales well	scales less well	scales well
Liveness	proven	proven	proven
Finality	instant in the case of strongly synchronous network	probabilistic finality	periodical checkpointing to finalize blocks
DoS resistace	inherent to usage of the VRF	suggestion to add Dandelion as mitigation	achieved by the utilization of the anonymization layer

Table 6.1: Table containing results of different serialization methods

Chapter 7

Network anonymization techniques

There exists a number of approaches to anonymization of network identifiers (IPs). User can for example use proxies or VPNs to hide its real source address or use internet service providers (ISP) based solutions [23] that provide faster access to web services, but these solutions put some kind of trust into infrastucuture runned by ISP, which implies other security risks. This section will focus on onion routing based services as they are the building block for DoS resistace of COPOR.

7.1 Onion routing

Focus of onion routing is to protect network from traffic analysis, eavesdropping and other attacks both by users inside and outside onion protected network. It also tries to tackle a problem with censorship blocking access to certain sites or services. First idea of onion routing emerged in 1996. Since then 3 generations [34] have been developed, generation 0 in US testbed, generation 1 was abandoned because it was deemed as clumsy and crufty by the time it came into implementation phase, and some of its ideas were transformed into Tor [10, 12] (short for The onion router) network. Tor is a second generation onion routing protocol. We will further assume onion routing as proposed by Tor protocol, and describe its features.

Circuit establishment

Each node in Tor network, that wants to communicate anonymously, first establishes circuit (depicted in Figure 7.1) throught which messages will be send. Tor typically uses one circuit with 3 nodes in this circuit. Generally, a node can extablish **m** circuits with **n** nodes in it. Tor uses directory service to provide a list of trustfull nodes, which serve as an entry point for process of circuit establishment. Now, let us assume a node A, which wishes to create a new circuit. First it selects a node (OR1) from this directory, and sends to it a message ***Create c1,*** $\boldsymbol{E(g^{x1})}$ where $\boldsymbol{E(g^{x1})}$ stands for a nodes part of Diffie-Hellman symetric key encrypted by A's public key and $\boldsymbol{c1}$ stands for A's internal designation of circuit from A to OR1. OR1 then responds with ***Created c1,*** $\boldsymbol{g^{y1}}$***, H(K1)***, where $\boldsymbol{g^{y1}}$ is OR1's part of Diffie-Hellman symetric key and $\boldsymbol{H(K1)}$ is the hash of the key $\boldsymbol{K1 = g^{x1y1}}$.

Once the circuit is established, the node A can communicate with OR1 using symetric encryption by key $\boldsymbol{K1}$ created before. In order to extend circuit further, A sends a ***Relay c1(Extend, OR2,*** $\boldsymbol{E(g^{x2})}$***)*** to OR1. OR1 unwraps the message and sends ***Create c2,*** $\boldsymbol{E(g^{x2})}$ to OR2, where $\boldsymbol{c1}$ is OR1's internal designation of circuit from R1 to OR2.

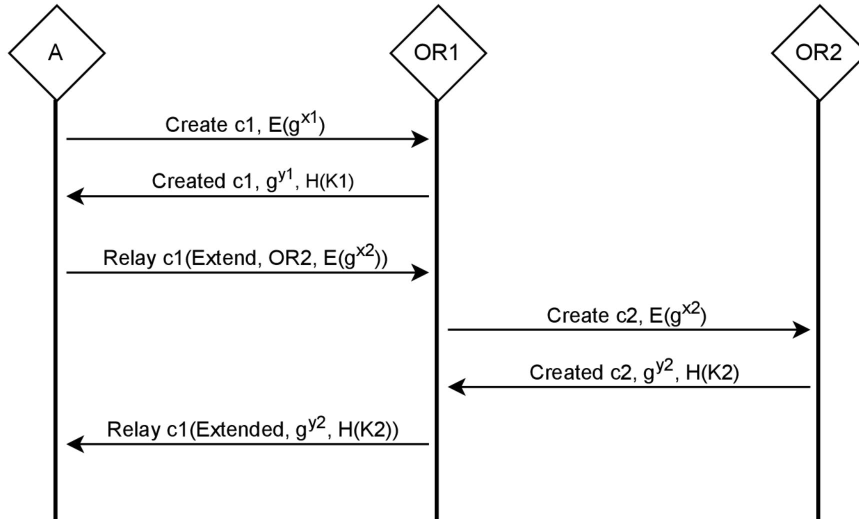

Figure 7.1: Circuit establishment

OR2 decrypts the message, creates her part of symetric Diffie-Hellman key and send message ***Created c2, g^{y2}, H(K2)*** to OR1. OR1 then responds to A with message ***Relay c1(Extended, g^{y2}, H(K2))***.

To extend the circuit to a third OR or beyond, the node proceeds the same as above, always telling the last OR in the circuit to extend one hop further.

This protocol for circuit creation achieves unilateral entity authentication (A knows she is sending message to the OR, but the OR doesn't care who she is communicating with, as A sends no public key to OR and remains anonymous) and unilateral key authentication (A and the OR agree on a key, and A knows only the OR learns it).

Message sending/recieving (relaying)

Upon reception of a relay message, OR looks up the corresponding circuit (by identifier **c**, that is in message header), and then decrypts the relay header and payload with the session key for that circuit. If the message is headed from the node, OR then checks whether the decrypted message has a valid digest, and if yes, she proceeds as described above. In case message is headed to the node, OR looks up the **c** identifier of the next circuit and OR**c**, and sends the decrypted relay message to the next OR**c**.

If the node wishes to send relay message to a given OR*i*, it assigns the digest and then iteratively encrypts the message relay header together with payload with symmetric key of each OR in the circuit up to OR*i*. Because the digest is encrypted to a different value at each step, only at the recipient OR*i* will it have a meaningful value after decryption. This *leaky pipe* circuit topology allows the node to to choose different ORs as exit points for message transmission.

When an OR*i* sends a reply to the node with a relay message, it encrypts relay header together with payload with a single symmetric key it shares with the node and send it to the next OR*i-1* in the circuit. Subsequent ORs encrypt the message further with the symmetric keys they share with the recieving node. Upon message reception, the node decrypts the message with the keys she shares with ORs in the circuit, in a manner that the last decryption is done with key K*i* shared with OR*i*.

How the node A can communicate anonymously with a distant service (server) using the established circuit is depicted in Figure 7.2.

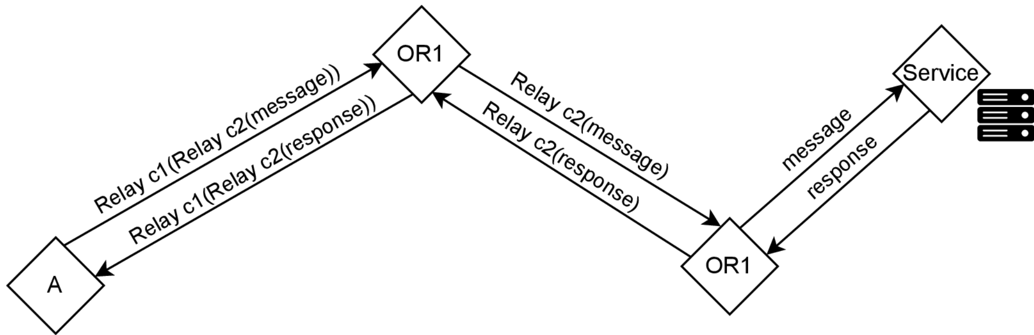

Figure 7.2: Anonymous communication in Tor network.

7.2 Mixing networks

Mixing of network traffic is another popular technique for network anonymization, sometimes used in conjunction with tor [29]. The first notion of mixnets was by David Chaum in 1981. These mixnets are based on public key infrastucuture (PKI) [31], where the user that wants to send an anonymous message m through chosen set of mixers X of size i has to first encrypt it with public key of each mixer in a manner similar to the onion routing. The resulting message is first encrypted by the last mixer in the chosen set, then by the previous one etc., until it is finally encrypted by the first mixer (the resuting message is of structure $m_{enc} = E_{pk_{x1}}(E_{pkx_{x2}} \dots (E_{pkx_{xi}}(m))))$. Each mixer than decrypts the message with its private key and when enough messages intened for the same next mixer are accumulated, they are randomly reordered to make adversary unable to link individual messages with one traffic flow. This type of mixnets is called **decryption mixnets**. If there would be no wait for certain number of messages, this anonymization would be equal to tor anonymization, with one crucial difference, which is that the adress of the next mixers have to be send together with the encrypted message, because there is no circuit like in tor, and messages can be send to arbitrary chosen mixers.

Re-encryption mixnets, based on properties of ElGamal encryption, were developed to address the following problems with previous model [30]:

a) The onion layer gets thinner at each mixer.

b) The sender is able to trace its onion encrypted message as it traverses through the mixers.

c) The sender is required to encrypt as many times as the number of chosen mixers.

d) The decription is performed in a predetermined sequence.

In these mixnets, the senders first derive the public key of the chosen set of mixers, and then encrypts the whole message with this key. Each mixer still needs to decrypt the message with its private key, but but its then reencrypted with together with random string to address the **b** issue. These mixnets, in contrast to decryption mixnets, do not transmit

routing information and therefore the set of mixer thought which the sender passes his messages need to be fixed and known before.

The described models are just one of too many developed mixnets models. Furthermore, these models also utilize sending of dummy messages when low traffic, to send messages with low delays and also to not undermine the security of message reordering, as reordering with less than predefined treshold of messages makes it easier for the adversary to identify the traffic flow of each of the messages.

7.3 Dandelion

Dandelion [5, 11] was specificly created as a lightweight mechanism to adress problem with bitcoins transaction spreading. The way it spreads without dandelion, the attacker can learn graph paths and patterns, and traceback transactions to their origin. Dandelion adresess this isssue by alternation of transaction spreading (Figure 7.3), where transaction is initially sent only to one peer in **stem** phase. At each peer, the coin is flipped (meaning some random value is generated), and based on its result, the peer continues in stem phase, or starts the **fluff** phase, where transaction is "diffused" to multiple peering nodes [4]. It is important to note that dandelion does not add any encryption of message content, it only makes it harder for the network traffic watching adversary to traceback transaction to its originator.

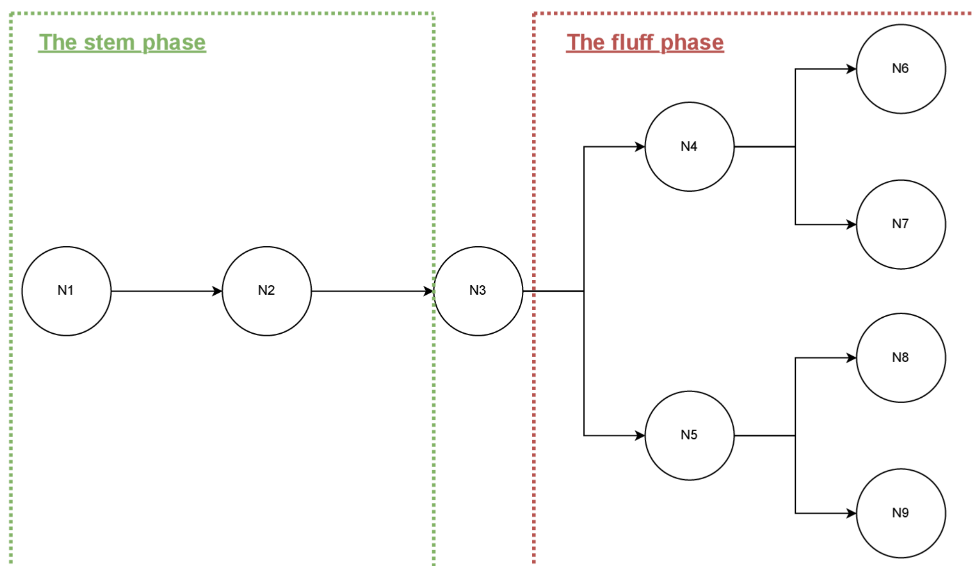

Figure 7.3: Stem and fluff phases of transaction passing in dandelion [14].

Chapter 8

Implementation details

The **python 3.8** was as the programing language used for implementation.Some of the created python modules and primitives are shared among all of the protocols, like transaction, mempool and global state structure along with network primitives for processing incoming and outgoing messages and encryption utilities. As transaction structure is shared, we can refer back to transaction structure of COPOR. We utilize both symmetric and asymmetric encryption in our implementation. For signing of transactions and blocks, we use ecdsa algorithm on eliptic curve secp256k1. RSA with key size of 4096 bytes is used for encrypting messages during circuit establishment phase (which is needed in order to use anonymization). And at last, AES256 is used for symmetric encryption when anonymization is turned on.

The existing implementation of **COPOR** was only slightly adjusted, fixing minor bugs and changing the way of how certain values are stored in memory to relief its usage, storing them as bytes instead of hexadecimal strings.

To implement PRF used in **LaKSA**, we used `hmac` function provided by the standard python library (as it turned out to be faster than implementations from external libraries by our tests). LaKSA assumes the stake is of integer type, but we allowed for the stake to have a floating point value. Because of this, the stake value of the node has to be rounded to nearest integer value, and then sampled to array with the lenght equal to the stake in the `Sample` function. We also had to create the vote class representing the vote and implement the corresponding functions to process and validate votes in order to make complete LaKSA implementation. All the other primitives were handily reused from previous COPOR implementation.

In order to implement **Algorand** implementation, we had to look for VRF function that would work with keys and signatures created on secp256k1 eliptic curve, as we used this in previous protocol implementations and did not want to change it for Algorand. We found library written in C that supports secp256k1 keys. We had to compile this library and create python module that would abstract work with it. To convert python data to C compatible data and call functions from the precompiled, we used the `cffi` library. Next thing to implement was the `Sortition` algorithm. Here we utilized `scipy` library that provided us with function to calculate binomial probability. To implement all the phases present in Algorand, we took advantage of synchronization primitives from `threading` libraries to prevent errors that could happen by accesing variables crucial for consensus progression concurently.

8.1 Network layer

We use UDP raw sockets for passing of messages, like most of the blockchains do. We allow for maximum size of payload to be $65,000$ bytes (note that maximum allowed size for UDP packets is $65,507$), and do not support larger messages. If message is larger than $65,000$ bytes, trailing bytes are stripped and not transmitted. This results in unability to process the message by the adressee.

8.2 Structure of messages

Structure of messages is the same for all implemented protocols, with minor differences. In this section, we describe each type of message that can be passed among the nodes, and it use case. Each of the messages is a dictionary, where the first key *type* contains a numeric value representing a type of the message (e.g. value 11 corresponds to message *CircuitCreate*, etc.).

- ***CircuitCreate*(type, circuit_id, key)** - sent by the nodes (let's call them A nodes) which want to establish a new onion circuit. *Circuit_id* is the proposed designation for the new circuit, *key* is the A node's part of the shared secret used to create the shared symmetric key.

- ***AlreadyUsed*(type, used_circuit_id, proposed_circuit_id)** - sent by the nodes that received the message *CircuitCreate* but were unable to create the circuit due to circuit id collision. *used_circuit_id* is the id which caused collision (and is sent back to make it possible for the other node to identify the circuit which failed to be created), *proposed_circuit_id* is the id which is collision free on the node and is send back to the other node as a proposal for possible new circuit id (this value is sent to possibly speed up the process of selection of the collision free circuit id).

- ***CircuitCreated*(type, circuit_id, key, hash)** - sent by the nodes (let's call them B nodes) that derived a shared symmetric key and established the onion circuit after receiving the *CircuitCreate* message to the node that sent this message. *Circuit_id* is the designation for the established circuit on both nodes, *key* is the B node's part of the shared secret used to create the shared symmetric key and *hash* is the hash of the derived shared symmetric key, generated by the B node. The hash is used to check that indeed both nodes derived the same shared symmetric key.

- ***Relay*(type, circuit_id, msg)** - this message is sent to the nodes in onion circuit. Field *msg* contains a message encrypted in onion manner by the symmetric keys shared with all nodes in the circuit, and *circuit_id* is the idetifier for the circuit. When the first node in the circuit recieves this message, it decrypts field *msg* with the shared symmetric key and replaces the *circuit_id* with id that is used as identifier for the route between first and second node in the circuit.

- ***Extend*(type, next_node, key)** - a node (called A node) that wishes to extend the lenght of the so far established onion circuit sends this type of message to the last node in her circuit. *Next_node* is the address of the node with whom the A node wishes to establish a new shared symmetric key and by that extend the circuit, and *key* is the A node's part of the shared secret used to create the shared symmetric key. The last node in the circuit than takes care of choosing colision free circuit id between

her and *next_node*, and after success she obtains message *CircuitCreated*, which is relayed way back to the A node as the message *Extended*, which finalizes the circuit extension and new symmetric key establishment.

- ***Extended*(type, key, hash)** - this message is the response for *Extend* message in case of success, otherwise no message is send at all (the A node that wishes to extend the circuit has to keep track of whether she recieved the response and established the shared key or not by herself, and take the corresponding action). Similarly as in message *CircuitCreated*, *key* is the B node's (newly added node to the circuit) part of the shared secret used to create the shared symmetric key and *hash* is the hash of the derived shared symmetric key, generated by the B node.

- ***Wrapped*(type, to_addr, msg)** - this kind of message indicates that the message in *msg* field is unencrypted message, that is intended to be sent to the *to_addr* node. Used by the *Relay* messages (typically field *msg* in *Relay* messages contains a *Wrapped* message).

- ***KeyExchange*(type, key)** - sent by the node (let's call her A) that wishes to establish a shared symmetric key with the other node (usually her peer). *Key* is the A node's part of the shared secret used to create the shared symmetric key. This message usage is different from the *CircuitCreate* usage.

- ***KeyExchanged*(type, key, hash)** - sent by the node (let's call her B) that successfuly derived the shared symmetric key. *Key* is the B node's part of the shared secret used to create the shared symmetric key and *hash* is the hash of the derived shared symmetric key, that is used to check that indeed both nodes derived the same shared symmetric key.

- ***SymmetricEncrypted*(type, msg)** - field *msg* contains a message encrypted by the symmetric key shared between two peering nodes.

- ***Block*(type, blk)** - field *blk* contains a block (used by both copor and laksa protocols).

- ***BlockAlgorand*(type, blk, proof, proof_hash)** - field *blk* contains a block, *proof* contains a proof generated by the VRF function and *proof_hash* contains a hash of the proof. This message is specific to Algorand protocol.

- ***Priority*(type, i, priority, proof, blk_hash)** - field *i* identifies the round (consecutive block number starting from 0), *priority* is the computed priority of the given *blk_hash* (hash of the block created in the current round *i*) and *proof* is the proof of the VRF output.

- ***Vote*(type, vote)** - this message contains in field *vote* a vote, used by LaKSA and Algorand consensus.

- ***Initialized*(type)** - sent by the nodes to all the other nodes to make them aware that they are ready to start the blockchain. Used only in the beginning, to synchronize the nodes (so they at least start at the same time).

Some messsage types were omitted in this description, as they are not currently used in the implementation. All of the messages have to be serialized (transformed) to bytes

in order to be transmitable via network. We implemented different serialization strategies in order to find how they would impact the throughput of the protocols, and also in an attempt to find serialization that results in the smallest payload size. We present these serialization types and conducted experiments later.

8.3 Message processing

Each incoming message is received by the server thread, which only task is to listen for incoming messages, and pass them further to thread (or threads) which will process it. When using json serialization, server can deal with multiple concatenated messages (e.g. we recieve two messages with blocks and accidentaly read them at once, without separation), in a manner that it will first replace all "}{" with "}},{{" in order to split it by "},{" characters. Resulting splitted array is then iterated over and each string is checked for whether it starts with "" and ends with "" (note that this check would help us to implemen processing of messages larger than specified max of 65,000 bytes, but it could not be utilized with other serialization types, and we would be also dependant on message order, which cannot be assured). If yes, we know that we have a valid json string and can process it further, if no, we can either save the string as unprocessed part and then append it to the front of the new recieved message, or discard it. Currently we append it to the new message.

So after recieving a message, it is deserialized back into the python dictionary, whose structure was described before Section 8.2. Based on the *type* field of the message, the corresponding action is taken. If the given *type* does not match to any of the defined types, error is raised.

8.4 Anonymization layer

Anonymization layer is also shared amongst all of the protocols. It takes care of circuit establishment for anonymization, encryption of messages that are to be send via established circuits, handling of encrypted messages, and establishment of shared keys for symmetric encryption. Each node which uses anonymization layer keeps 3 dictionaries (if you are not familiar with dictionaries in python, you can think of them as mapping of keys to values, which is known in other languages as map) that store circuit ids, namely **in_circuits** - keeps track of circuits established by the node with other nodes, **out_circuits** - keeps track of circuits established by other nodes with the given node and **relay_circuits** - manages circuits that were established for other nodes (the ones we have in out_circuits) with the given node, as the result of the command to extend the circuit.

To assure that no colision happens when establishing the circuit, meaning that the selected circuit id won't be used twice as the key in any of the aforementioned dictionaries, a following mechanism is used (in source code, it can be found in function `createCircuitID()`). First, a lock named *create_circuit_id_mutex* must be obtained (this lock allows only one thread to generate new circuit id at a time), and only then, a new circuit id of size 32 bits is created. Then, newly generated circuit id is searched for in the dictionaries *out_circuits*, *in_circuits* and *relay_circuits*, and if the match is found in any of the dictionaries, circuit id is regenerated until it no match is found. Then, the new circuit id is returned. We must add, that release of the *create_circuit_id_mutex* lock happens outside of outside of the `createCircuitID()` right after the newly obtained circuit id is inserted into the one of the

3 dictionaries. This is needed to prevent other thread from unintentionaly obtaining the same circuit id, because otherwise it would not yet be inserted into one of the 3 dictionaries and so marked as unused. We also implemented a mechanism to deal with colisions when a node creating the circuit picks a circuit id and this id is already used in one of the 3 dictionaries on the other node. In this case, the other node generates a new circuit id which is not yet used in her 3 dictionaries, and sends a message, where she notes that proposed circuit id was already used and proposes a new one. The node which receives this message can then accept the proposed circuit id if it is not present in her 3 dictionaries, or generate a new one try to establish circuit with it. Again `createCircuitID()` function takes care of checking whether the proposed circuit id is not present in the node's 3 dictionaries, returning the proposed circuit id if it satisfies given conditions, or a new one if not.

8.4.1 Key exchage mechanism

Exchange of the key is conducted in the same manner as in the tor protocol. The A node first creates a public-private key pair using secp384r1 eliptic curve. Public key is then encrypted by the B node's RSA public key (this key is known and obtained from some shared directory), and sent in message *CircuitCreate* to the B node. Upon reception of this message, the B node decrypts encrypted A's public key, generates her public-private keypair and uses it together with the A's public key to establish a shared key. This key is then passed to HKDF derivation function, to derive a more secure shared symmetric key. After success of all previous operations, she generates a hash of the shared symmetric key (using sha3_256) and sends it back together with the B's public key in the message *CircuitCreated*. Note that B's public key is sent unencrypted - this is because the B node does not know the identity and address of A (and such feature is also unwanted), as the sender of the message could be some node in the A's circuit, not the A node. When A recieves the message, she establishes the shared key and then passes it to HKDF function to derive a more secure shared symmetric key, similarly like the B node. Then she generates a hash of the newly created symmetric key and compares it to the hash obtained from the B node. If they are the same, shared symmetric key was successfuly established, otherwise key establishment failed. Currently in the implementation, when key establishment fails on the comparison of key hashes, whole circuit is discarded and the A node picks a new one, which she tries to establish.

8.4.2 Finding a circuit

Circuits are picked randomly before the attempt to establish them. Each time a given circuit fails to be established, it is added to the array called *invalid_routes*. If is is established successfuly, it is added to array *initialized_routes*. Upon all picked circuits were tried to be established, for those that failed we pick a replacement routes that are not present in both the *invalid_routes* and *initialized_routes*. If no such routes could be found (meaning that we tried all possible routes), runtime error is raised, resulting in failure of node start. Remark that we pick the nodes in the circuit randomly and we raise runtime error if we fail to pick such a random circuit, that is not present in both aforementioned array after attempting it for 10 times.

8.4.3 Different anonymization types

In search of the fastest block gossiping, we implemented different anonymization schemas. Each of them provides different DoS resistance. We named them *tor*, *gossip-node* and *dandelion* and describe the in this subsection.

To better explain to you these schemas, let us first state these preconditions:

- We have **5 nodes** named N1, N2, N3, N4, N5.
- Each of the nodes has **2 peers**. For our examples, we just need to know that N1 has peers (N4, N5) and N3 has peers (N2, N5).
- Each node has exactly **one onion route** (circuit) **with 2 nodes in the route** (meaning that n=1, m=2). For our examples, it is only important to know that N1 has onion route through (N2, N3) and N5 has onion route through (N4, N2).
- Following conditions are descriptions of symbols used in the figures. For two nodes N1 and N2 let **k12** denote a shared symmetric key between these two nodes. Consencutively k13 denotes the shared symmetric key between nodes N1 and N3. These keys are used by the N1 node when she wants to send an onion encrypted message via her circuit through nodes (N2, N3).
- For two **peering** nodes N3 and N5 let **Sk35** denote a shared symmetric key between them. Note that this key is different from the previously described key as it is not used in the onion circuits. Even if the lenght of the onion circuit is one (m=1), this key is not to be interchanged with k35 (that is not present in our examples, but we can imagine it's existance). Keys prefixed with k are used for encryption/decryption in onion circuits, and keys prefixed with *Sk* are used for encryption/decryption between two peers in *gossip-node* anonymization mode.
- Expression $E_{k12}(E_{k13}(N3, blk))$ means that the message $(N3, blk)$ was encrypted first by symmetric key $k13$ and then by symmetric key $k12$. E here stands for encryption. Key $k12$ could also be reffered to as $k21$, but the prior better express that N1 is the node that iniciated the circuit.
- Expression $E_{Sk35}(blk)$ means that the message (blk) was encrypted by symmetric key $Sk35$. This key could also be reffered to as $Sk53$.

In each of our examples, we are at the start of the round in Copor protocol and the leader is node N1. She wants to gossip its block to her peering nodes (N4, N5).

Tor anonymization type

In *tor* anonymization type, each node that wants to gossip the block first randomly picks one of her circuits, and then send the message to her peer through the chosen circuit. The peering node that receives the message again gossips it to her peers via one of her randomly selected circuits.

All this process is depicted in Figure 8.1, where the node N1 first has to encrypt the message in onion manner. Her onion circuit goes through nodes (N2, N3), so she first encrypt her message with key $k13$ and then with key $k12$. Resulting message $E_{k12}(E_{k13}(N5, blk))$ is then send to the node N2 together with the circuit designation (which is ommited to

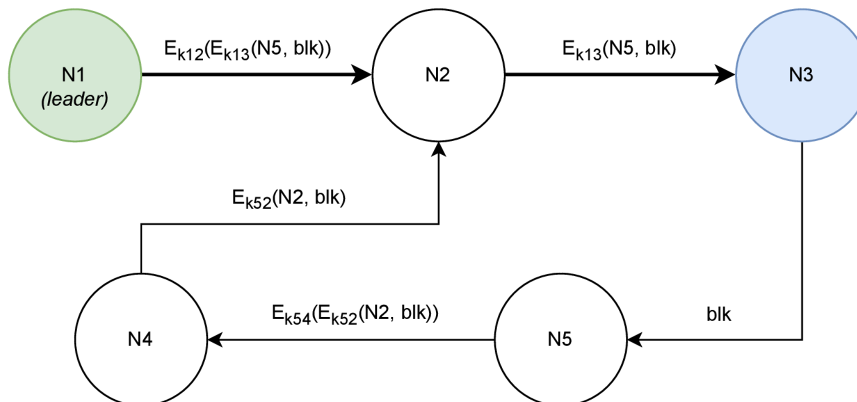

Figure 8.1: Block gossiping in the **tor** anonymization type

simplify the figure). Based on the circuit designation, the node N2 knows what key she has to use in order to decrypt the message (each circuit id has a unique corresponding shared key, you can imagine it as a mapping) and then she looks for a next hop in the given circuit. Here she finds that the next hop is N3, and sends to her the message stripped of one encryption - $E_{k13}(N5, blk)$, together with circuit designation between (N2, N3). The node N3 looks at the circuit designation and finds the corresponding key, then decrypts the message with it. Then she looks at the next hop for the given circuit, finds out that next hop does not exists and procceds to further procces the message. The message (N3, blk) states that the blk is intended to be sent to the node N5, and so the node N3 sends the unencrypted blk message to the node N5 (this effectively looks like for the attacker that intercepts the traffic as the leader node in this round was N3, as he does not know the content of encrypted messages passed between nodes (N1, N2) and (N2, N3)). The node N5 which received the block first process it, and after verifying that it is valid, she gossip it to her peers through onion circuit. In the figue Figure 8.1, she sends the message $E_{k54}(E_{k52}(N2, blk))$ to her peer N2. We know how the message will be processed on the node N4 from previous descriptions. Let us jump at the node N2, when she decrypts the message and finds no next hop exists. She finds out that the message is intended for her and so processes it further by herself (no need to send it to herself via UDP socket), and then gossips it to her peers via the onion circuit.

Gossip-node anonymization type

Gossip-node anonymization type differs from *tor* anonymization type a little bit. At first, round leader that created a block gossips it only to the last nodes in her onion circuit. These nodes then gossip the block to all their peers encrypted by the shared symmetric key they have exchanged before. Both *gossip-node* and *dandelion* anonymization type reseble the dandelion proposed in bitcoin by first sending the block through the pre-established circuit (similar to stem phase) and only upon reception by last node in the circuit, the block is gossiped to all peering node (fluff phase). Additional security could be added by randomly deciding at each circuit end-node whether to continue in the stem phase or start the fluff phase.

Again in the figure Figure 8.2, N1 sends message $E_{k12}(E_{k13}(N3, blk))$ through her onion circuit, but this time, the recipient is the last node in her circuit - N3. Upon the node N3 finds out that the message is intened for her, she processes the block and gossip it to

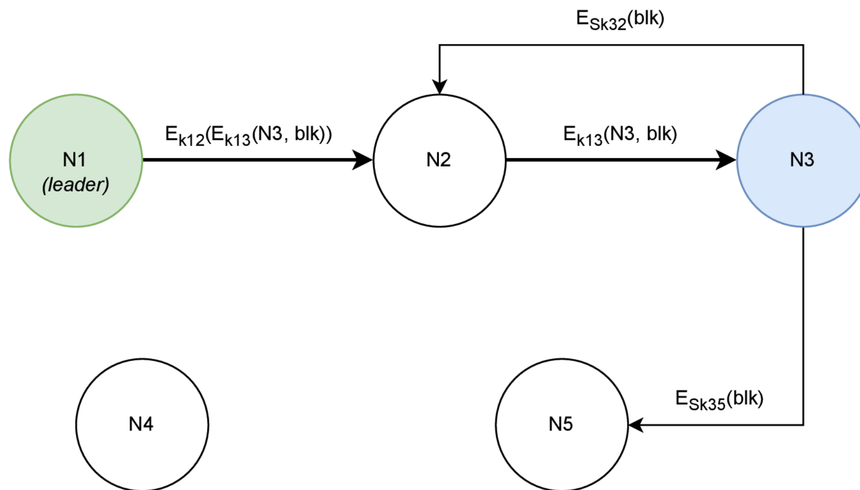

Figure 8.2: Block gossiping in the **gossip-node** anonymization type

her peers (N2, N5) encrypted by the shared symmetric key - messages $E_{Sk32}(blk)$ and $E_{Sk35}(blk)$. Nodes N2 and N5 look at the messages type (which will be *SymmetricEncrypted*) and by selecting the address of the message sender, they idenitify the key they need to use to decrypt the message (in this case, address of the peer is mapped to the shared symmetric key). After decryption, they proceed to process the message, verifying the block the block and gossiping it to their peers encrypted by the symmetric keys shared between them (these keys were agreed upon during the bootstrap process of the consensus).

Dandelion anonymization type

Dandelion anonymization type can be described as *gossip-node* anonymization type stripped of the encryption by symmetric keys shared between peers.

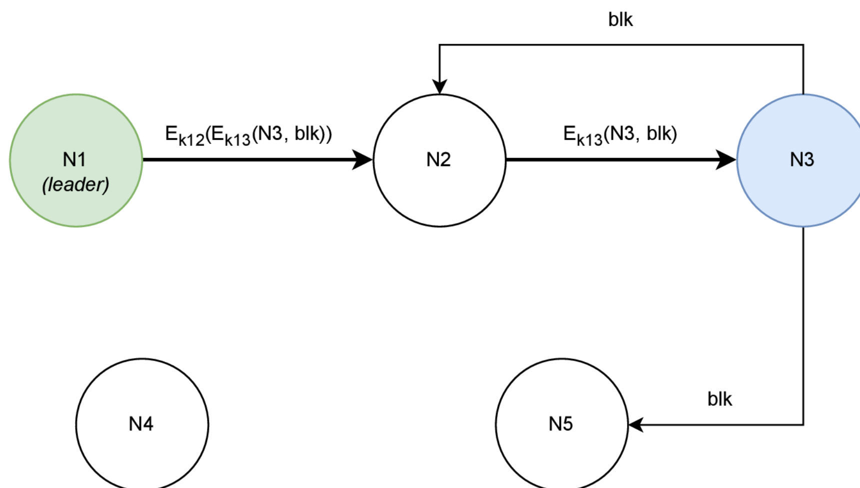

Figure 8.3: Block gossiping in the **dandelion** anonymization type

You can see it in action in Figure 8.3. Here, similarly as in previous figures, N1 sends message $E_{k12}(E_{k13}(N3, blk))$ through her onion circuit. The N3 node upon reception of

the message, its decryption and identification that it is intended for her processes the block contained in it and after successful block verification, she gossips it to her peers unencrypted. The peers that verify and procces the block send it to their peers again unencrypted. As we mentioned with *tor* anonymization type, this will for the attacker able to intercept the traffic look like the node N3 created the block, and will wrongly consider the node N3 as the leader with the id present in the block, althought the id will belong to the node N1. We will discuss possible attacks and vulnerabilities of the implemented anonymization types later in chapter Discussion.

8.5 Protocol bootstrapping

The exact process with all the necessary steps is described in the Readme.md file of the submitted repository. Here we describe the steps only to give you some insight into the bootstrapping process.

Before the consensus simulation can be runned, all nodes specified in corresponding config files (each consensus type has its own config file because some variables are consensus specific, although most of them are shared, like the array with nodes addresses, we decided to go this way to make configs more readable) must first have created their ecdsa (for transaction signing) and rsa (for encryption during onion circuit establishment) keypairs. This effectively means running the main program with the node's id and argument *--gen-keypair*. After all nodes have their key generated, the genesis blockchain configuration has to be created. This is again done by running the main program with any of the presents node's ids and argument *--gen-genesis*. This will create 2 configuration files named *blockchain.config* and *tx.config*. As we did not implemented gossiping of the transactions, all nodes generate the number of transaction specified by the GEN_MEMPOOL_SIZE configuration variable by themselves. To make all nodes start with the same set of transactions, we provide an option to first generate the transactions by running the *gen_txs.py* python script, which generates transactions and stores them to the specified file (we store them in pickle serialization format). Then you can start all the node passing them the file with pre-generated transactions via the argument.

Previous paragraph was about config generation etc., now we move onto what actually happens when you run the main program and when the nodes start to work under the terms of the selected consensus. Each of the nodes starts with 5 threads: *reciever*, *block consumer*, *message consumer*(optional), *node* and *shell*. The *reciever* thread listens for incoming messages on the node's UDP socket, deserializes them and passes them to the *message consumer* thread (if *message consumer* is not present, it will create a new thread for each message, which will take care of given message processing). The *message consumer* thread picks messages from a blocking queue named *msg_queue*, and process them one by one. How are incoming messages processed was described in more detail in section Section 8.3. When the *message consumer* encounters a message containing a block, it will serialize the block into the underlaying block object and put it into the *blk_queue* only if the hash of the obtained block is not yet present in the array with recieved block hashes. The *block consumer* thread picks blocks from the *blk_queue* as they are added and processes them in the `UponRecvBlk()`. By having a separate thread for block processing, we allow for fast processing of the other incoming messages, as we know that processing of the block takes a significantly larger portion of time (all transaction signatures have to be checked together with block signature, which is costly operation).

The *node* thread is usage is different in copor and other protocols implementions. The common usage of this thread is the wait for the starting lock. Further, in copor, it is used to start the first block creation round and after that the thread exits. Following rounds are started when new valid block is received from the round leader or peers, or if not received within τ timeout, round is restarted by the timer. In other protocols, this thread runs for the whole lifetime of the program, until the exit event is set. Copor could probably be refactored to work within the *node* for the whole lifetime, but we did not do it as it would probably not result noticable performance gain. The *shell* thread fills mempool with transactions, if anonymization is turned on it also finds and initializes configured number of onion routes and then notifies other nodes that the node is ready to start the consensus protocol. Upon recieving the confirmation of initialization from all the nodes present in the configuration file, the starting lock is released and the *node* thread can continue in its work.

8.6 Used libraries

Here are listed all external libraries (the ones that are not packed with python3.8) that were used and for what their were used.

- **rich**[1] - was used for prettier printing of responses to commands when user interface is turned on (only implemented for copor protocol). It was also used as an adapter for standard logging, when logging is set to be outputted to standard output, it is colorized by rich. Also elements like progress bar for displaying the progress of transaction creation/mempool filling were used.
- **secp256k1-py**[2] - is a wrapper to the higly optimalized implementation of ECDSA cryptography on curve secp256k1 in C language. It was used for creating block and transaction signatures and its verification.
- **sortedcontainers**[3] - was used for sorted storage of transactions in the mempools, so that the transactions with higher fees are placed as first.
- **eth-mpt**[4] - this library is used to store global state of the blockchain.
- **cryptography**[5] - functions for performing shared key derivation (HKDF), symmetric encryption/decryption (AES256 in CTR mode) and asymmetric encryption/decryption (RSA) and more were used from this library.
- **bson**[6] - another serialization library used because compared to json, it supports serialization of bytes values without prior conversion to hexadecimal string.
- **msgpack**[7] - is one of serialization libraries used. Similarly as bson, it supports serialization of bytes values.

[1] https://github.com/Textualize/rich
[2] https://github.com/ludbb/secp256k1-py
[3] https://github.com/grantjenks/python-sortedcontainers
[4] https://github.com/popzxc/merkle-patricia-trie
[5] https://github.com/pyca/cryptography
[6] https://github.com/py-bson/bson
[7] https://github.com/msgpack/msgpack-python/

- **cffi**[8] - library that allows you to call C code from precompiled libraries. Used to call VRF implemented in C (from **libsecp256k1-vrf** libraries).
- **scipy**[9] - well known scientific library. We used the `binom.cdf()` function from its module *scipy.stats* to implement sortition used in algorand.
- **libsecp256k1-vrf**[10] - library that provided us with VRF implementation supporting secp256k1, written in C language.

[8] https://github.com/python-cffi/cffi
[9] https://github.com/scipy/scipy
[10] https://github.com/aergoio/secp256k1-vrf/tree/master

Chapter 9

Testing

Serialization tests and localhost tests were runned on a machine with Windows 10 operating system utilizing Ubuntu 22.04 on WSL1. Hardware specifications of the machine:

- processor(CPU): Intel Core i5 9600KF, 3.70GHz, 6 cores, 6 threads
- memory(RAM): 2x Kingston KHX2666C13, 2401MHz, 16GB
- motherboard: ASUS ROG STRIX B360-H GAMING
- storage: 1TB SSD Intel 665p, 2TB Seagate BarraCuda (256MB cache)

Graphic card on this machine is Nvidia GeForce RTX 3060 Ti, but we consider it unused by our implemented program. However, it could be used by external libraries, such as **secp256k1-py**, but we did searched for whether it is true, and whether any of the used libraries significantly utilizes GPU.

9.1 Serialization tests

In an attempt to decrease the size of the transmitted block, we tested 4 serialization libraries: pickle, msgpack, bson and json. The old implementation used json for serialization, but it was unsuitable as all bytes values had to be encoded to hexadecimal strings. So we looked for other options.

Serialization libraries were tested on how fast they perform serialization of the *Block* message send when during Copor consensus, and also what is the size of the resulting serialized message. The block contained 10 transaction, each of the transactions had size of aproximately 258 bytes (discarding string names of the transaction fields from the size). In each tests, blocks were first serialized, then transmitted via UDP socket and deserialized back into the block object. Results are shown in Table 9.1 (where serialization/deserialization time is the rounded average of measured values.)

	pickle	msgpack	bson	json
message size (in bytes)	654	2103	2464	7954
serialization time	0.08 s	0.1 s	0.25 s	0.16 s
deserialization time	0.04 s	0.06 s	0.15 s	0.08 s

Table 9.1: Table containing results of different serialization methods

Pickle library clearly performs the fastest serialization/deserialization and also results in smallest packet size, but is uportable, in a sense that we cannot create another implementation of copor in other language, because it would be unable to perform serialization/deserialization, because pickle is a python specific library. Thanks to that, it is able to pack the object into message of the smallest possible size. But such message can only be processed by another python program.Another reason why pickle is unsuitable in our case is because we use **secp256k1-py**, that effectively allows us to use optimalized C code in our python program. Field *coinbase* in block header is a public key created by aforementioned library, and therefore stored in memory as a Cdata variable. Pickle is unable to serialize Cdata, and we have to convert this Cdata to bytes by using a method bound to the public key - *coinbase.serialize()*, as is done in all other tested serializations.

First library behind pickle is msgpack. In terms of speed, the slowdown is unsignificant, lacking only by $0.02s$ margin behind pickle. But size is almost $4x$ times the size of pickle serialized message. Bson differs slightly to msgpack in terms of size of the resulting serialized message, but speedwise it is slower by more than $0.1s$ second margin, which can pose a real problem in our case where we want to achieve high throughput. Lastly, json serialized message has a huge size, $12x$ times the size of the result from pickle and almost $4x$ times larger that result from msgpack. But speedwise it keeps the pace with pickle and msgpack, lacking $0.08s$, $0.04s$ on pickle and only $0.06s$, $0.02s$ on msgpack in serialization/deserialization respectively. This would make json a viable option if we were not also searching for the smallest possible serialization size, which json does not provides us with.

9.2 Local tests

We first tested our implementation on local machine, where we conducted a series of tests with 3, 5 and 10 active nodes. We tested each of the implemented protocols and observed how different serialization types and enabled anonymization affect the throughput (number of executed transactions).

During preparation of the tests we noticed that we can set low delays (meaning variables *tau*, *delta_1*, *delta_2*) when anonymization is turned off, but as we turn anonymization on, these low delays are unmaintainable. Sometimes some node picks an alternative leader/starts new round, because she did not yet received a block in the given round, but if the anonymization would be turned off, she would have received it. Because of this phenomenom, we introduced a slight increase to all delay variables (of 0.25 s) when anonymization is turned on to tackle this problem, as when we increase the delay, overall throughput of protocols also increases (because no unnecessary alternative leader pick/new round start happens and the node can recieve the block faster).

Specifications for tests with **3 node** (Table 9.2) are:

- 1000 transactions, block size of 25 transactions, $n_routes = 1$, $m_nodes = 1$
- **Copor** - $tau = 0.5$
- **Laksa** - $q = 3$, $l = 1$, $delta_1 = 0.5$, $delta_2 = 0.75$
- **Algorand** - $expected_leader = 26$, $expected_filter = 100$, $expected_cert = 200$, $treshold_filter = 6$, $treshold_cert = 12$, $lambda = 0.65$

The first set of tests (Table 9.2) is there to showcase the potential throughput of the protocols when the number of nodes is low. In these tests we see that *tor* anonymization

		pickle	msgpack	bson	json
Algorand	no anonymization	18.072 tx/s	18.098 tx/s	18.299 tx/s	18.039 tx/s
LaKSA	no anonymization	20.210 tx/s	20.259 tx/s	20.344 tx/s	20.196 tx/s
	tor	14.485 tx/s	14.493 tx/s	14.487 tx/s	14.484 tx/s
	gossip-node	14.486 tx/s	14.486 tx/s	14.485 tx/s	14.471 tx/s
	dandelion	14.518 tx/s	14.501 tx/s	14.486 tx/s	14.503 tx/s
COPOR	no anonymization	122.436 tx/s	118.903 tx/s	120.708 tx/s	117.7 tx/s
	tor	120.216 tx/s	120.989 tx/s	119.458 tx/s	117.11 tx/s
	gossip-node	87.809 tx/s	94.273 tx/s	87.925 tx/s	91.038 tx/s
	dandelion	92.072 tx/s	93.926 tx/s	92.392 tx/s	93.5 tx/s

Table 9.2: Table with results for 3 nodes

type does not significantly influence the throughput of the COPOR protocol. However, the other two anonymization types, *gossip-node* and *dandelion*, that were designed to be more lightweight than *tor* have decreased throughput by a noticable margin (but still relatively negligible). The reason for this is the small number of present nodes, where it is unsuitable to run these anonymization type. The number of established onion routes highly influences throughput in these anonymization types as messages containing blocks or votes are initialy only send to endpoint nodes in these routes. Only after validation and execution by the endpoint node (which takes noticable portion of time, roughly 300 ms for block of 25 transactions in our implementation), the block is broadcasted to all her peers. This fact significantly reduces throughput when number of established routes is less than number of peering nodes. One way to eradicate this issue is to gossip incoming blocks/votes regardles of their validity. However, this can secondary cause network congestion. It is important to choose which issue is more important to us and decide accordingly to either implement "instant gossiping" or not.

Further we can see that usage of different anonymization types does not have a major impact on the throughput of LaKSA, as it depends mainly on the parameters *delta_1* and *delta_2*.

Specifications for tests with **5 nodes** (Table 9.3, Table 9.4) are:

- 1000 transactions, block size of 25 transactions, $n_routes = 1$, $m_nodes = 1$
- **Copor** - $tau = 0.5$
- **Laksa** - $q = 3$, $l = 1$, $delta_1 = 0.5$, $delta_2 = 0.75$
- **Algorand** - $expected_leader = 26$, $expected_filter = 100$, $expected_cert = 200$, $treshold_filter = 6$, $treshold_cert = 12$, $lambda = 0.65$

The second set of tests shows the same throughput as before when all the nodes are online (Table 9.3). The situation gets worse when 2 nodes are offline (Table 9.4), where throughput of COPOR reduces by roughly a half in each anonymization type. Throughput of LaKSA also changes similarly. We notice that for *dandelion* anonymization type, we get better results than with the others anonymization type, but this can be purely due to the different load on the local machine caused by other processes for each running test, which also affects the results obtained and we are not able to completely ensure the fair conditions for each test (so the side workload on the local machine would not affect it).

		pickle	msgpack	bson	json
Algorand	no anonymization	18.092 tx/s	18.102 tx/s	18.179 tx/s	18.126 tx/s
LaKSA	no anonymization	20.232 tx/s	20.367 tx/s	20.323 tx/s	20.216 tx/s
	tor	14.495 tx/s	14.459 tx/s	14.482 tx/s	14.483 tx/s
	gossip-node	14.479 tx/s	14.411 tx/s	14.431 tx/s	14.424 tx/s
	dandelion	14.612 tx/s	14.503 tx/s	14.492 tx/s	14.472 tx/s
COPOR	no anonymization	116.516 tx/s	116.603 tx/s	114.07 tx/s	112.023 tx/s
	tor	111.21 tx/s	112.352 tx/s	110.381 tx/s	109.953 tx/s
	gossip-node	68.129 tx/s	67.972 tx/s	65.903 tx/s	66.158 tx/s
	dandelion	66.923 tx/s	66.324 tx/s	65.324 tx/s	64.778 tx/s

Table 9.3: Table with results for 5 nodes

		pickle	msgpack	bson	json
Algorand	no anonymization	18.323 tx/s	18.423 tx/s	18.413 tx/s	18.418 tx/s
LaKSA	no anonymization	13.256 tx/s	13.855 tx/s	13.367 tx/s	12.211 tx/s
	tor	7.646 tx/s	8.197 tx/s	9.059 tx/s	8.98 tx/s
	gossip-node	7.519 tx/s	9.362 tx/s	9.748 tx/s	8.424 tx/s
	dandelion	9.533 tx/s	8.276 tx/s	8.459 tx/s	10.037 tx/s
COPOR	no anonymization	65.513 tx/s	65.723 tx/s	65.636 tx/s	64.823 tx/s
	tor	51.431 tx/s	42.431 tx/s	45.637 tx/s	31.574 tx/s
	gossip-node	25.633 tx/s	26.162 tx/s	25.002 tx/s	23.11 tx/s
	dandelion	25.486 tx/s	27.034 tx/s	26.955 tx/s	26.318 tx/s

Table 9.4: Table with results for 5 nodes where 2 of them are offline

Specifications for tests with **10 nodes** (Table 9.5, Table 9.6, Table 9.7) are:

- 1000 transactions, block size of 25 transactions, $n_routes = 2$, $m_nodes = 3$
- **Copor** - $tau = 0.75$
- **Laksa** - $q = 3$, $l = 1$, $delta_1 = 0.75$, $delta_2 = 1$
- **Algorand** - $expected_leader = 26$, $expected_filter = 100$, $expected_cert = 200$, $treshold_filter = 6$, $treshold_cert = 12$, $lambda = 0.65$

		pickle	msgpack	bson	json
Algorand	no anonymization	18.093 tx/s	18.113 tx/s	18.084 tx/s	18.04 tx/s
LaKSA	no anonymization	5.232 tx/s	5.407 tx/s	6.018 tx/s	5.211 tx/s
	tor	5.143 tx/s	5.312 tx/s	5.923 tx/s	5.989 tx/s
	gossip-node	5.076 tx/s	5.178 tx/s	5.102 tx/s	4.992 tx/s
	dandelion	5.164 tx/s	5.265 tx/s	5.174 tx/s	5.013 tx/s
COPOR	no anonymization	68.033 tx/s	64.126 tx/s	75.346 tx/s	78.188 tx/s
	tor	69.011 tx/s	62.446 tx/s	55.437 tx/s	44.493 tx/s
	gossip-node	60.882 tx/s	59.672 tx/s	57.703 tx/s	46.238 tx/s
	dandelion	61.913 tx/s	60.021 tx/s	58.904 tx/s	60.312 tx/s

Table 9.5: Table with results for 10 nodes

		pickle	msgpack	bson	json
Algorand	no anonymization	16.983 tx/s	17.003 tx/s	18.173 tx/s	17.112 tx/s
LaKSA	no anonymization	4.032 tx/s	3.901 tx/s	3.976 tx/s	3.932 tx/s
	tor	3.93 tx/s	3.891 tx/s	3.913 tx/s	3.781 tx/s
	gossip-node	3.979 tx/s	4.017 tx/s	3.922 tx/s	3.892 tx/s
	dandelion	3.894 tx/s	3.885 tx/s	3.974 tx/s	3.895 tx/s
COPOR	no anonymization	50.143 tx/s	51.218 tx/s	50.437 tx/s	47.268 tx/s
	tor	32.341 tx/s	37.287 tx/s	35.408 tx/s	34.974 tx/s
	gossip-node	30.412 tx/s	33.03 tx/s	31.204 tx/s	29.176 tx/s
	dandelion	31.03 tx/s	31.419 tx/s	30.871 tx/s	30.342 tx/s

Table 9.6: Table with results for 10 nodes where 3 of them are offline

		pickle	msgpack	bson	json
Algorand	no anonymization	18.167 tx/s	18.623 tx/s	18.703 tx/s	18.660 tx/s
LaKSA	no anonymization	4.542 tx/s	5.59 tx/s	6.021 tx/s	8.109 tx/s
	tor	3.93 tx/s	3.391 tx/s	3.421 tx/s	3.375 tx/s
	gossip-node	3.383 tx/s	3.389 tx/s	2.917 tx/s	2.895 tx/s
	dandelion	3.011 tx/s	2.971 tx/s	3.712 tx/s	4.646 tx/s
COPOR	no anonymization	23.934 tx/s	25.353 tx/s	23.197 tx/s	22.287 tx/s
	tor	4.874 tx/s	5.03 tx/s	4.928 tx/s	3.973 tx/s
	gossip-node	5.312 tx/s	6.107 tx/s	4.726 tx/s	4.937 tx/s
	dandelion	6.733 tx/s	6.774 tx/s	6.217 tx/s	6.016 tx/s

Table 9.7: Table with results for 10 nodes where 5 of them are offline

The third set of tests shows mixed results (Table 9.5, Table 9.6, Table 9.7). Again, the usage of anonymization layer has no inflence on throughput of LaKSA protocol. However, we can finaly see *gossip-node* and *dandelion* be on pair with *tor* anonymization, and even sometimes outperforming it. Algorand has displayed in all of the locally performed tests its ability to produce and commit new blocks even when large portion of nodes is offline.

9.3 Metacentrum tests

To find out how our implementation handles more present nodes in the protocol, we tested it on cloud machine consisting of 18 virtual CPUs in Muni metacentrum. We further decided to run tests only with *pickle* and *msgpack* serialization types, as these provided the smallest final message size and therefore were the most suitable to use in case of real network deployment. Tests were done using **10 nodes** (Table 9.8, Table 9.9, Table 9.10) and their specifications are:

- 1000 transactions, block size of 25 transactions, $n_routes = 3$, $m_nodes = 3$
- **Copor** - $tau = 3.75$
- **Laksa** - $q = 3$, $l = 1$, $delta_1 = 1.5$, $delta_2 = 2.75$
- **Algorand** - $expected_leader = 26$, $expected_filter = 100$, $expected_cert = 200$, $treshold_filter = 6$, $treshold_cert = 12$, $lambda = 1.5$

Tests of Algorand implemention sadly did not produce any meaningful results and so are not included in the resulting tables.

One thing to notice is that the throughput of LaKSA was not much influenced by the number of offline nodes. This can be due to cryptographic sampling, which select leader and voters, selecting the nodes that are online most of the time. COPOR shows decrease in performance with each increase of the number of offline nodes. However, when we tried to run COPOR with lower *tau* delay, messages containing blocks often did not reached some nodes, which resulted in even worse throughput.

		pickle	msgpack
LaKSA	no anonymization	6.802 tx/s	6.121 tx/s
	tor	5.163 tx/s	5.152 tx/s
	gossip-node	5.236 tx/s	5.517 tx/s
	dandelion	5.271 tx/s	5.618 tx/s
COPOR	no anonymization	74.872 tx/s	86.172 tx/s
	tor	57.095 tx/s	62.15 tx/s
	gossip-node	47.937 tx/s	52.595 tx/s
	dandelion	60.064 tx/s	56.392 tx/s

Table 9.8: Table with results for 16 nodes

		pickle	msgpack
LaKSA	no anonymization	5.804 tx/s	5.982 tx/s
	tor	5.172 tx/s	5.164 tx/s
	gossip-node	5.264 tx/s	5.347 tx/s
	dandelion	5.324 tx/s	5.611 tx/s
COPOR	no anonymization	55.347 tx/s	56.825 tx/s
	tor	36.176 tx/s	35.927 tx/s
	gossip-node	37.802 tx/s	37.612 tx/s
	dandelion	38.015 tx/s	38.212 tx/s

Table 9.9: Table with results for 16 nodes where 2 of them are offline

		pickle	msgpack
LaKSA	no anonymization	4.829 tx/s	4.922 tx/s
	tor	4.172 tx/s	4.175 tx/s
	gossip-node	4.204 tx/s	4.173 tx/s
	dandelion	4.342 tx/s	4.372 tx/s
COPOR	no anonymization	37.209 tx/s	38.011 tx/s
	tor	23.163 tx/s	23.953 tx/s
	gossip-node	23.724 tx/s	23.065 tx/s
	dandelion	24.511 tx/s	24.236 tx/s

Table 9.10: Table with results for 16 nodes where 5 of them are offline

9.4 Discussion

At first, we need to give strong emphasis on the fact that many of the tests we inteded to run have failed, and only results from those that executed to the end and produced files with statistics were presented. The main reason for their failure is the congestion of sockets belonging to the individual nodes, which prevented them from obtaining blocks within the boundary of the round they were created in, and sometimes not recieving them at all. This problem could be rectified by creating a mechanism for periodic resending of the messages containing blocks, votes, etc. Further increase of delay timeouts would only be meaningful with the proposed mechanism in place.

The performed tests proven that the implemented additional anonymization layer introduces moderate delays that are negligible for some testing scenarios. Further testing is needed to obtain results that more closely match real-world conditions. Such testing needs to be performed on multiple machines where each machine will represent only one node and messages with blocks, votes, etc., will be sent via real physical component in the network (routers, switches) instead of local interface (which was the subject of tests).

Looking back at our implementation we will analyse its risks and intristic problems. For the anonymization layer, it is necessary to periodically rotate the circuits, due to the possibility of tracing the patters of anonymous messages sent in the network by the adversary, given a sufficiently large amount of captured communication. In our implementation, the only countermeasure implemented so far that adresses this issue is to always send message over a randomly selected circuit from all the established circuits. However, if the number of established circuits is too low, the added security of this measure decreases.

Another point of interest is to study how nodes are encouraged to forward anonymous messages over the network. The content of such messages is only revealed during final decryption by the end node, and the other nodes do not know whether they are forwarding a message that really belongs to the protocol (a message containing a block, voice, etc.) or whether it is some completely unrelated message. Such messages will be discarded by end nodes and pointlessly wasted network bandwidth. Also, nodes are not rewarded in any way for forwarding anonymous messages (the only incentive is that they want to commit a new block), which may lead to a decision not to forward such messages and halt protocol progression.

Some primitives were abstracted in our implementation, such as construction of round beacon in LaKSA, other were not implemented at all, like mechanism that prevent creation of forks. Their implementation and testing is another possible direction for future work. It would also be interesting to see how the use of different transmission protocols (UDP, TCP and others) affects throughput, network load and message dropping.

Chapter 10

Conclusion

In this thesis, Proof of Stake protocols Algorand, LaKSA and COPOR were studied and their underlying building blocks were thoroughly described. They were compared with with each other in terms of blockchain crucial properties - throughput, scalability, liveness, finality and safety - with additional look at their fairness and DoS resistace. Possibilities of hiding the messsage originator in the network were also studied and described, mainly onion routing and mixing networks. In addition to these techniques, technique for unpredictable message spreading proposed for Bitcoin was presented for its lightweightness. Missing real onion routing in COPOR was implemented and added to the source code as a separate `anonymity_layer` module. This makes it reusable for future consensus protocol implementations. LaKSA protocol was implemented together with Algorand, to compare them against each other, as LaKSA was created to adress issues in Algorand. All of the implemented protocols underwent a series of tests in order to evaulate their performance and impact of additional anonymity layer on transaction throughput. Results were discussed together with issues of the implementation.

The moderate and sometimes negligible impact of the anonymization layer on throughput that was demonstrated during the tests indicates that the research effort was put in the right direction. Comparison of our proof of concept results shows promissing higher throughput of COPOR protocol. However, implementation in other more optimalized language is encouraged, together with usage of some established protocol for messages like Google's protobuf, to further reduce message size.

The enhanced implementation of COPOR can be extended with other algorithms for leader election, like homomorphic sortition, that could remove the need for anonymity layer. Other existing protocols can also be implemented, enhanced by anonymity layer and evaulate on how they perform with it. Another possible direction of future research could be to search for more reliable transmission protocol to replace the used UDP protocol, e.g. QUIC (which is not yet implemented in the used python language). Mechanism that prevent responsible for finality and safety (i.e. prevent forks) in COPOR are not currently implemented and it would be interesting to study different existing techniques used to test these mechanisms and try them out in some future work.

Bibliography

[1] Algorand foundation. *Algorand Blockchain Features Specification Version 1.0* [online]. [cit. 2024-05-09]. Available at: https://github.com/algorandfoundation/specs/blob/master/overview/Algorand_v1_spec-2.pdf.

[2] Anonymous author. *A Guide To The Algorand Consensus Protocol* [online]. [cit. 2024-05-09]. Available at: https://github.com/onplanetnowhere/AlgorandConsensusProtocolMD.

[3] Belotti, M., Božić, N., Pujolle, G. and Secci, S. A Vademecum on Blockchain Technologies: When, Which, and How. *IEEE Communications Surveys & Tutorials*. 2019, vol. 21, no. 4, p. 3796–3838. DOI: 10.1109/COMST.2019.2928178.

[4] Bitcoin Wiki. *BIP 0156* [online]. September 2017 [cit. 2024-05-12]. Available at: https://en.bitcoin.it/wiki/BIP_0156.

[5] Bojja Venkatakrishnan, S., Fanti, G. and Viswanath, P. Dandelion: Redesigning the Bitcoin Network for Anonymity. *Proc. ACM Meas. Anal. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. june 2017, vol. 1, no. 1. DOI: 10.1145/3084459. Available at: https://doi.org/10.1145/3084459.

[6] Buterin, V. and Griffith, V. *Casper the Friendly Finality Gadget*. 2019.

[7] Chase, B. and MacBrough, E. Analysis of the XRP Ledger Consensus Protocol. *ArXiv*. 2018, abs/1802.07242. Available at: https://api.semanticscholar.org/CorpusID:3440860.

[8] Chen, P.-W., Jiang, B.-S. and Wang, C.-H. Blockchain-based payment collection supervision system using pervasive Bitcoin digital wallet. In: *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2017, p. 139–146. DOI: 10.1109/WiMOB.2017.8115844.

[9] Daian, P., Pass, R. and Shi, E. Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake. In: Goldberg, I. and Moore, T., ed. *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2019, p. 23–41. ISBN 978-3-030-32101-7.

[10] Dingledine, R., Mathewson, N. and Syverson, P. Tor: The Second-Generation Onion Router. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, August 2004. Available at: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

[11] Fanti, G., Venkatakrishnan, S. B., Bakshi, S., Denby, B., Bhargava, S. et al. Dandelion++: Lightweight Cryptocurrency Networking with Formal Anonymity Guarantees. *Proc. ACM Meas. Anal. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. june 2018, vol. 2, no. 2. DOI: 10.1145/3224424. Available at: https://doi.org/10.1145/3224424.

[12] Feigenbaum, J., Johnson, A. and Syverson, P. A Model of Onion Routing with Provable Anonymity. In: *Proceedings of the 11th International Conference on Financial Cryptography and 1st International Conference on Usable Security*. Berlin, Heidelberg: Springer-Verlag, 2007, p. 57–71. FC'07/USEC'07. ISBN 3540773657.

[13] Gilad, Y., Hemo, R., Micali, S., Vlachos, G. and Zeldovich, N. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, October 2017, p. 51–68. SOSP '17. DOI: 10.1145/3132747.3132757. ISBN 9781450350853. Available at: https://doi.org/10.1145/3132747.3132757.

[14] Grin foundation. *Dandelion++ in Grin: Privacy-Preserving Transaction Aggregation and Propagation* [online]. [cit. 2024-05-09]. Available at: https://docs.grin.mw/wiki/miscellaneous/dandelion/.

[15] Homoliak, I., Venugopalan, S., Reijsbergen, D., Hum, Q., Schumi, R. et al. The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses. *IEEE Communications Surveys & Tutorials*. 2021, vol. 23, no. 1, p. 341–390. DOI: 10.1109/COMST.2020.3033665.

[16] Hud, J. *Security and Performance Testbed for Simulation of Proof-of-Stake Protocols*. Brno, CZ, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/145471.

[17] King, S. and Nadal, S. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. august 2012. Available at: https://www.peercoin.net/read/papers/peercoin-paper.pdf.

[18] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. may 1998, vol. 16, no. 2, p. 133–169. DOI: 10.1145/279227.279229. ISSN 0734-2071. Available at: https://doi.org/10.1145/279227.279229.

[19] Lamport, L. Paxos Made Simple. In:. 2001. Available at: https://api.semanticscholar.org/CorpusID:1936192.

[20] Lamport, L., Shostak, R. and Pease, M. The Byzantine generals problem. In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, p. 203–226. ISBN 9781450372701. Available at: https://doi.org/10.1145/3335772.3335936.

[21] Lepore, C., Ceria, M., Visconti, A., Rao, U. P., Shah, K. A. et al. A Survey on Blockchain Consensus with a Performance Comparison of PoW, PoS and Pure PoS.

Mathematics. 2020, vol. 8, no. 10. DOI: 10.3390/math8101782. ISSN 2227-7390. Available at: https://www.mdpi.com/2227-7390/8/10/1782.

[22] Malkhi, D. *BFT on a DAG* [online]. 2022 [cit. 2024-09-05]. Available at: https://blog.chain.link/bft-on-a-dag/.

[23] Mendonca, M., Seetharaman, S. and Obraczka, K. A flexible in-network IP anonymization service. In: *2012 IEEE International Conference on Communications (ICC)*. June 2012, p. 6651–6656. DOI: 10.1109/ICC.2012.6364931. ISBN 978-1-4577-2052-9.

[24] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. march 2009. Available at: https://bitcoin.org/bitcoin.pdf.

[25] Ongaro, D. and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, p. 305–319. ISBN 978-1-931971-10-2. Available at: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[26] P4Titan. Slimcoin: A Peer-to-Peer Crypto-Currency with Proof-of-Burn. may 2014. Available at: https://github.com/slimcoin-project/slimcoin-project.github.io/blob/master/whitepaperSLM.pdf.

[27] Prisco, R. D., Lampson, B. and Lynch, N. Revisiting the paxos algorithm. *Theoretical Computer Science*. 2000, vol. 243, no. 1, p. 35–91. DOI: https://doi.org/10.1016/S0304-3975(00)00042-6. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/S0304397500000426.

[28] Reijsbergen, D., Szalachowski, P., Ke, J., Li, Z. and Zhou, J. LaKSA: A Probabilistic Proof-of-Stake Protocol. 2021. Available at: https://arxiv.org/abs/2006.01427.

[29] Ren, J. and Wu, J. Survey on anonymous communications in computer networks. *Computer Communications*. 2010, vol. 33, no. 4, p. 420–431. DOI: https://doi.org/10.1016/j.comcom.2009.11.009. ISSN 0140-3664. Available at: https://www.sciencedirect.com/science/article/pii/S0140366409002989.

[30] Ribarski, P. and Antovski, L. Mixnets: Implementation and performance evaluation of decryption and re-encryption types. In: *Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces*. 2012, p. 493–498. DOI: 10.2498/iti.2012.0432.

[31] Sampigethaya, K. and Poovendran, R. A Survey on Mix Networks and Their Secure Applications. *Proceedings of the IEEE*. 2006, vol. 94, no. 12, p. 2142–2181. DOI: 10.1109/JPROC.2006.889687.

[32] Schollmeier, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: *Proceedings First International Conference on Peer-to-Peer Computing*. 2001, p. 101–102. DOI: 10.1109/P2P.2001.990434.

[33] Singh, A., Kumar, G., Saha, R., Conti, M., Alazab, M. et al. A survey and taxonomy of consensus protocols for blockchains. *Journal of Systems Architecture*. 2022, vol. 127, p. 102503. DOI: https://doi.org/10.1016/j.sysarc.2022.102503. ISSN 1383-7621. Available at: https://www.sciencedirect.com/science/article/pii/S1383762122000777.

[34] Syverson, P. and contributors collective of Onion routing. *Onion Routing* [online]. 2013 [cit. 2023-12-27]. Available at: https://www.onion-router.net/.

[35] Tamaškovič, M. *Fast, Scalable and DoS-Resistant Proof-of-Stake Consensus Protocol Based on an Anonymization Layer*. Brno, CZ, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/136727.

[36] Wang, W., Hoang, D. T., Hu, P., Xiong, Z., Niyato, D. et al. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. *IEEE Access*. 2019, vol. 7, p. 22328–22370. DOI: 10.1109/ACCESS.2019.2896108.

[37] Xiao, Y., Zhang, N., Lou, W. and Hou, Y. T. A Survey of Distributed Consensus Protocols for Blockchain Networks. *IEEE Communications Surveys & Tutorials*. 2020, vol. 22, no. 2, p. 1432–1465. DOI: 10.1109/COMST.2020.2969706.

[38] Xie, M., Liu, J., Chen, S. and Lin, M. A survey on blockchain consensus mechanism: research overview, current advances and future directions. *International Journal of Intelligent Computing and Cybernetics*. january 2023, vol. 16, p. 314–340. DOI: 10.1108/IJICC-05-2022-0126.

[39] Zhang, S. and Lee, J.-H. Analysis of the main consensus protocols of blockchain. *ICT Express*. 2020, vol. 6, no. 2, p. 93–97. DOI: https://doi.org/10.1016/j.icte.2019.08.001. ISSN 2405-9595. Available at: https://www.sciencedirect.com/science/article/pii/S240595951930164X.

Appendix A

Included CD contents

In the attached CD there is located source code for the implementation, this source code and its compiled pdf version. Structure of directories:

- `src` - contain source code of the implementation written in python3.8
  - `.vscode` - configuration files for tasks and debugging inside VScode editor.
  - `chain` - cointains mudules `copor`, `laksa` and `algorand` with primitives specific to them. Shared primitives are in module `shared`.
  - `docs` - tutorials for usage of terminal user interface (implemented only for COPOR protocol)
  - etc.
- `thesis` - contains source code for this thesis written in latex
- `thesis.pdf` - this thesis in pdf format