
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# DESIGN AN EXPERIMENTAL POS DAG-BASED BLOCKCHAIN CONSENSUAL PROTOCOL

**NÁVRH EXPERIMENTÁLNÍHO POS BLOCKCHAIN KONSENSUÁLNÍHO PROTOKOLU ZALOŽENÉHO NA DAG STRUKTUŘE**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **Bc. TOMÁŠ HLADKÝ**
**AUTOR PRÁCE**

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



**BRNO 2025**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Master's Thesis Assignment

164893

| | |
|---|---|
| Institut: | Department of Intelligent Systems (DITS) |
| Student: | **Hladký Tomáš, Bc.** |
| Programme: | Information Technology and Artificial Intelligence |
| Specialization: | Cybersecurity |
| Title: | **Design an Experimental PoS DAG-based Blockchain Consensual Protocol** |
| Category: | Security |
| Academic year: | 2024/25 |

Assignment:

1. Study principles of blockchains and blockchain consensual protocols, especially Proof-of-Work (PoW), Proof-of-Stake (PoS), and DAG-oriented consensual protocols.
2. Compare existing DAG-based protocols with their PoW and PoS counterparts. Highlight their limitations and strengths.
3. Analyze and evaluate leader selection mechanisms in PoS blockchains. Compare existing solutions implemented in well-known PoS blockchain protocols such as Ethereum, Polkadot, and Algorand.
4. Select at least one leader selection technique and design a novel DAG-based experimental PoS blockchain protocol.
5. Implement the designed protocol and develop experiments for simulating and testing its performance and functionality against other well-established PoS protocols.
6. Create a simulation and testing environment for the protocol. Simulate its operation under both standard behavior and scenarios involving potential attackers.
7. Evaluate experimental results, identify limitations, and discuss security guarantees of the protocol, potential improvements, and future extensions.

Literature:

- Simulátor pro ověření vlastností DAG-based consensus protokolů
  https://www.vut.cz/studenti/zav-prace/detail/145161
- Incentive Attacks on DAG-Based Blockchains with Random Transaction Selection
  https://arxiv.org/abs/2305.16757
- Sycomore: a Permissionless Distributed Ledger that self-adapts to Transactions Demand
  https://hal.science/hal-01888265/document
- Single Secret Leader Election
  https://eprint.iacr.org/2020/025
- SASSAFRAS
  https://research.web3.foundation/Polkadot/protocols/block-production/SASSAFRAS
- Whisk: A practical shuffle-based SSLE protocol for Ethereum
  https://ethresear.ch/t/whisk-a-practical-shuffle-based-ssle-protocol-for-ethereum/11763/1

Requirements for the semestral defence:
1-4.

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

| | |
|---|---|
| Supervisor: | **Perešíni Martin, Ing.** |
| Head of Department: | Kočí Radek, Ing., Ph.D. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 21.5.2025 |
| Approval date: | 9.1.2025 |

# Abstract

Proof of Stake (PoS) blockchain systems face a fundamental tension between security and fairness. Denial of Service (DoS) attacks became critical for blockchains that enable public leader selection. We analyze existing techniques and identify their limitations in major PoS protocols (Ethereum, Polkadot, Algorand). As a response, we present a PoS consensus protocol built on a Directed Acyclic Graph (DAG) structure, incorporating a Single Secret Leader Election (SSLE) mechanism. The proposed protocol uses a commitment scheme for SSLE that leverages zero-knowledge proofs ZKPs, Merkle proofs, and Edwards-curve Digital Signature Algorithm (EdDSA) to guarantee unique, fair, and unpredictable leader election. To achieve more realistic results, we collected latency data from 246 location-specific servers and extended an existing peer-to-peer network topology generator to generate networks from the created dataset. We then developed a flexible DAG structure that adjusts the number of parallel branches based on the transaction load, enabling higher throughput under high demand. The containerized node client implementation and its supporting services provide a modular architecture that simplifies both setup and experiment execution.

# Abstrakt

Proof of Stake (PoS) blockchainové systémy čelia fundamentálnemu napätiu medzi bezpečnosťou a férovosťou. Utoky typu Denial of Service (DoS) sa stali kritickými pre blockchainy, ktoré umožňujú verejný výber vodcu. Analyzujeme existujúce techniky a identifikujeme ich obmedzenia v hlavných PoS protokoloch (Ethereum, Polkadot, Algorand). Ako reakciu predstavujeme konsenzuálny PoS protokol postavený na štruktúre acyklicky orientovaného grafu (DAG), ktorý zahŕňa mechanizmus výberu jediného tajného vodcu (SSLE). Navrhovaný protokol využíva záväzkovú schému pre SSLE, ktorá využíva zero-knowledge dôkazy (ZKP), Merkle dôkazy a digitálny podpisový algoritmus na eliptických krivkách (Edwards-curve Digital Signature Algorithm, EdDSA), aby zabezpečil jedinečný, férový a nepredvídateľný výber vodcu. Na dosiahnutie realistickejších výsledkov sme nazbierali údaje o latencii zo 246 serverov na svete a rozšírili sme existujúci generátor topológie peer-to-peer sietí tak, aby generoval siete na základe vytvoreného datasetu. Následne sme vyvinuli flexibilnú DAG štruktúru, ktorá prispôsobuje počet paralelných vetiev podľa zaťaženia transakciami, čo umožňuje vyššiu priepustnosť pri vysokom dopyte. Kontajnerizovaná implementácia blockchain klienta s podpornými službami dosahuje modulárnejšiu architektúru, ktorá umožňuje jednoduchšie nastavenie a realizáciu experimentov.



# Keywords

blockchain, DAG-based Consensus Protocol, Zero-knowledge proof, Proof-of-Stake, Single Secret Leader Election, Docker, peer-to-peer

# Klíčová slova

blockchain, konsenzuálny protokol založený na štruktúre DAG, Zero-knowledge proof, Proof-of-Stake, voľba jedného tajného vodcu, Docker, peer-to-peer


# Reference

HLADKÝ, Tomáš. *Design an Experimental PoS DAG-based Blockchain Consensual Protocol.* Master's thesis. Supervisor Ing. Martin Perešíni, . Brno: Brno University of Technology, Faculty of Information Technology, 2025.

# Rozšířený abstrakt

Technológia blockchainu revolučným spôsobom zmenila spôsob, akým uvažujeme o decentralizovaných systémoch, keďže poskytuje dôveryhodné, transparentné a nemenné záznamy. Blockchainy sa od svojho počiatku, kedy využívali tradičné konsenzuálne mechanizmy ako Proof of Work (PoW), výrazne vyvinuli. Najmä protokoly typu Proof of Stake (PoS) sľubujú nižšiu spotrebu energie a lepšiu škálovateľnosť, no zároveň priniesli výzvy v oblasti férovosti a ochrany pred útokmi typu Denial of Service (DoS). Mnohé PoS blockchainy sa spoliehajú na verejný výber lídrov alebo umožňujú výber viacerých vodcov (prípadne žiadneho), pričom výber zostáva utajený.

Táto práca sa zameriava na výber jediného tajného vodcu (SSLE), ktorého cieľom je v každom kole potajme zvoliť iba jedného vodcu. Tento prístup je predmetom výskumu v rámci projektu Ethereum, ktorý slúžil ako základ pre náš návrh.

Štruktúra acyklicky orientovaného grafu (DAG) sa zároveň objavila ako alternatíva k jednoreťazcovým blockchainom, ktoré trpia nízkou priepustnosťou pri vysokom počte transakcií. V našej práci sme predstavili štruktúru (inšpirovanú existujúcou štruktúrou Sycomore), ktorá tento problém rieši triedením transakcií do jednotlivých vetiev na základe ich hash hodnôt.

Navrhli sme záväzkovú schému pre SSLE konsenzus s využitím blockchainovej štruktúry založenej na DAG, schopnej spracovávať viacero reťazcov paralelne pri realistickej peer-to-peer komunikácii so simulovanými latenciami. Táto práca je súčasťou prebiehajúceho výskumu na Security@FIT, Fakulte informačných technológií Vysokého učení technického v Brne. Medzi jej príspevky patria:

- Navrhli a implementovali sme záväzkovú schému v oblasti výberu jediného tajného vodcu pomocou zero-knowledge dôkazov (ZKP), Merkle dôkazov a digitálneho podpisového algoritmu na eliptických krivkách (Edwards-curve Digital Signature Algorithm, EdDSA).
- Zo serverov na 246 geografických lokalitách sme zozbierali údaje a vytvorili čistý dataset obsahujúci informácie o polohe každého servera a odozvách (ping) medzi všetkými 245 ostatnými lokalitami.
- Rozšírili sme existujúci generátor topológií peer-to-peer sietí o podporu pre vytvorený dataset, upravili jeho výstupy a pridali podporu pre menšie siete.
- Vyvinuli sme DAG štruktúru pre blockchain, ktorá dokáže zvyšovať a znižovať počet paralelných vetiev podľa aktuálneho dopytu po transakciach.
- Implementovali sme klienta blockchainového uzla, ktorý využíva náš konsenzuálny mechanizmus a DAG-založenú blockchainovú štruktúru s doplnkovými službami (napr. generátor topológie siete, služba na inicializáciu, služba na spracovanie dát, služba na správu uzlov), aby bolo možné spúšťať simulácie v kontajnerizovanom prostredí.
- Otestovali sme náš protokol v peer-to-peer sieťach so 40 a 60 uzlami.

# Design an Experimental PoS DAG-based Blockchain Consensual Protocol

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Martin Perešíni. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tomáš Hladký
May 19, 2025


## Acknowledgements

I would like to thank my supervisor Ing. Perešíni Martin for his valuable guidance and advices. I also acknowledge that this work is also part of ongoing research with the Security@FIT at Brno University of Technology.

# Contents

# Chapter 1

# Introduction

Blockchain technology has revolutionized the way we think about decentralized systems, providing trustless, transparent, and immutable ledgers. Blockchains have evolved since the beginning of a traditional consensus mechanism like Proof of Work (PoW). In particular, Proof of Stake (PoS) protocols promise reduced energy usage and better scalability, but they introduced challenges in fairness and Denial of Service (DoS) protection. Many PoS blockchains rely on public leader election or allowing multiple or no leader selection while keeping them secret.

This work focuses on single secret leader election (SSLE), which aims to secretly select only one leader per round. This approach is in research in Ethereum, which served as a base for our design.

Furthermore, the Directed Acyclic Graph (DAG) structure emerged as an alternative to single-chain blockchains that suffer from transaction bottlenecks during high transaction loads. In our work, we introduced a structure (inspired by Syocomore [5]) that eliminates the problem of transaction by sorting transactions to individual branches based on their hash.

In our work, we designed a commitment scheme for SSLE consensus with the utilization of a DAG-based blockchain structure capable of processing several chains in parallel with realistic peer-to-peer communication of simulated latencies. This work is part of ongoing research with the Security@FIT at Brno University of Technology, Faculty of Information Technology, and its contributions are as follows:

- We designed and implemented a commitment scheme within a single-secret leader election domain using zero-knowledge proofs, Merkle proofs, and Edwards-curve Digital Signature Algorithm (EdDSA).
- We gathered data from 246 geolocated servers and created a clean dataset with each server's location and ping latencies to the other 245 locations.
- We extended an existing peer-to-peer network topology generator to support the created dataset, modified its output, and added support for smaller networks.
- We developed a DAG structure for blockchain, which can increase and decrease the number of parallel branches based on transaction demand.
- We implemented a blockchain node client that uses our consensus scheme and DAG-based blockchain structure with additional services (e.g., network topology generator,

bootstrapping service, data processing service, node control service) to run it as a simulation process in a containerized environment.

- We evaluated our protocol on peer-to-peer networks with 40 and 60 nodes.

The organization of this work is as follows: Chapter 2 describes the technical background of blockchain principles, explains the widely used consensus protocol with the cryptographic primitives used, outlines blockchain vulnerabilities, and provides related work. Finally, it explains the SSLE and the problem it aims to solve in our work. Chapter 3 provides the detailed design of our commitment scheme and specifications about its implementation in the node client. Chapter 4 provides implementation details about individual services and overall architecture. Chapter 5 contains an evaluation to verify the functionality of our protocol and experiment with transaction processing throughput. Finally, Chapter 6 provides a summary of the achieved results in this work, offers a conclusion, and discusses potential future work.

# Chapter 2

# Blockchain theory

This chapter describes the base blockchain principles and their structure, and outlines terms used in blockchain-based systems. It further focuses on Proof-of-Work (PoW) blockchains and their tradeoffs and compares them to Proof-of-Stake (PoS) blockchains. Furthermore, discusses cryptographic primitives used in PoS-based protocols, outlines security risks, and provides related work to mitigate Denial of Service (DoS) attacks known in PoS consensus leader election. Finally, it outlines the DAG-based blockchain structure and describes the requirements of Single Secret Leader Election (SSLE), which is further used in our work.

## 2.1 Lack of trust

The term blockchain, also known as an electronic distributed ledger, was popularized by publishing a proposal: „Bitcoin: A peer-to-peer electronic cash system“ written by an author with the pseudonym Satoshi Nakamoto [55]. The proposal was published in 2009, but the author's identity remains unknown.

Several factors motivated Nakamoto to create such a proposal. One of the most significant problems is decentralized trust. Today's world is highly dependent on the internet, and the use of fiat paper money or government-issued physical cash is declining. The world is oriented toward the utilization of payment systems and digital money. Some countries, such as Norway or Sweden, already use digital money more than physical money. Digital money is convenient to use, but problems arise in cases of trust. However, trust is not a reliable factor in finance [55].

Trust can be defined as a form of social contract. It is an equilibrium where legal enforcement leads people to maintain their promises. The importance of trust is even more significant in areas where law enforcement is not prompt. When transferring value from a physical item to a database, all parties involved must establish an element of trust. This includes database protection from data tampering, double-spending, and unauthorized transfers, which requires a strong foundation of trust [29].

In light of this, a blockchain design emerged that aims to re-establish the lost trust using cryptography. It aims to minimize the factor of trust by utilizing a decentralized consensus rather than relying on a centralized entity, which poses a risk to a single point of failure and malfeasance.

Blockchain or DLT (distributed ledger technology) creates transparent records that are difficult to alter without network consensus. Over time, solutions other than Bitcoin have been created, such as Ethereum [72], Litecoin [42], Polkadot [71], Algorand [15], and IOTA

[62]. However, none offer a highly decentralized, scalable, and secure solution. Each has tradeoffs, and the demand to fulfill the essential requirements is part of active research.

This work mainly focuses on Proof-of-Stake (PoS) systems, such as Ethereum, which directly influence both security and reliability in the network. Trust in these systems is crucial because the consensus mechanism depends on validators deciding in the network's best interest. A trusted environment encourages more nodes to participate in staking and become validators, which enhances network resilience and decentralization.

## 2.2 Distributed ledger technology overview

At a high level, a blockchain is a decentralized, append-only list of transaction records shared within a peer-to-peer network. From a network perspective, access to the blockchain system can be private or public. Participants within this network, who choose to contribute and have the required resources (e.g., sufficient mining power or amount of locked stake) can process received transactions created by users who possess blockchain tokens, verify them, and include them in the ledger. Blockchain technology utilizes well-known cryptographic constructions and mechanisms such as hash functions, asymmetric encryption, and digital signatures. Mixed up together, it forms the base for more complex structures such as Merkle trees, the concept of Proof of Work (PoW), and consensus rules to achieve consistency [73].

## 2.3 Blockchain characteristics

This section defines fundamental characteristics that define blockchain functionality, allowing it to be used for various practical, real-world use cases [30].

### 2.3.1 Decentralisation

The centralized entity is vulnerable to several risks, such as being compromised, becoming unavailable due to being a single point of failure or geopolitical reasons, and lacking transparency and trust. These vulnerabilities can cause censorship and control of the system by a single entity. In the economic sector, one example is central banks such as the Federal Reserve in the United States. On the other hand, a decentralized blockchain system aims to remove the necessity of trust in a single entity. Decisions inside the system are made by a consensus mechanism where participants share their beliefs in the source of truth, which is inherently affected by standard protocol rules [33]. Furthermore, having multiple shared copies in the peer-to-peer network eliminates the single point of failure.

### 2.3.2 Coherence

Coherence or a single source of truth ensures that every network participant sees an identical copy of the blockchain. This is guaranteed by the consensus algorithm. Participants follow the same rules to maintain their copy of the chain and to resolve conflicts (e.g., two competing chains) by applying the strongest chain rule (see Fig. 2.2), replacement of the longest chain rule (see Fig. 2.1). Thereby, securing coherence increases overall trust in the blockchain system.

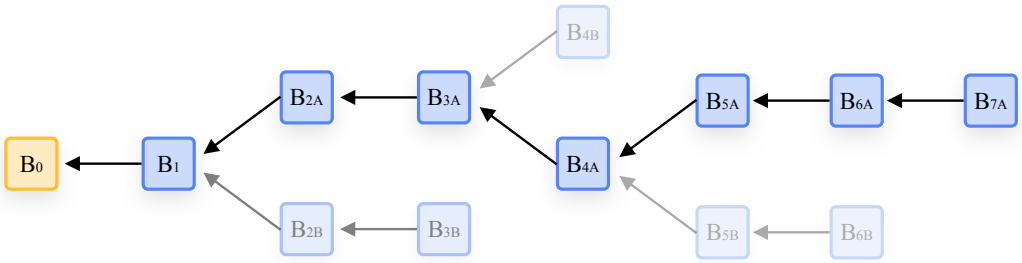



Figure 2.1: Longest chain rule. Leading chain is selected by number of blocks [31].

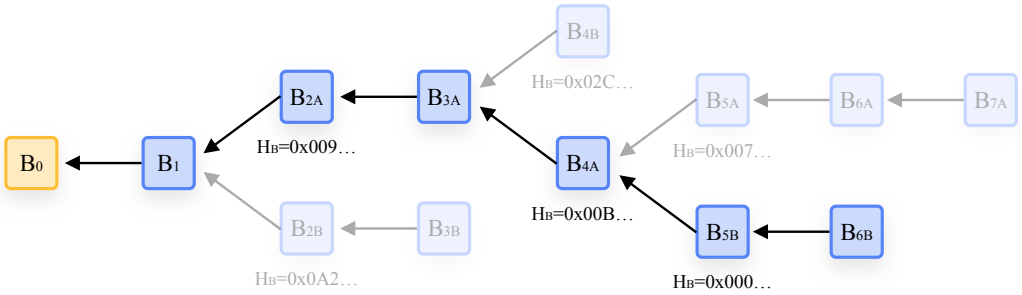



Figure 2.2: Strongest chain rule. Leading chain is selected by highest amount of invested mining power [31].

### 2.3.3 Immutability

Blockchain, by design, does not allow rewriting the content of the ledger, which is achieved by linking blocks together with their hashes. Specifically, each block contains a hash of the previous block, with the expectation of the genesis block, the first block in the ledger that is usually hardcoded to ensure that the network starts with the same initial state. These hashes are created from the block header and its content - transactions. If any transaction is modified, the resulting hash will be different and thus could not match the hash included in the subsequent block (see Fig. 2.3).

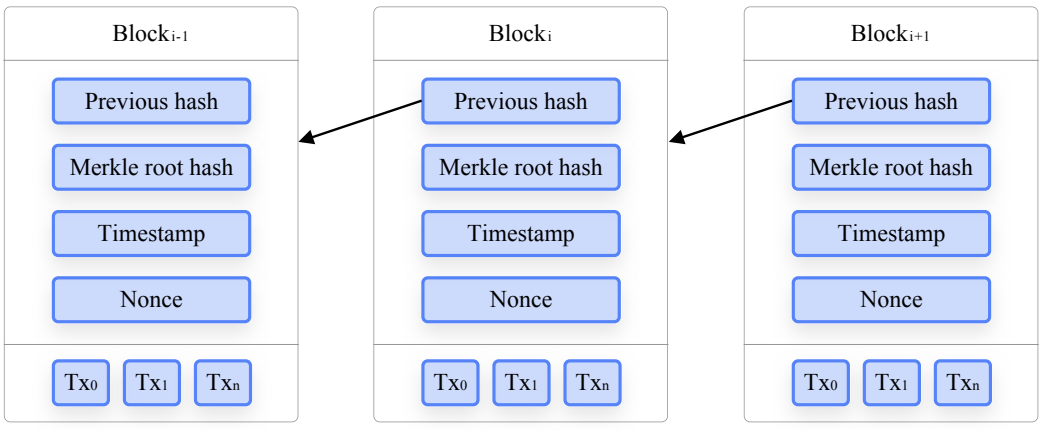



Figure 2.3: Blockchain structure overview [31].

### 2.3.4 Transparency

The blockchain ledger contains a history of all transactions, allowing the balance of every blockchain user to be determined at any point in time, as long as their address is included in some transaction. Permissioned blockchains have restricted access to this information, but permissionless blockchains, such as Bitcoin or Ethereum, are open to everyone. While these ledgers are often considered fully anonymous, this is misleading; they are pseudo-anonymous. This means identity is initially hidden but can be traced and linked with additional information. Identity linking, such as address clustering, is often performed to uncover criminal identities involved in illegal activity, such as money laundering or ransomware payment. It is important to note that there are blockchains that focus on privacy and allow individuals to anonymize their transactions using cryptographic techniques. Examples of such protocols include Monero [59] and ZCash [10].

### 2.3.5 Blockchain trilemma

Current challenges are to make it scalable across large networks and maintain a balance between security and decentralization. This concern is known as the blockchain trilemma. According to the trilemma, a blockchain system can have only two out of three properties: throughput, security, and decentralization (e.g., significantly improving decentralization reduces either transaction throughput or security) (see Fig. 2.4).

The price to reach a fully decentralized system can be high, mainly for two reasons:

- **A consensus mechanism** is a protocol that allows distributed participants to achieve agreement without a central authority. It defines how participants communicate and decide to secure a coherent, immutable, and transparent ledger.
- **Network scalability** limits the number of fully decentralized nodes or several messages in a short time to reach a consensus.

These reasons answer why consensus protocols such as Bitcoin could be faster in transaction processing, as they aim to ensure security, which adds extra overhead to the protocol and makes them highly inefficient. Several academic works try to emphasize this overhead and ask the question, „Do you need a blockchain?“ [73].

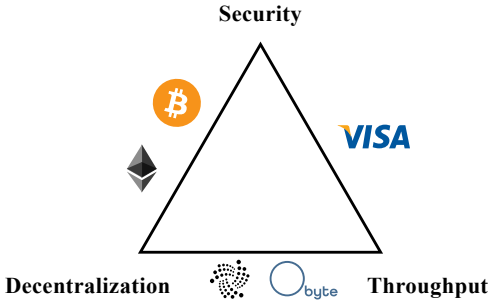



Figure 2.4: Blockchain trilemma showing real-world examples utilizing only two out of three properties. Bitcoin and Ethereum aim to offer high decentralization and security, but lack throughput. On the other hand, Visa offers high transaction throughput and provides good security for its centralized tradeoff. DAG-based protocol, IOTA, and Obyte strive for decentralization and high transaction throughput, which results in security flaws that they aim to balance by involving a centralized element [31].

## 2.4 Cryptographic preliminaries

This section describes the necessary cryptography used later in this work.

### 2.4.1 Cryptographic hash functions

A *cryptographic hash function* is a cryptographic construct that produces a fixed-length output for any input. The cryptographic hash function ($h$) is defined by the following properties:

- **Pre-image resistance.** It is computationally infeasible to find an input $x$ for a given hash value $y$ such that $h(x) = y$.
- **Second pre-image resistance.** It is computationally infeasible to find an input $y$ for given input $x$ such that $h(x) = h(y)$, where $x \neq y$.
- **Collision resistance.** It is computationally infeasible to find two different inputs, $x$ and $y$, such that $h(x) = h(y)$, where $x \neq y$.
- **Avalanche effect.** A slight change (even one bit) in the input will create an entirely different output. This effect contributes to the uniformity of output distribution.

For a sufficient cryptographic hash function, the produced output is assumed to behave as if it is independent and uniformly distributed in $\{0,1\}^n$, where $n$ is the size of the output. In theory, the number of inputs is infinite, and the fixed size of the output limits the total number of outputs. Thus, by definition of the pigeonhole principle, if $n$ inputs are mapped into m outputs and $n > m$, then at least one output contains more than one input mapping (see 2.5). However, the probability of finding such a combination for a sufficient output length (e.g., 256 bits or 512 bits) is with current computation power extremely low. According to the birthday attack [8], the probability of a collision starts to become significant at 50% after around $2^{128}$ hash operations (sample space of $2^{256}$ possibilities), which makes the probability of a theoretical attack computationally infeasible with current technology [51, 18].

For instance, Bitcoin uses SHA-256 (Secure Hash Algorithm 256-bit) to ensure data integrity and security, including operations such as block hashing, transaction identification, and Merkle tree generation. For similar use cases, Keccak-256 is used in Ethereum. Besides that, it is also used in use cases not available in Bitcoin, such as Ethereum's Turing-complete smart contract platform, commitment schemes, and zero-knowledge proofs.

In summary, cryptographic hash functions are widely used in cybersecurity and blockchain technologies.

### 2.4.2 Digital Signature Algorithm

The Digital Signature Algorithm (DSA) is a cryptographic algorithm based on the discrete logarithm problem in prime-order subgroups. It is used to generate digital signatures that provide authentication, data integrity, and non-repudiation. Assume two parties, the sender and receiver. Two keys are associated with the sender: the private key, which is only owned by the sender, and a public key, which is publicly available to the receiver. The algorithm allows the sender to create a signature for the hashed input message using their private key. The signature is then sent to the receiver along with the message data. The receiver can

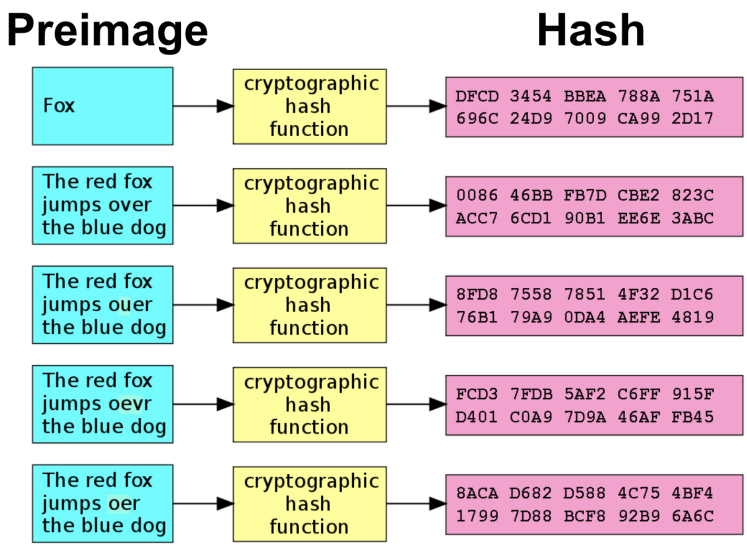



Figure 2.5: An example of the avalanche effect in a cryptographic hash function [50].

verify using the sender's public key that the signature for the message is valid (see Fig. 2.6). This ensures that the message truly originates from the claimed sender (authentication), the signed message was not altered (data integrity), and that the sender created the signature, as they are the only entity capable of creating this signature using their private key (non-repudiation) [56].

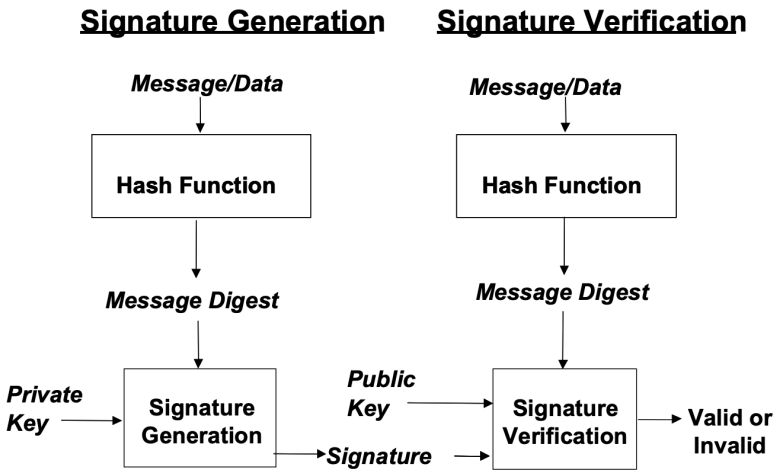



Figure 2.6: Visualization of signature generation and signature verification for DSA [56].

The elliptic-curve variant, Elliptic Curve Digital Signature Algorithm (ECDSA), employs the same signature framework but operates over elliptic curves to achieve equivalent results with smaller keys [34].

EdDSA (Edwards-curve Digital Signature Algorithm) further refines the signature system with Schnorr's scheme on (possibly twisted) Edwards curves, achieving high performance. The EdDSA has been standardized in RFC 8032 [35].

## 2.5 Blockchain structure

In this Section, we describe the blockchain structure. Although the blockchain structures differ across multiple blockchain implementations, we mainly focus on traditional Nakamoto's structure as it provides the base concepts from which other blockchains are inherited. However, we also mention newly adopted components that can be seen with the rise of the Ethereum network.

### 2.5.1 Transaction

A transaction represents a movement of funds from one address (a.k.a., sender) owned by a signer of the transaction to another address (a.k.a., recipient). The size of the movement is specified by the number of tokens that are required to be available in the sender's balance for the transaction to be considered valid.

In Bitcoin, the recipient field can be specified by multiple addresses, thereby specifying one transaction input to multiple outputs. This functionality is tied to Bitcoin's UTXO (unspent transaction output) model, where transaction consumes outputs from previous transactions and create new outputs (see Fig. 2.7). The UTXO model increases the traceability of all fund movements and increases privacy if multiple addresses are utilized [55]. In contrast, Ethereum uses an account balance model, which allows one recipient address per input [72]. This model is more intuitive, and simplicity is more beneficial for fund manipulation using smart contracts within the Ethereum network. Besides that, the account balance is also easier for balance check validation.

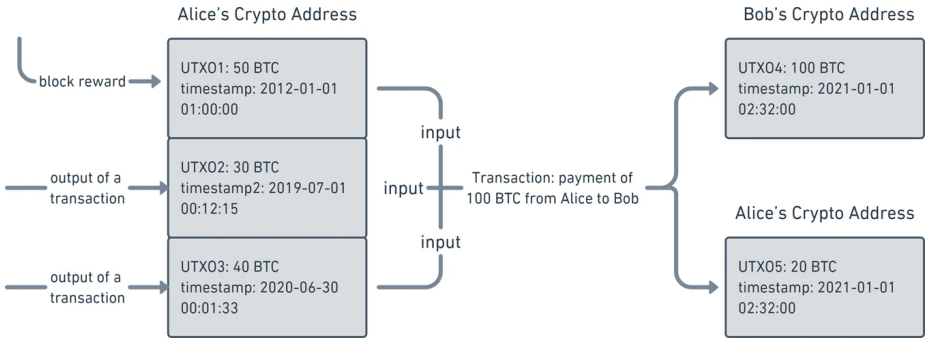



Figure 2.7: Example visualization of UTXO model [46].

The transaction can also contain an optional data field. This is primarily used in blockchains that support smart contracts, such as Ethereum. It may include function calls and their parameters, allowing users to interact with deployed smart contracts.

Once a transaction is created and broadcast into the blockchain network, it will be processed by including it inside a block. Miners (or validators in PoS blockchains) verify transactions that include the sender's signature and a sufficient number of tokens available in the sender's balance for transaction output. After that, an incentive for the miner is required to include this transaction in the next upcoming block. The transaction creator provides the incentive in the form of a fee. The higher the fee, the bigger the incentive for miners to process transactions earlier than others. Blockchains such as Ethereum that involve smart contract function invocation use gas instead of the traditional fee model to reward the computational effort required to process transactions fairly.

### 2.5.2 Merkle tree

Merkle tree, also known as a hash tree, is a cryptographic binary tree of hash values that enables efficient and secure verification of large data sets. In a Merkle tree, the last node of the tree is called the leaf node, and it is created by hashing the data value associated with it. The Merkle root is created by recursively hashing pairs of child nodes up to a single root. All leaves are grouped in pairs, and every pair has a computed hash stored in the parent.

One of the places in blockchain where the Merkle tree is utilized is transactions inside a block. Transactions are at the leaves position and in direction up to the root, producing a snapshot of all transactions in one block (see Fig. 2.8) [52].

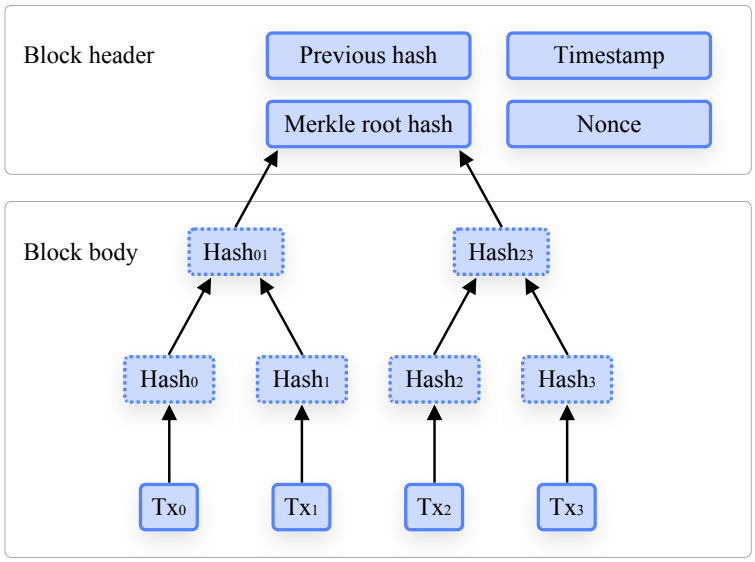



Figure 2.8: Block structure comprises a block header and a block body [31].

### 2.5.3 Merkle proof

Merkle proof allows for verifying any data block's value membership in the tree by providing a Merkle authentication path from the leaf up to the root with only the necessary sibling node required for the hash value of the parent node (see Fig. 2.9). A benefit of Merkle proof is that we only have a logarithmically-sized path instead of nodes of the whole tree to prove the membership [52].

### 2.5.4 Block

A block is a major element in the blockchain containing a set of processed transactions. All of these transactions need to be valid (i.e., the sender has enough spendable funds and the signature of the transaction is valid). Otherwise, the network will deny the block. When a new block in blockchain is created, its content and header fields serve as input for the hash function. The produced hash output is then linked to the following block, which allows for the creation of a binding between blocks to form the chain of blocks called the blockchain (see Fig. 2.8). The creation of the block varies based on the consensus protocol.

In PoW-based blockchains, including Bitcoin, miners are aiming to find a solution to a computational problem (value for nonce), specifically to a hash algorithm, such as SHA-

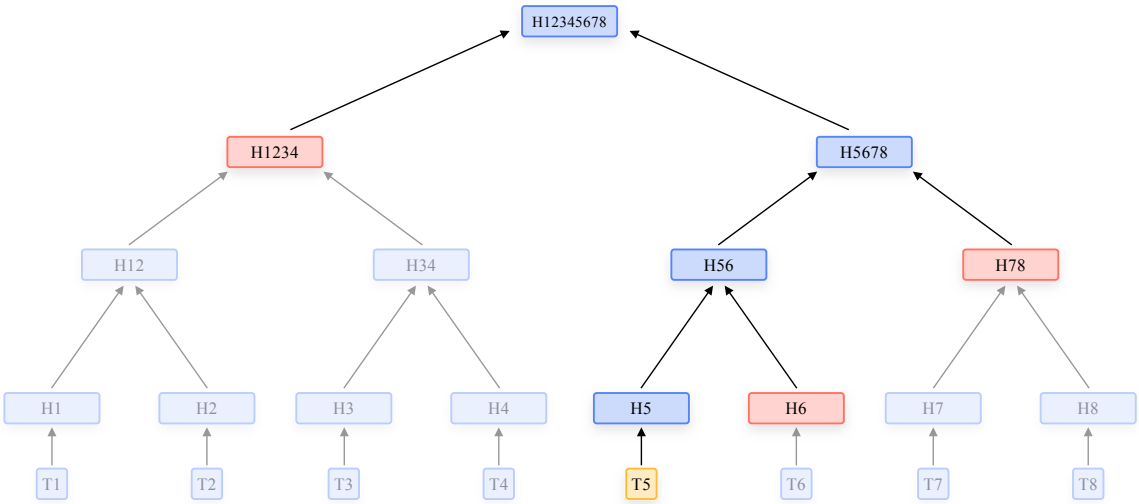



Figure 2.9: Merkle proof visualization. Yellow node is node that we want to prove membership and red nodes are the only required nodes to construct valid Merkle proof [52].

256. The resulting hash value needs to be less than the target value, which is determined by difficulty. This search has no mathematical pattern, and nodes need to try different values randomly. The target value is dynamically adjusted so the average time to create a block stays the same. This ensures that as computational hardware gets better, the time to create will stay the same. The average time to generate a block in this protocol needs to be high enough. Low time value would result in more than one block being generated in a short amount of time, with the same parent block, resulting in many conflicting blocks. The network can accept only one block referring to the same parent. The block that gets propagated across the network the fastest will win the race — it will be included in the blockchain, its author will be rewarded, and other valid blocks will be discarded (i.e., stale blocks). This would result in disincentivization of other participants because their resources would be wasted. By having a time window large enough, significantly fewer stale blocks will be produced. Another benefit of this approach is that it is computationally infeasible to rewrite existing blocks [55, 41].

By contrast, PoS-based protocols replace computational mining with a validator selection process where node lock their assets to participate in block production. Participants then select a leader for every block to generate (either by amount of stake, trust accumulated in other nodes, or randomly). The block production then does not require a nonce because the validator is already selected. The chosen validator constructs a block containing valid transactions, signs it, and broadcasts it. Other participants then verify transactions and the proposer's stake-based signature [27].

### 2.5.5 RANDAO

Generating untampered, unbiased, and transparent randomness in blockchain systems is a complex challenge. At the Ethereum 2.0 upgrade, RANDAO (Randomized Decentralized Autonomous Organization) was introduced as a core part of the randomness generation mechanism. RANDAO is a verifiable, decentralized randomness beacon based on a commitment-reveal scheme. In commitment-reveal, members commit their randomly chosen values to a group, and the final value is the combination of these values, commonly via

an XOR operation. The RANDAO is used for validator shuffling to ensure fair validator selection [67].

The RANDAO has been the subject of several research publications that show how randomness can be biased [67, 4, 54]. For this reason, the Ethereum Foundation is actively working on a solution based on Verifiable Delayed Function (VDF). The idea of this improvement is to delay the calculation of a random value in a way that the output is known long after the input has been committed. Because the output is not visible, it is hard to bias it [75].

## 2.6 Terms

This section introduces the key blockchain terms used throughout our work.

### 2.6.1 Full node

Full nodes are known as the backbone of the blockchain, as they possess complete copies of the whole blockchain. Depending on their activity in the network, they can be split into two categories [33]:

- **A consensus node** is an entity actively participating in the PoW blockchain (a.k.a. miner). They take responsibility for producing blockchain with processed transactions and appending them to the ledger. Besides that, they also validate received blocks and transactions from other miners. It is also important to note that they have the capability to prevent malicious behavior to a certain degree (e.g., by not including invalid transactions in a block or refusing to accept chains with faulty integrity). From a PoS perspective, this includes active participation in elections and block-proposing events. Some PoS protocols also utilize a stake-locking system (i.e., Ethereum PoS, Cardano) to lock a certain amount of stake as a form of commitment to participation as a consensus node. If the node is unable to fulfill its responsibilities or acts maliciously, the locked stake will be removed.

- **A validating node** is an entity that validates as a consensus node but cannot do a write ledger operation. Also, their ability to prevent malicious behavior is limited because they can only detect it and choose not to spread the received information further.

### 2.6.2 Mining pool

In Nakamoto's protocol, the mining difficulty variable is a key factor impacting miners' profitability. In order to keep the average block generation at around 10 minutes for Bitcoin, the difficulty dynamically increases or decreases depending on the time to mine the last 2,016 blocks (i.e., approximately every two weeks) [55].

- The difficulty increases as more miners join the network or upgrade their hardware equipment to have a higher chance of producing blocks faster. This technique, however, leads to new problems: One of them is high energy usage on a global scale. Miners try to find a correct nonce to produce a block, but the purpose of the found nonce is to create competition between miners and have preferably only one clear winner. Energy spent on nonces that do not lead to solutions by miners is wasted. On

the other hand, there are also arguments that the found nonce helps add to the chain strength, so it is harder for an adversary to overwrite the main chain with different chains.

- Another problem is that with increasing difficulty, it is constantly harder for long-lasting miners to produce a block without upgrading their mining hardware. The continuously lower chance of mining a block and the uncertainty of not being rewarded emerged as an opportunity for miners to form a mining pool.

A mining pool is a form of cooperative mining. Instead of finding a proper nonce individually, miners combine computational resources. They share the results of their findings, and when someone finds the solution, they split the reward. Nevertheless, there might also be less common payout schemes for reward splits. Mining pools now contribute to around 99% of the total hash power in Bitcoin, and individual mining has become rare (see Fig. 2.10) [68].

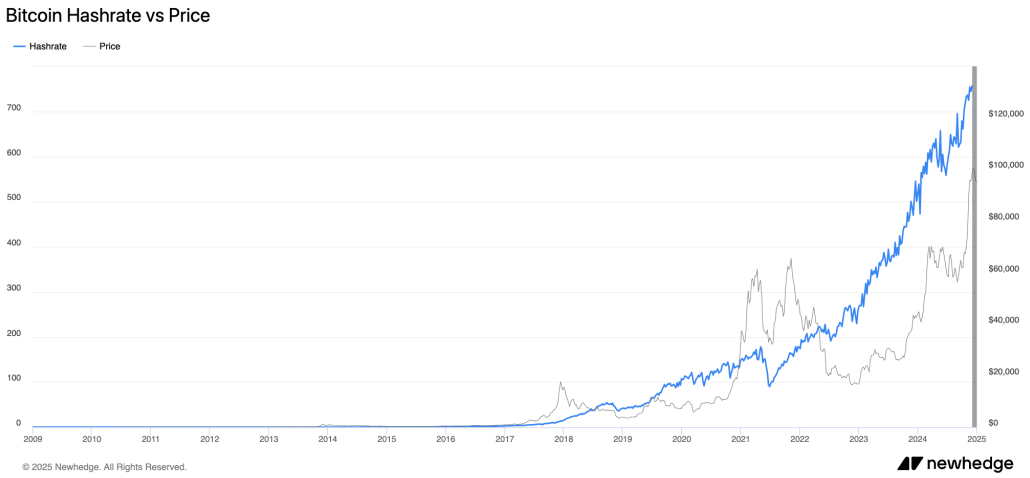



Figure 2.10: Total Bitcoin hashrate (EH/s) in comparison with its price between years 2009 and 2025 [57].

### 2.6.3 Consensus mechanism

A protocol used in distributed systems, such as blockchains, to achieve agreement among network participants on the current blockchain's state, its content and order, and the validity of data structures shared in the blockchain, such as transactions, blocks, and commitments. The protocol ensures a consistent view on the blockchain [40]. Widely used protocols include PoW, PoS, and their modifications.

### 2.6.4 Scalability

Scalability defines how well blockchain scales in terms of the number of active participants in the network. PoW-based blockchains are highly scalable because they have low communication demand. This property is suitable for achieving full decentralization. PoS-based protocols are less scalable because they require high communication demands. For this reason, PoS blockchains are widely used in private blockchains with limited access and a

low number of nodes [33]. To achieve the same decentralization in public blockchain, they deploy techniques to reduce message overhead in the network, which in turn decreases their security or throughput.

### 2.6.5 Throughput

Throughput refers to the rate of transactions that can be processed per given period of time, commonly measured in seconds as transactions per second (TPS). The result depends on the number of blocks processed during this time window and the number of transactions included in individual blocks. In practical terms, for blockchains with a traditional single-chain structure, TPS equals the block production rate multiplied by the average transaction count [33]. For DAG structure with an adaptable number of parallel chains; however, this number can vary greatly based on transaction load.

### 2.6.6 Finality

Finality is the guarantee that the transaction has been irreversibly recorded on the blockchain and cannot be modified or removed. It specifies the number of blocks to append after the block containing the target transaction before the transaction is considered as processed. This number may vary across different sources, as they may require different certainty. The higher the number of blocks to wait, the higher the confidence that the transaction has been processed and is valid [33].

## 2.7 Proof-of-Work

Proof-of-Work (PoW) is the foundational consensus protocol that was initially popularized as Nakamoto's consensus protocol in its real-world implementation known as Bitcoin. Active participants in PoW that maintain the ledger by processing transactions and producing blocks with a significant amount of „work“ known as a lottery. The produced work, as mentioned earlier, includes solving cryptographic puzzles. From a cryptographic perspective, there is a limited number of valid solutions for the nonce value. Once a valid solution is found, any node can verify the work, which is computationally hard to produce but easy to verify [55].

As the oldest running blockchain, Bitcoin has a PoW consensus protocol proving that the security model has withstood over a decade of testing in a real-world environment. Its hybrid decentralized-secure model attracted many new miners, which created a speed race in mining blocks. From an economic perspective, Bitcoin is ranked as the fifth largest asset (when comparing cryptocurrencies, commodities, and equities) in the world by its market cap, following Gold, and companies Microsoft, Apple, and Nvidia. Its current market cap (i.e., around $2.049 T) is larger than silver and almost 10% of the market cap of gold[1]. On top of that, its transparency and independence provide users greater trust and control over their finances.

### 2.7.1 Downside of Proof-of-Work

The success of Bitcoin was not for free. The machines participating in PoW mechanisms consume vast amounts of energy; see Fig. 2.11 for the total annualized Bitcoin footprints.

[1] According to data from: `https://8marketcap.com/` captured on 9th May 2025.

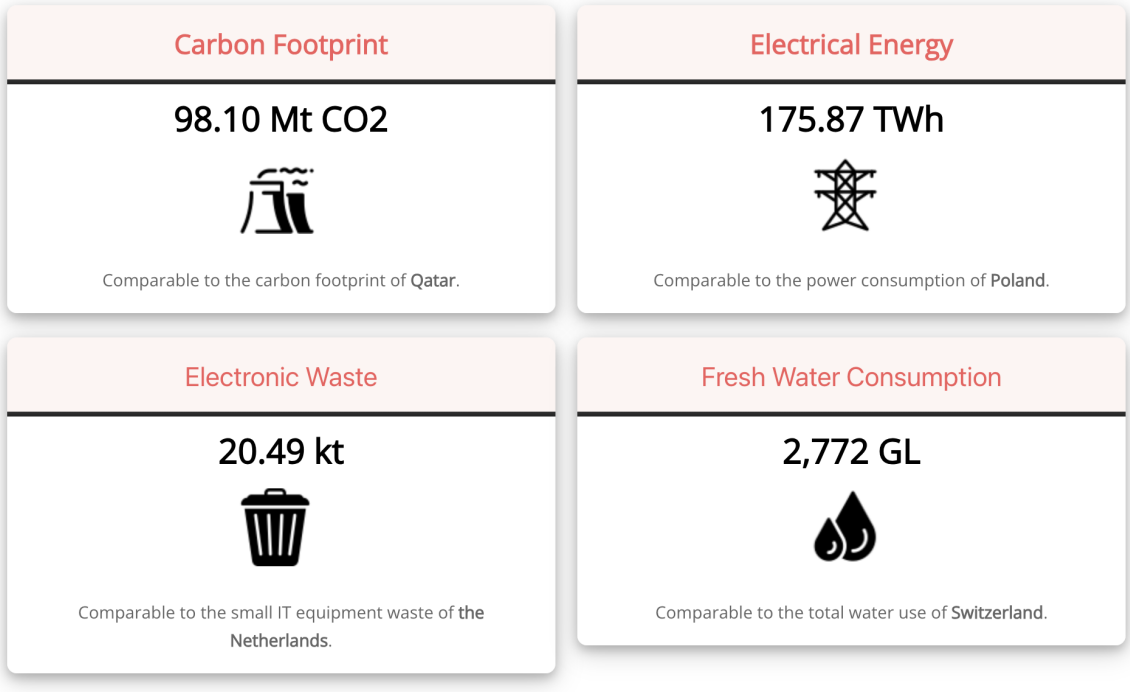



Figure 2.11: Annualized Total Bitcoin Footprints captured on 9th May [19].

The continuous block mining cycle incentivized people worldwide to mine Bitcoin. The mining process can generate a substantial stream of revenue[2] and participants are willing to upgrade their equipment to increase their chances. This results in an increase in concurrency and a raised block mining difficulty. The never-ending race this way requires more and more energy. See Fig. 2.12, which shows the current footprints for a single Bitcoin transaction.

The hardware used for mining is leading towards more specialized, less affordable hardware, and is preferred by individuals who have the resources to obtain such hardware. Bitcoin mining was originally fully capable of running on commodity hardware CPU (Central Processing Unit). This was later replaced by the GPU (Graphical Processing Unit) and eventually by ASIC (Application Specific Integrated Circuit). ASIC is hardware optimized for one task, enabling it to perform significantly faster. In the context of Bitcoin mining, that is optimization for the SHA-256 hashing algorithm. The problem with ASIC is that its usage is limited only to one specific algorithm and cannot be used for other applications.

Finally, security and decentralization traits limit Bitcoin throughput to 3-7 transactions per second. In comparison, Visa's centralized financial system already had a peak of 10,547 transactions per second [65].

## 2.8 Proof-of-Stake

An incentive to address these issues arose. One of them, motivated by high PoW energy usage, proposes a different approach in the consensus protocol. The proposal is Proof-of-Work (PoS), an energy-efficient and economically driven consensus mechanism in which validators (instead of miners) produce blocks with no extra overhead. Validators lock their assets for a certain period of time (i.e., stake) to gain the right to participate in block

[2]On 9th May 2025, the value from mining one block is approximately $327,000

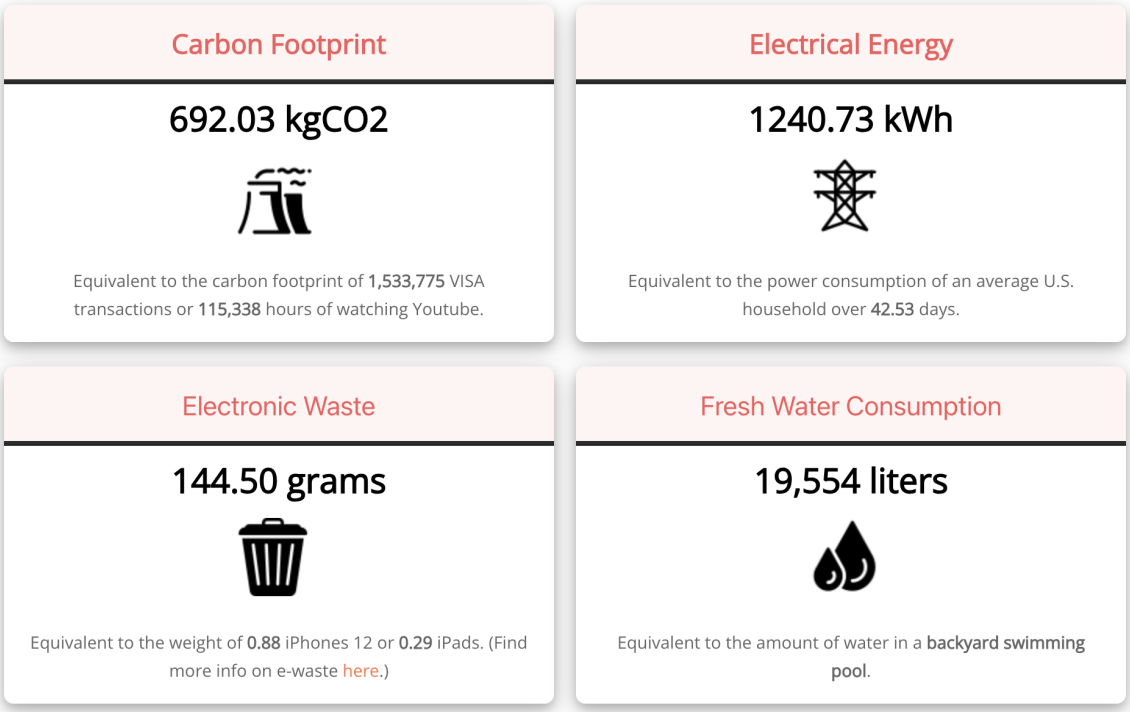



Figure 2.12: Single Bitcoin Transaction Footprints captured on 9th May [19].

generation. Validators are incentivized to act honestly since misbehaviour is punished by losing part or all of their staked funds. A selection of the validators is usually random, but may also be weighted by the amount of stake. The entry requirements for PoS as an actively participating consensus node are different compared from those for PoW. In PoW, a node requires a strong machine to have a reasonable chance of mining a block. A PoS protocol requires assets. The blockchain sets the minimum stake, which can vary. In Ethereum, this amount is set to 32 ETH [27], an equivalent of around $77,000[3]. Utilizing this staking mechanism gives the PoS blockchains higher throughput than PoW. The first implementation of PoS was introduced in PeerCoin [38].

### 2.8.1 Downside of Proof-of-Stake

While PoS offers significant advantages, they are accompanied by tradeoffs. The benefit of the increased throughput often comes at a cost of the sacrifice of either decentralization or security. In private permissioned blockchains such as Hyperledger Fabric [6], decentralization is highly limited due to the nature of limited participant access. These systems employ different consensus mechanisms, tailored for a centralized environment.

For protocols that do not want to sacrifice their decentralization property, the communication overhead lowers node scalability, and as a result, different modifications for PoS have been proposed, including hybrid PoS/PoW models, a scalable multilayer in a PBFT-style protocol for publicly available PoS blockchains [44]. These approaches aim to balance the benefits of decentralization for increased throughput and scalability.

[3]Calculated from Ethereum current price available at `https://coinmarketcap.com/` on 9th May.

### 2.8.2 Decentralized randomness

Besides throughput and scalability, PoS protocols also face a security challenge — specifically, a challenge with decentralized randomness. In a blockchain environment, where trust is limited, a single source of truth is essential to ensure consistency and fairness, prevent double-spending, and maintain consensus across all participants. In the decentralized random number generation (DRNG), the following properties are essential [58]:

1. **Bias-resistance.** The generated random output should adhere to a uniform distribution.
2. **Unpredictability.** Knowledge of one state or subsequence of states should not enable any participant to determine a prior state or predict a future state.
3. **Public-verifiability.** A third party, even one that did not participate in the random generation process, should be capable of verifying the validity of the random output.
4. **Availability.** Random number generation should be immune to manipulation by either a single party or multiple parties.
5. **Scalability.** The number of parties involved should not limit the random number generation process.

A sufficient DRNG is achieved by a combination of cryptographic techniques such as commit-reveal (e.g., RANDAO), verifiable random function (VRF), homomorphic encryption (HE), and zero-knowledge proofs (ZKP).

## 2.9 Commitment scheme

A commitment scheme is a two-phase cryptographic protocol that allows a sender (commitment owner) to deliver a value or statement to other protocol participants (receivers). Its content is hidden from the receiver until the sender reveals it (hiding property). Once the sender spreads the commitment, they cannot change the value or statement they had committed (i.e., infeasible for the owner to spread a commitment that can be later open to two or more different statements), also called binding property [26].

The protocol contains two phases:

1. **Commit phase.** The sender produces a commitment from the statement and publishes it to the receiver. To generate a commitment, the sender uses a private input created before with sufficient randomness, which is not shared with the receiver during this phase.
2. **Reveal phase.** The sender sends the private input, which was used in the commitment phase, to the receiver. The receiver can then verify the authenticity of the commitment by reconstructing the commitment using private input. Note that the sender has complete control over when the reveal phase begins.

### Hash commitments

The commitment can be represented as Commit(m, r), where m is the committed value or statement and r denotes a random value (i.e., nonce) to add further randomness and

ensure that the attacker cannot brute-force guess the m. The sender then uses the hash function Hash(m || r), the output of which is then sent to the receiver, where || denotes the concatenation of bytes. It is also essential to use an adequate hash function, providing a strong level of security against collision resistance. During the reveal phase, the sender reveals and publishes m and r to the receiver. If the receiver can correctly reconstruct the hash function output in the same way as the sender, we say that the commitment is valid [37].

### Pedersen commitments

Pedersen commitments use a mathematical cyclic group instead of the hash function. Specifically, two public generators, $G$ and $H$, are chosen on the cyclic group with order $q$ in a way that makes the discrete logarithm for $H$ with respect to $G$ not known (i.e., the value of $y$ for $y * G = H$ is unknown). Furthermore, given a random secret $r$ chosen from set $Z_q$ ($X$: $\mathbb{Z}$), the sender creates a commitment $C$ such that $C = r * G + m * H$, where $m$ is the secret value to be committed. The sender then publishes commitment $C$. For the reveal phase, the sender makes public $m$ and $r$. The receiver is then given $C$, $m$, $r$. The $G$ and $H$ remain public. Next, the verifier can confirm that $C$ can be created correctly from the provided information. Security of binding property can be proven as follows: given two commitments $C$ and $C'$ that $C = x * G + a * H$ and $C' = x' * G + a' * H$, where $C = C'$ and $a! = a'$. From this, we can demonstrate the calculation of the discrete logarithm by $H = \frac{(x'-x)}{(a-a')*G}$ [74].

## 2.10 Verifiable random function

A verifiable random function is a cryptographic primitive that enables the computation of a pseudorandom output $\beta$ on a given input $\alpha$ and produces proof $\pi$. The function with output $\beta$ takes a secret key — confidential information only available to its owner, and input $\alpha$. The functions with output $\pi$ take secret key, $\alpha$, and $\beta$ as inputs. This way, any party can verify using $\pi$ that $\beta$ was generated correctly. Additionally, the VRF's output is entirely unpredictable to a party that does not possess knowledge of $\alpha$ and the secret key [53].

Practical usage of the VRF function in blockchain is during the consensus round, where eligible participants generate a random value using the VRF function with the same input $\alpha$. Every participant then compares its output $\beta$ to the threshold. The threshold value is defined in the protocol. If the $\beta < threshold$, the participant is eligible to produce a block. However, there might be situations during which multiple participants satisfy this condition and also have the same $\beta$ value. This conflict is resolved by itself; whoever propagates the proposed block first wins.

On the other hand, there might be situations in which no one satisfies the necessary condition of the $\beta$ value. This rare case needs to be resolved so that no one is prioritized. A typical solution is to deploy a secondary deterministic selection of a validator that produces blocks. These blocks only apply when no producer was found in the primary selection.

In summary, the byproduct of none and multiple verifiers in VRF is not desirable behaviour. Although the VRF function is used in several PoS protocols, such as Polkadot [71] and Algorand [15], this issue still remains a concern.

## 2.11 Homomorphic encryption

Homomorphic encryption (HE) is a kind of encryption scheme that enables a third party to compute encrypted data. A typical scenario captured in Fig. 2.13 displays a scenario in which client encrypts their private data (message $m$) and sends encrypted data to the server. Later, the client requests a query with an evaluation of the function $f$ of their data. The server evaluates encrypted data and returns the encrypted result to the client. Finally, the client decrypts the received data and obtains $f(m)$. A real-world example is using confidential patient data in medical data analysis. Hospitals and research institutes can perform analytics like disease risk modeling directly on encrypted data while preserving privacy [1].

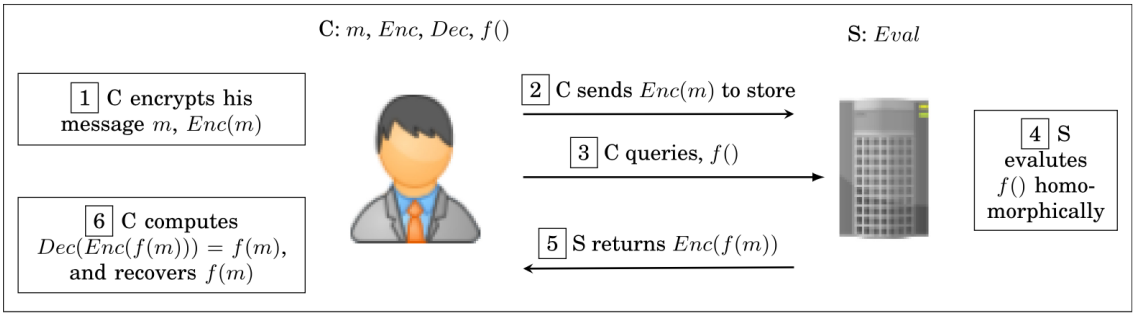



Figure 2.13: A simple client-server HE scenario where C denotes client and S stands for server [1].

In the context of blockchain, the HE can be combined with the VRF function to ensure unbiased randomness or, as presented in [25], to create homomorphic sortition as an SSLE encryption solution.

A main drawback of HE is its significant performance overhead.

## 2.12 Zero-Knowledge Proof

Zero-Knowledge Proof (ZKP) is a cryptographic method that allows one party (the prover) to demonstrate the truth of a statement to another party (the verifier) without revealing any information about the statement beyond its validity. This form of proof can be extremely effective in scenarios where details of the proved statement need to remain confidential.

Blockchains are designed to be transparent, but there are some situations where privacy is required. For instance, smart contracts may contain sensitive information such as personal identification or financial details, and their revelation could lead to security risks or breaches of confidentiality.

ZKPs come in two forms: interactive and non-interactive. The interactive method requires the prover to repeat the process for individual verifiers. Non-interactive; however, it allows the prover to generate proof that anyone can verify.

The core properties of the ZKP system are [69]:

- **Completeness.** If the provided statement is true and both parties correctly follow the protocol, the verifier can determine that the prover possesses sufficient knowledge.
- **Soundness.** If the statement is false, then no malicious prover can convince the verifier that they possess knowledge about the correct input.

- **Zero-knowledge.** If the statement is true, the verifier learns nothing beyond its validity from the prover.

A particular type of ZKP used in this is zk-SNARK, which stands for *zero-knowledge succinct non-interactive argument of knowledge*. Zk-SNARK construction algorithm (e.g., Groth16 [28]) derives its security and succinctness from bilinear pairing on a special Weierstrass-form, pairing-friendly curves such as BN254 or BLS12-381. A common implementation inside arithmetic circuits utilizes elliptic-curve operations efficiently using Twisted Edwards curves defined over the scalar field of the pairing curve. A major advantage of zk-SNARKs over traditional ZKP is that they are not interactive, which is a highly beneficial property for blockchain networks. The main disadvantage of zk-SNARKS is the required initial setup via multi-party computation. If the setup phase gets compromised, it can break the soundness property [7, 69].

As an alternative, zk-STARKs (i.e., *zero-knowledge scalable transparent argument of knowledge*) remove the need for initial setup, provide quantum resistance, and require weaker cryptographic assumptions. However, this comes at a cost of significantly bigger proof size. Zk-SNARKs' proof size complexity is $O(1)$, and for zk-STARK it is $O(log^2 n)$. In practice, this can mean that where a zk-SNARK Groth16 proof has hundreds of bytes, a zk-STARK proof can contain hundreds of kilobytes. This drawback limits the usage of STARKs in layer one blockchains limited. Additionally, zk-STARKs are new in the ZKP technology area and; therefore, their support in existing libraries is limited [49, 69].

### ZKP libraries

According to benchmark [21], we analyzed the selection of libraries to find the most suitable one for our implementation.

- **Circom**: a compiler for zkSNARK circuits implemented in Rust. The circuits can be further used in SnarkJS (implemented in JavaScript) or Rapidsnark to generate proof. The performance of SnarkJS is the worst in the benchmark. Both SnarkJS and Rapidsnark use the Groth16 algorithm [9].

- **Gnark**: an efficient framework of zkSNARK implemented in Golang that supports Groth16 and Plonk algorithms. It supports several curves, including BN254, BLS12-381, and BLS12-377. It also comes with a solid cryptographic library that already implements basic cryptographic constructs, including hashing and EdDSA signatures. Considering only zkSNARKs, this framework achieves the best result in terms of proof generation time and CPU utilization [12].

- **Starky**: a zk-STARK framework that achieves the best results among other zk-SNARK frameworks in terms of proof generation time and peak memory usage. However, due to the enormous proof size, this framework does not have practical usage in our work [64].

## 2.13 Security risks

This section outlines key security risks in blockchain systems and focuses in detail on the Denial of Service attack in PoS.

### 2.13.1 Double spending

The double-spending attack allows an attacker to spend tokens utilized in the blockchain more than once. For instance, assume that Alice is an attacker and decides to transfer her tokens to Bob. Alice also sends these tokens to Carol; however, she does not have enough tokens to send to both parties. Double-spending attacks may occur after the fork, during which previously completed transactions might be invalidated. Ultimately, the same tokens are promised to two parties but delivered only to one (see Fig. 2.14) [24].

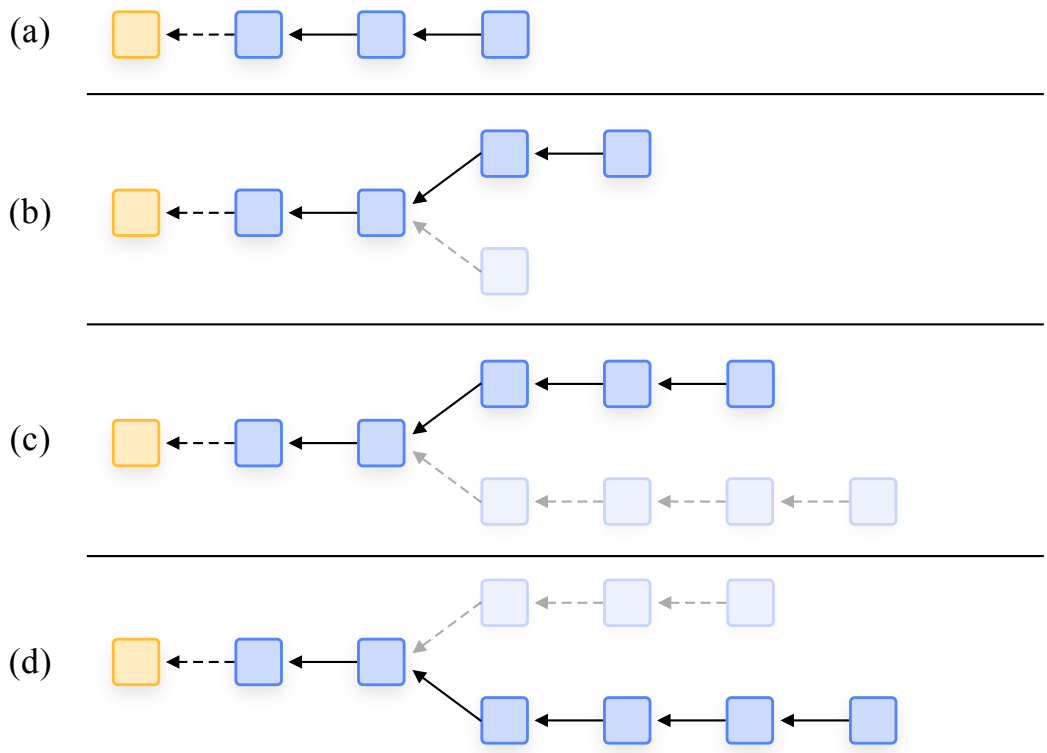



Figure 2.14: Illustration of double spending attack: (a) Initial state of blockchain in which all transaction are considered valid. (b) Honest nodes continue extending the valid chain by putting blocks (upper chain), while the attacker secretly starts mining a fraudulent branch. (c) The attacker succeeds in making the fraudulent branch longer than the honest one. (d) Part of main chain is overwritten by fraudulent branch [48].

From an economic perspective, this attack also leads to the invalidation of the uniqueness of digital assets, undermining trust. Although it does not increase the official monetary supply, it creates an illusion of a temporary value, similar to counterfeiting money, potentially creating instability in the economy.

In Bitcoin, transactions are considered confirmed after six blocks, as it lowers the probability of the occurrence of such an attack.

### 2.13.2 51% Attack

A 51% attack occurs when an attacker controls more than 50% of the computing power of a PoW blockchain or more than 50% of validation power (i.e., stake) for the PoS blockchain, allowing them to manipulate blocks and transactions in their favor that can compromise

blockchain trustworthiness and integrity. Such manipulation can involve declining transaction confirmation, freezing the network, rewriting recent blocks, applying double-spending attacks, and creating a monopoly to gather most of the rewards just for themselves.

Performing such an attack for a single entity is significantly expensive on resources, especially for already running public blockchains with high hash rates for PoW. However, this is more probable to happen within a mining pool that encourages individual miners to form pools (see Section 2.6.2) [33].

The case for PoS is affected by the current price for the tokens and the availability to buy these tokens on the open market. The price is derived from the market cap for the currency and the amount of all available tokens. Some of the existing PoS shifted consensus mechanisms to Delegated Proof-of-Stake (DPoS) to involve delegation and the election process, which, while not the primary motivation, helps to make this attack harder to realize [38].

### 2.13.3 Denial-of-Service attack

A DoS (Denial of Service) or DDoS (Distributed Denial of Service) is an attack that disrupts the normal functioning of a targeted server, node, or service by overwhelming it with a flood of internet traffic. For a successful DoS attack, the attacker must send more network requests than the victim can handle by depleting the node's resources [39]. In the context of PoS blockchains focusing on the consensus layer, this attack exploits the vulnerability of validator knowledge in PoS blockchains. It aims to disrupt blockchain availability or compromise a specific individual [63].

In PoS design, the validator identity is known in advance, which exposes them to targeted attacks. In PoS, nodes use their assets to participate in block production. These assets are locked for a certain period and cannot be unlocked in any other way. In case of malicious behaviour of the node, the stake is slashed.

An attacker can use the knowledge of the node identification to plan a DoS attack for a specific time to a specific target. Because reserved time to produce a block is inherently short (e.g., in Ethereum, 12 seconds), the time required to recover from an attack may be longer, resulting in an empty gap in the blockchain. The protocol will continue with the expectation to create a block from another validator, i.e., another target for the attacker. The attack can continue until the attacker does not have enough resources or the validator cannot defend itself against DoS. However, highly decentralized blockchains are designed so that anyone meeting certain requirements can join and participate in block production. This includes small nodes that cannot protect themselves against DoS attacks. In addition, nodes affected by this attack are punished even though they were prepared to produce a block, and their inactivity was not intentional. This leads to disincentivization of small nodes to participate in staking. Blockchain users who have resources that can be locked for staking but cannot or, on purpose, do not want to stake on their own. Instead, they delegate their stake to bigger institutions capable of defending against DoS attack, which ultimately reduce blockchain decentralization [43].

Realisation of denial of service can be combined with a Sybil attack, in which the attacker creates or controls many fake identities to influence nodes in the peer-to-peer network [22]. The vulnerability to a Sybil attack depends on how cheaply messages from new identities can impact the network, and the degree to which nodes accept messages from nodes that did not reach a certain level of trust in the system.

## 2.14 Ethereum

Ethereum was formerly based on PoW. However, this changed in September 2022 when Ethereum 2.0 was released, also known as Ethereum's PoS upgrade, which led to the transition from PoW to PoS. This results in an improvement of energy efficiency, scalability, and security. The impact on transaction throughput is negligible.

From a technical perspective, Ethereum 2.0 runs two layers: the execution and consensus layer, each with its distinct client software part. Both execution and consensus clients must communicate with each other. The execution layer retains responsibility for validating and gossiping transactions. When a validator is selected to propose a block, transactions from the node's transaction pool will be passed to the consensus client using a local RPC connection, packaged into Beacon blocks. Consensus clients then gossip beacon blocks on a peer-to-peer network. The consensus layer aims to achieve consensus among validators [27, 60].

In the context of the consensus mechanism, Ethereum now uses validators instead of miners in the PoS scheme. Anyone who owns at least 32 ETH can become a validator. During block production, validators are selected randomly to propose and attest to blocks. Validators receive rewards at both layers. On the execution layer, the validator receives rewards from priority fees and direct user payments, which are 0.1 ETH on average per block. On the consensus layer, validators are rewarded for block proposals, attestation, and participation in sync committees. The reward is approximately 0.04 ETH for a successful proposal and 0.00001 ETH for a successful attestation [27].

The malicious behaviour or acting differently than specified in the protocol (e.g., proposing and signing two different blocks for the same slot) results in penalties in the form of slashing, which removes at least 1/32 of the validator's locked assets [27].

A significant upgrade change was the introduction of the beacon chain, which takes responsibility for handling block and attestation management, executing the fork choice algorithm, and addressing rewards and penalties. Besides that, it also manages eligible validators inside the validator registry that are stored in the SSZ (Simple Serialize) list in the Beacon state. The hash-tree-root of Beacon state („Merkleized“ using SSZ) guarantees state integrity and enables efficient verification of individual validator entries [23].

### 2.14.1 Leader election

Ethereum's current approach to leader election for individual rounds involves a publicly computable random function. The function output always leads to a single validator. This cooperative approach contrasts with the PoW competitive approach, where every participant has an independent random chance of being chosen as a validator at any given time, with the chance being affected by their mining power corresponding to Poisson distribution. In addition, individual slots are not capped by a specific time limit. In Bitcoin, the average time to mine a block is 10 minutes, but occasionally it is more or less. The exact time value is not essential to reaching a consensus because participants determine the valid chain with minimal communication using the chain's strength, and the difficulty in mining a block is periodically adjusted after a certain number of blocks.

Proper synchronization and reserved time for every slot are crucial in PoS SLE. The PoS protocol allows the generation of blocks at a much higher rate because participants make decisions ahead of time about who will be responsible for block generation at specific slots. With their locked stake, the future leader then guarantees to follow protocol rules

and do the block generation once required. In Ethereum, each slot has 12 seconds, which is a significant improvement over the 10 minutes used in Bitcoin [27]. However, achieving security and decentralization in Ethereum has more overhead than in Bitcoin, resulting in a minor improvement in transaction throughput. The improvement of TPS is part of active research and development.

From the energy usage perspective, the PoS does not involve the computation of discarded values, thus vastly reducing electricity usage.

### 2.14.2 Denial-of-Service mitigation

In 2022, a major co-creator, Vitalik Buterin, shared the problem in single and non-single leader elections. Block proposers are chosen in advance using a deterministic random function. This way, anyone can obtain information about who will be proposing at what time. This makes future block proposers vulnerable to DoS and DDoS attacks that can be used to harm their reputation in the form of locked stakes as they have been unable to fulfill their obligations within the given time, or harm the protocol as a whole, making them unable to process transactions [14].

The DoS vulnerability is already known in PoS-based blockchain protocols. Algorand and Polkadot mitigate this issue by deploying VRF, which creates a new problem in the possibility of having multiple or no block proposers per slot (see Section 2.17).

Buterin's proposed solution is Whisk, a shuffle-based single secret leader election (SSLE) protocol for Ethereum to ensure only one proposer is selected every round and their identity remains unknown until they propose the block. At the same time, every block proposer can only know the rounds in which they have been elected to propose a block and prove it to others without revealing any information related to their identity [36].

### 2.14.3 Whisk

As of 2025, the Whisk protocol is still part of active research, and its current state is insufficient for real-world implementation. Several parts of the specifications are unclear or incomplete, including formal proof.

The protocol starts with initial candidate selection (see Fig. 2.15). The entire set of validators currently has approximately 262,144 ($2^{18}$) validators. From this set, 16,384 ($2^{14}$). These candidates are then placed in the candidate list inside the Beacon chain. This list is then shuffled by block proposers. According to protocol, block proposers are also supposed to provide a ZKP to demonstrate that shuffling was performed correctly.

However, the document does not specify in-depth details regarding ZKP, and the reference attached to the Github code refers to non-existent lines (e.g., one link refers to line 256 while the document on Github only contains 237 lines).

Whisk further specifies the shuffling of the set of 16,384 candidates as the shuffling of a 128x128 matrix. Each round, one row or column (selection of row and column changes every round) gets shuffled. The proposed algorithm is called Feistelshuffle [36].

At the end of the shuffling, Whisk introduces a cooldown to wait for RANDAO to collect enough entropy to make the output harder to predict.

Later in this work (see Chapter 3), we propose an SSLE protocol that shares some of the concepts used in Whisk.

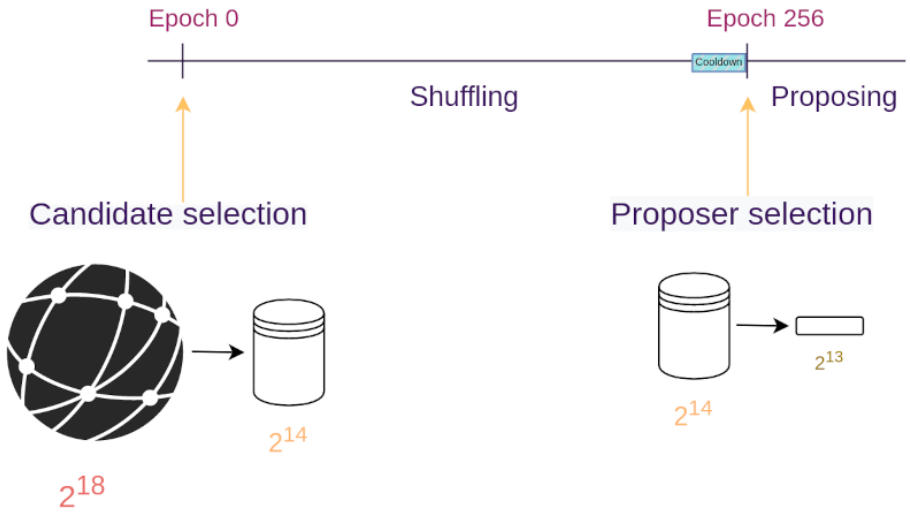



Figure 2.15: Whisk protocol flow overview [36].

## 2.15 Related work

In this section, we present two other well-known PoS-based protocols that are backed by academic research and aim to stay decentralized — Polkadot and Algorand.

### 2.15.1 Polkadot

Polkadot uses the NPoS (Nominated Proof-of-Stake) protocol. The NPoS adds to the existing PoS element of nomination, in which stakeholders can nominate individuals they consider trustworthy. This adds another level of security as a node with significant influence can quickly lose its power if individuals conclude it is untrustworthy. Polkadot uses the VRF function to determine validators for block production. In case of multiple validators, it is a race to see which block reaches the most of the network faster. In the case of no validators, the deterministic selection algorithm runs in the background, determining a secondary validator for block production. Blocks from secondary validators should always be produced. If a block from primary validity reaches the network, the block from the secondary validator is discarded [70].

### 2.15.2 Algorand

Algorand is based on PPoS (Pure Proof-of-Stake), which means that the selection of validators is based on the individuals' stakes. Consensus can be reached if non-malicious actors own a supermajority of the stake. The consensus mechanism uses VRF and consists of three steps: proposing, confirming, and writing a block to the blockchain.

During the first phase, every node runs the VRF function, and the number of staked Algo tokens influences the chance of being selected. If the node is selected, it propagates the proposed block along with the VRF output and proof.

During the second phase, nodes validate received VRF proofs and filter out the proposer with the lowest VRF output. After this action, each node will run a second VRF lottery to participate in the voting committee. If the account is eligible, the weight of their vote will

be based on the staked amount of Algos. Once selected nodes verify VRF proofs of each other, a committee is formed, and its strength is measured in the number of Algos.

In the final step, the selected committee checks for overspending, double-spending, or other problems with the proposed block and, based on the result, places their „certified vote“ on the proposed block.

Suppose a quorum is not reached in a certifying committee within a specific time. In that case, the network will enter recovery mode, and nodes decide to continue processing the last known block or proceed to a new block proposal [15, 3].

## 2.16 DAG-based consensus protocols

To address the issue of limited throughput, a new blockchain structure emerged in the form of a directed acyclic graph. The main advantage of this structure over a traditional back-linked list is that it can process multiple blocks at the same or similar height (see Fig. 2.16). This could lead to increased throughput.

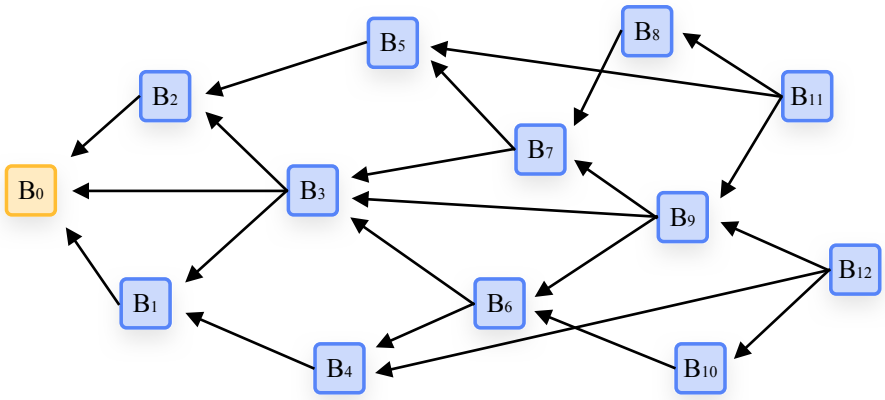



Figure 2.16: Example of DAG structure in blockchain. Time increases in the direction to the right [31].

However, this structure emerges with the problem of transaction collision, which does not exist in the traditional blockchain structure. This problem arises when the intersection of processed transactions in two or more blocks at a similar height is not empty. The primary purpose of having multiple blocks simultaneously is to increase the number of processes. However, if miners or validators have the choice to select transactions, rational behavior is to select those with the highest transaction fee to increase overall profit. If the produced blocks contain a high intersection of transactions, thus transaction collision is high, it only leads to increased redundancy, wasted computational resources, and no or very low improvement in transaction throughput [31].

There have been several attempts to create a blockchain with a DAG structure, such as IOTA [62] or Byteball [17], but at the cost of decentralization. These protocols aim for high transaction throughput. The transaction collision problem and double-spending attack are resolved by involving a centralized element that keeps the state of the blockchain secure [31].

Protocols PHANTOM and GHOSTDAG [66] propose to use random transaction selection instead of rational transaction selection. Authors, however, did not analyze the

potential attack of intentionally choosing the highest fee transactions. Work [31] showed by simulation that this behavior leads to significantly higher profit. Also, protocols do not provide proof for „random“ transaction selection. Thus, it is hard to prove which transactions have been selected randomly.

### 2.16.1 Sycomore

Sycomore's proposal aims to resolve the problem of transaction collision in the DAG structure by partially eliminating the option for block producers to choose which transactions should be included in the block. The blockchain DAG structure starts with one chain. If the demand to process transactions is high (i.e., the number of transactions or the size of the block crosses the threshold), the branch will split into two in the next round. Blocks are organized in the branches by their hash prefix. This means that transactions with zero at the first bit are assigned to the upper chain, the rest with one at the first bit get assigned to the lower chain (see Fig. 2.17). If transaction demand is still high (now evaluated for individual chains separately), branches continue to split. For the second split, they use the second bit from the prefix. This approach can continue for new splits with the following until transaction demand exceeds a threshold or reaches a maximum branching depth. If transaction demand is low, two previously created branches with the same branch parent can merge into one to save resources [5].

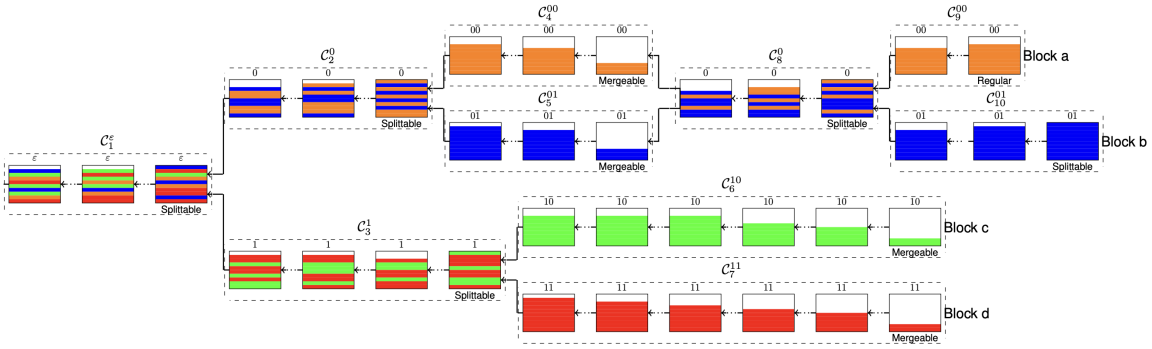



Figure 2.17: Example DAG structure in Sycomore. The $\mathcal{C}$ denotes specific chain [5].

The Sycomore allows for the dynamic adjustment of the size of the DAG structure based on the current number of transactions to process. The approach avoids transaction collision by employing an algorithm to sort transactions into the appropriate branch. Sycomore was designed for the PoW consensus protocol, and it has no real-world implementation.

We further adopted the idea of the Sycomore structure in our work.

## 2.17 Secret Leader Election (SLE)

In permissioned and permissionless distributed systems, the leader election protocol plays a fundamental role in coordinating system functionality and reaching consensus. Secret leader election (SLE) is a process to randomly and fairly[4] select a leader for one round [16].

[4]Fairness may be affected by the amount of locked stake, i.e., a higher amount of locked stake will result in a higher chance of being elected.

Furthermore, the SLE approach utilized in protocols such as Algorand and Polkadot is probabilistic, meaning that multiple leaders may be elected or none. This results in conflicting proposals or potential waste of resources, time delay, and reduced reliability.

The role of the leader in the context of this work involves the following activities:

- Process the maximum amount of transactions according to the protocol rules and group them into a block. This block is then shared with other blockchain nodes.
- Shuffle the election matrix during the shuffling phase. This action is skipped during the cooldown phase.

We emphasize that if nodes fail to produce the block, for any reason, then they will face the consequences in the form of loss of their locked stake. Specific loss amount varies in different PoS protocols. This condition ensures the minimization of empty slots without a block, which causes a temporary time gap in the protocol during which transactions are not processed.

## 2.18 Single Secret Leader Election (SSLE)

A single secret leader election (SSLE) emerged as a response to the DoS attacks to guarantee that exactly one, and only one leader, is elected every election round. The proposal of SSLE shows several schemes, including Decision Diffie-Hellman (DDH), which was adapted into Whisk [16].

Beyond performance considerations, SSLE enhances system resiliency by making it harder to conduct an attack on a leader. Leader identity remains cryptographically concealed until the leader produces a block and voluntarily uncovers their identity. Until the revealing event, no party knows the exact target to perform a DoS attack. The SSLE protocol guarantees that only one participant can provide valid proof of eligibility as a leader. Moreover, it is too late to conduct a DoS attack when the leader's identity is revealed after the block production.

SSLE must satisfy the following three attributes: [11]:

- **Uniqueness**. Only one leader is selected per election round.
- **Fairness**. Every participant has an equitable chance of being selected per election round (i.e., all participants have an equal chance, or their chance is amplified by their locked stake).
- **Unpredictability**. Every participant can only determine if they were selected as a leader for a specific round for themselves, but not for anyone else. This remains true until the moment the leader produces a block, thus deliberately revealing their identity. Utilizing an attack on the leader at this moment would not provide any benefits.

### 2.18.1 PoS and DAG-based Blockchain terms

Below, we explain a selection of the PoS and DAG-based blockchain terms used in our work. Some of these terms differ compared to Ethereum and other blockchains.

- **Transaction** - The smallest writeable unit in blockchain, which cannot be recorded on the ledger independently and needs to be included in a block first. It contains

the addresses of the sender and the recipient, the transferred amount, the creation timestamp, and the hash. The hash is then used to compute the appropriate block in which the transaction should be included.

- **Block** - A fundamental unit in blockchain containing a set of validated transactions, a creation timestamp, and the Merkle root derived from the Merkle tree of transactions. Furthermore, it includes the public key of the block's author, a reference hash to the previous block, and optionally a hash to the second parent in cases where the block was formed as a merge of two different branches. The block contains additional information related to DAG structure, including row, column, and depth. Finally, extra fields include commitment hash, commitment secret, commitment signature, and approximate sequence number in the blockchain.

- **Slot** - A position within DAG's parallel structure, which can hold at most one block per specific round. Every slot is uniquely associated with exactly one commitment to determine the responsible validator for producing this block for the specific branch. The DAG structure in our work allows for having empty slots based on current transaction demand and branch split (see Fig. 3.2 in Section 3.3).

- **Round** - The time window during which the selected leader must generate a block. Every round contains the same number of slots, predetermined in advance, and must be a power of two. This restriction allows easier sorting of parallel block processing in a DAG structure. The time frame is equivalent to Ethereum's 12 seconds to propose and propagate a newly produced block over the network.

- **Epoch** - A fixed-duration period composed of a predetermined number of rounds, used to periodically create a checkpoint. The checkpoint serves as a boundary that separates ready commitment sets, commitment shuffling, and commitment generation (see Fig. 3.7 in Section 3.4). Additionally, also improves blockchain finality.

- **Leader** - A node participating in the PoS blockchain that has been selected to propose a block in a specific round. To be eligible, the node needs to lock a certain amount of assets associated with its public key, ensuring accountability.

- **Shuffle phase** - A phase during which the leaders produce blocks containing entropy for the RANDAO beacon. The collective entropy is used to shuffle another set of commitments that will be used in the upcoming epoch.

- **Cooldown phase** - A phase during which validators cannot affect shuffling with their produced blocks to prevent the malicious creation of bias in favor of the last block producer.

# Chapter 3

# Design

This chapter outlines the design principles and mechanisms of our SSLE protocol. It begins with commitment structure design, followed by a summary and discussion of its requirements (i.e., uniqueness, fairness, unpredictability). The design then shows the creation process of commitments and presents its usage in our DAG blockchain structure. We also describe the protocol flow and node peer-to-peer communication.

## 3.1 Commitment Structure

This section provides a detailed description of the ZKP commitment scheme used in our protocol. We first present the structural components of the scheme, then argue key design decisions, and finally, we discuss how the scheme is integrated into our consensus protocol.

### 3.1.1 Epoch synchronization

Each epoch requires a set of commitments to be generated and prepared from the preceding epoch. Node clocks are synchronized so that all participants begin the commitment-generation phase immediately at the beginning of a certain round. In our implementation, this event is triggered at the start of the second round of the epoch. However, exact timing in deployed protocol may vary based on parameters such as the maximum number of commitments per node, the maximum number of parallel blocks per round, the number of rounds per epoch, and the number of active nodes. The commitment generation needs to start early in the epoch and finish in the shortest possible time so there is enough time to shuffle the generated commitments.

The initial commitment generation starts by processing `BCHAIN_INIT_COMMITMENT`. This process is executed concurrently as a separate goroutine[1] in Go. To mitigate the risk of linking the generated commitments with their creator based on timing, the goroutine sleeps for a very short time to create a more randomized interval before producing each commitment.

### 3.1.2 ZK-SNARK setup

Each commitment in our scheme relies on a zkSNARK proof, which requires an initial setup specific to the target arithmetic circuit. Proofs are bound to a circuit; changing the circuit parameters means starting over with a new setup. This downside limits the flexibility of

[1] `https://go.dev/doc/effective_go#goroutines`

zkSNARKs. The gnark library provides a codebase to create the setup with an output of proving key (PK) and verifying key (VK). Generating these keys requires a trusted setup, a.k.a. „setup ceremony." In real-world applications, the multi-party computation (MPC) protocol is used to compute a joint function of participants' private inputs while revealing nothing but the output [45]. This way, multiple independent parties contribute entropy to interactively construct the common reference string (CRS) — a collection of public parameters for PK and VK. The critical component is the generation of secret parameter $\tau$, which is compounded from individual contributions $(\tau_1, \tau_2, ...\tau_n)$ where $n$ denotes number of contributions. It is crucial that $\tau$ is created in an irrecoverable way (a.k.a. toxic waste removal). As long as at least one participant honestly destroys their secret, the original toxic waste cannot be recovered. After this process, CRS consists of the powers of $\tau$ and encoded circuit data using $\tau$.

In our work, we leverage the gnark library to generate PK and VK under the assumption of a secure, single-machine randomness source. This is not fully usable for real-world scenarios and should be extended by MPC for practical application. However, for our work, the gnark's approach is sufficient. To optimize the performance during setup, we precomputed and persisted the binary representations of PK (`pk.bin`) and VK (`vk.bin`), which are loaded during the node initialization process.

### 3.1.3 Digital signature scheme

We adopt the Twisted Edwards curve Digital Signature Algorithm (EdDSA) instantiated over the BLS12-381 Twisted Edwards curve instead of the more traditional Elliptic Curve Digital Signature Algorithm (ECDSA). The decision was motivated by the superior efficiency of Twisted Edwards curves in ZKP arithmetic circuits and deterministic signature generation — an essential property for our commitment scheme. For ECDSA signature $(r, s)$, it is possible, under certain conditions, to recover the public key [13]. Keeping the public key confidential is crucial to prevent DoS attacks.

Every node generates a private scalar a (private key) and derives a public key A using scalar multiplication $A = aB$, where $B$ is the standard base point for the BLS12-381 curve. The public key is used as a unique identifier of the node on the consensus layer. Using this key, nodes can authenticate themselves that they locked their assets for staking. In a real-world implementation, the pair of keys would also serve for block and transaction signatures to ensure that no third party can impersonate the legitimate key-holder. It should be noted that both keys are inherently generated in elliptic curve cryptography. Thus, the public key is uniquely tied to its corresponding private key.

### 3.1.4 Commitment components and circuit variables

Our ZKP circuit encompasses six public inputs and twenty-six secret inputs. The six public inputs include:

1. First EdDSA signature ($sigMsg$) encompasses three public variables: the signature's compressed point [35] $R$ comprised of two variables, $x$ and $y$ coordinates, and scalar $s$.

2. Public message ($pubMsg$), used in combination with secret $b$ to create hash $m$ and signature $sigMsg$ for $m$.

3. Epoch number, which needs to correspond to the value used in $pubMsg$.

4. Merkle root. The root hash of the Merkle tree containing public keys of eligible block producers.

The twenty-six private variables include:

1. Prover's public key. It is a point; thus, it is stored as two variables.
2. The second EdDSA signature ($sigSec$) created on epoch number.
3. Secret $b$, derived from $sigSec$.
4. Hash $m = MiMC(pubMsg||b)$, where $||$ denotes concatenation.
5. Starting leaf for Merkle proof, a hash of the prover's public key.
6. Merkle path hashes (nine) and position indices for Merkle-proof construction (nine).

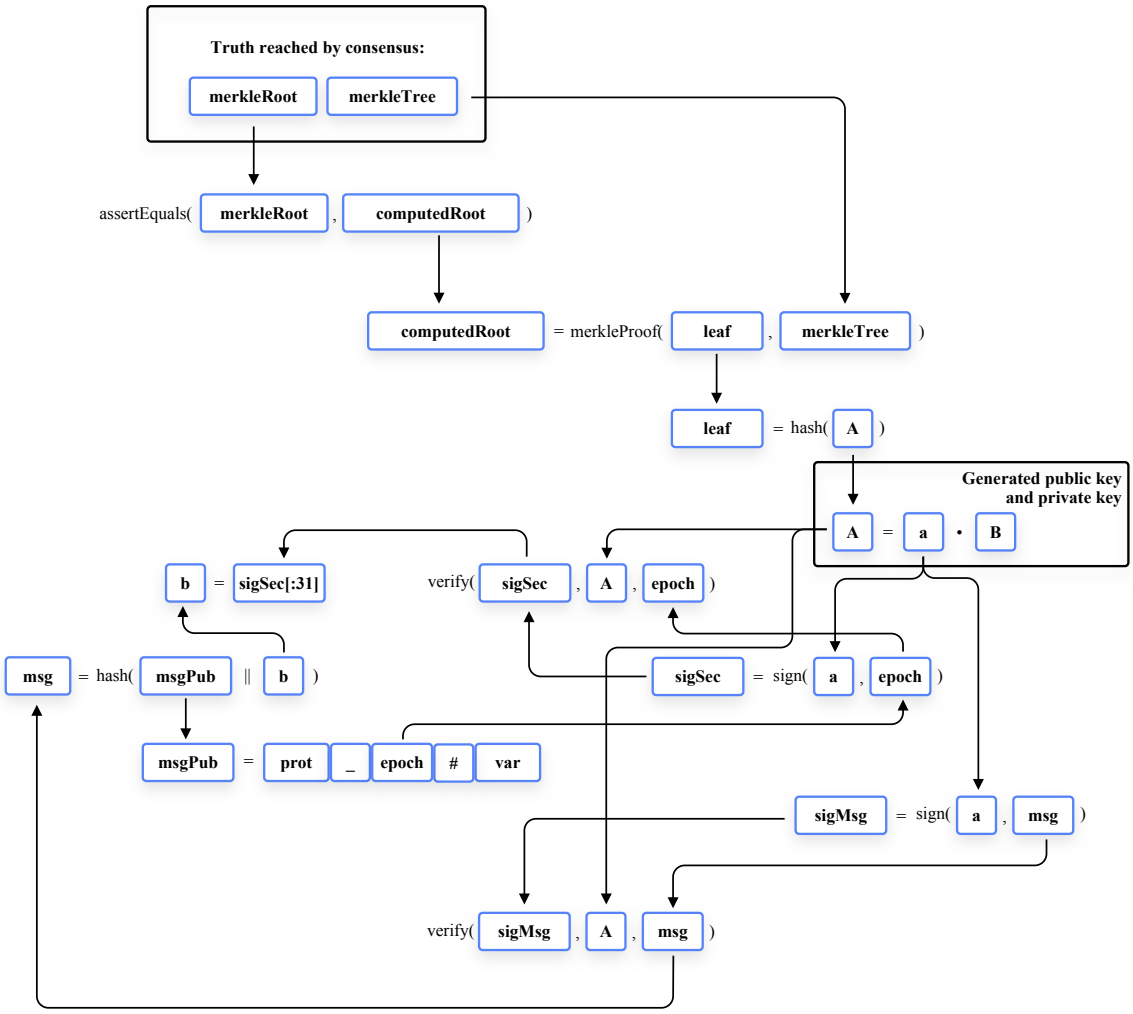



Figure 3.1: Structure of the commitment scheme visualized with individual components and their dependencies.

The commitment structure as a whole might be complicated to apprehend at first glance. Therefore, we developed Fig. 3.1, which graphically represents the relationships between individual parts of the commitment.

The prover's public key is used to verify signatures created by the same prover. The signatures are created using their private key. The private key is not included in the circuit. We emphasize this approach as a benefit of our design. The private key cannot be compromised under any circumstances. In the case of a manipulated ZKP setup, the attacker will not be able to retrieve it.

The purpose of $sigMsg$ is to create a binding between the prover's public key and commitments for the epoch. The length of the message is defined during the node initialization process as a string length of the protocol version, epoch separator character (i.e., „_“), a 10-digit zero-padded number representing the current epoch, variant separator character (i.e., „#“), and a 5-digit zero-padded number representing the variant. The protocol version serves as a unique identifier for the protocol so nodes can synchronize to use the same version in case of an update. The term epoch for our work is defined in Subsection 2.18.1. The variant is a number that allows protocol participants to generate multiple commitments per epoch. This does not apply to real-world scenarios where the set of potentially elected participants is large enough to cover all slots for the epoch. However, for experimental purposes, we allow the creation of a network of a minimum of two nodes. In such a small network, allowing participants to produce more than one block per epoch is necessary. The maximum number of variants is specified in the configuration file loaded during node initialization.

We emphasize that there is a difference between the message m and public message pubMsg used in $sigMsg$. The $m$ is created as a MiMC hash. The MiMC is a family of ZK-friendly cryptographic hash functions with minimal multiplicative complexity, enabling high arithmetic circuit efficiency [2]. The one used in our implementation is instantiated for the BLS12-381 scalar field. The purpose of $b$ is directly connected with the second signature, $sigSec$. We argue that we cannot use pubMsg in the sigMsg. This would result in a risk of revealing the prover's identity. Assume that the attacker has public keys of nodes involved in block production. They are publicly available because we need to be able to connect the identity of a block creator and participant of the network who decided to lock their state for the right to produce a block. A subset of them are directly involved in the upcoming epoch. The verify function for the given EdDSA signature requires two parameters to provide a boolean validity output — the message and the prover's public key. If the message is publicly available, then the public key will be the only remaining variable to verify the EdDSA signature. This means that the attacker can sequentially pick public keys and, by brute force method, find outputs that match valid signature output and reveal the commitment's author identity. For this reason, we keep the value of the message secret and reveal $pubMsg$ instead.

The property for Twisted Edwards curves over BLS12-381 used in EdDSA produces deterministic signatures, meaning the same message signed with the same key will always produce the same signature. This is a required property for our approach because this allows us to bind a public key with the $pubMsg$ with only one signature. The message's signature is mandatory and only allows the definition of several fields mentioned above. This way, the commitment producer cannot produce more commitments for the same variant in the same epoch using the same private key because the signature would be the same. It is only possible by changing the content of $pubMsg$, m, or using a different private key. Changing the content of $pubMsg$ would be denied once a commitment is received upon public variables validation. Changing the content of m would be rejected by ZKP proof verification on MiMC hash constraints with $b$. The $b$ needs to be produced in a way that no one except the commitment producer can determine, but at the same time, it needs

to be a value that can be produced only once per epoch. It cannot be a random value because provers then could create an indefinite amount of commitments and increase their chance of being elected. This could be eventually detected during block production, but it is behavior that disincentives other participants. For this reason, the epoch number should be the same for all commitments in the same epoch. However, it cannot be used in isolation because it would no longer remain confidential. In our approach, we have made a second signature, $sigSec$, which is created from an epoch number string. The signature output is used as a secret $b$. The $sigSec$ and $b$ are secret variables in the ZKP circuit. The verifier easily verifies that $sigSec$ was created from the desired epoch number string. The $b$ is then used in concatenation with $pubMsg$ in the MiMC hash function.

Another option to alter $sigMsg$ is to use different private and public keys. An attacker could manipulate $sigMsg$ by simply generating a fresh key pair and publishing commitments without authorization. To prevent this, our commitment structure requires a construct to verify that the public key used in the commitments is in the set of participants' public keys who are eligible to produce blocks — yet do so without revealing that key. We cannot use a data structure whose size can dynamically change. The ZKP circuit requires a precise size that was known before. A naïve fixed-size list of eligible public keys, padded with dummy value values and sorted, could be scanned in $O(n)$ time complexity. However, such looping and comparisons are expensive to implement directly inside a ZKP circuit. Instead, we organize the authorized keys of eligible block producers into a Merkle tree structure of predetermined depth (in our implementation, we use nine levels, supporting up to 512 participants). The Merkle tree needs to be constructed out of the ZKP circuit, and its content can change based on the number of eligible block producers. In our protocol, the Merkle tree is constructed once a node receives a message to share its public key with nodes. In real-world usage, nodes would additionally verify if this public key locked assets to participate in block generation. This way, all nodes retrieve all public keys in a list, sort it lexicographically, and fill the remaining values with „0“. This list is then used as the bottom level of the Merkle tree. This way, all nodes reach consensus to a final Merkle tree state and compute root. In the ZKP circuit, the prover then uses a hash of their public key, hashes of the nodes required to construct Merkle proof, and path indices to determine direction based on the starting position in the tree. The ZKP circuit then verifies the validity of the Merkle proof. Finally, the circuit checks that the hash of the public is equal to the hash provided for the Merkle proof. If they match, the creator of the commitment belongs to the set of eligible block producers[2].

### 3.1.5 ZKP circuit performance analysis

The current limitation of 512 eligible block producers is set for experimental purposes. It is not costly for the ZKP circuit to exponentially increase the capacity as it adds two additional secret variables in the ZKP circuit per level. The average proof-generation time on commodity hardware is 600–1,100 milliseconds. The circuit comprises 20,794 constraints: 14,078 for two EdDSA signatures, 6,048 for Merkle proof (for Merkle tree of depth nine), 666 for the MiMC hash, one to compare the provided hash root with computed one, and one to check the provided hash of secret $b$ with computed one.

[2]We note that the public key hash comparison is not implemented in our work due to Gnark's bug of incorrect behavior while hashing 32 or more bit values. The result of the hash leads to 0 and further undefined behavior.

### 3.1.6 Verification of secret variables on block publish

We implemented the protocol so that every node also publishes its public key, commitment hash, secret, and secret signature on block publish. Every node that receives this block performs additional verification that this signature belongs to the original author of commitment, the secret $b$ was created correctly, and using a hash function on $b$ and $pubMsg$ matches the $msg$. This way, nodes can conclude that the block was produced by the eligible author who was responsible for creating that specific block.

### 3.1.7 Possibly easier approach

During the study and implementation of this work, a simplification to the commitment scheme was identified. In contrast to our chosen commitment scheme, it only requires one signature ($sigMsg$). The $sigSec$, $b$, hash of the $msg$ and $b$, epoch round can be omitted. The change is to make $sigMsg$ as a secret variable in the ZKP circuit. This approach can reduce the complexity of the ZKP circuit. However, it was not studied in detail to prove it does not create vulnerability in the protocol for the attackers or compromise the fairness of the protocol. Consequently, it provides a promising direction for future work research.

## 3.2 Commitment scheme design summary

This section summarizes the system's SSLE requirements and addresses potential vulnerabilities in the commitment scheme.

### 3.2.1 Commitment identification within the shuffled bucket

To allow each node to determine its commitment's position before and after shuffling, we compute a unique identifier by encoding the commitment's output (i.e., public variables and proof) in Base64 encoding and hashing the result with Keccak-256. These hashes are stored in a list (the „bucket“) and remain invariant during shuffling, enabling any node to locate its commitment by matching hashes.

### 3.2.2 One commitment per slot (SSLE uniqueness)

Each slot in the blockchain DAG structure corresponds to a single commitment in the shuffled bucket. The commitment is identified by a hash. Content to create such hash is linked with precisely one public key using EdDSA signatures. Any tampering with the message, signatures, epoch, or variant invalidates the ZKP proof and alters the resulting hash.

### 3.2.3 Unlinkability of commitments to creators (SSLE unpredictability)

The creator's public key is kept as a secret variable in the ZKP circuit to prevent adversaries from correlating commitments with their original authors. In the Merkle tree, only the root is public. Thus, the public key cannot be derived from the Merkle proof path. Moreover, both signatures used in the circuit contain secret input. Without knowledge of these secrets, an attacker cannot feasibly recover a creator's public key using brute force or statistical analysis of signature outputs.

### 3.2.4 Equal election opportunity (SSLE fairness)

In our protocol, all participants must stake a uniform, fixed amount of tokens (e.g., 32ETH) to qualify for block production. By using this approach, we ensure that all participants have the same chance of being elected. An alternative approach is to allow the size of the locked tokens to be variable, resulting in biased selection towards larger stakeholders.

### 3.2.5 Bound on commitments per epoch (SSLE fairness)

To limit each node's generated number of commitments per epoch, the public message ($pubMsg$) is a public variable and contains a strict format about protocol version, epoch number, and variant. Altering any of those variables would result in an invalid commitment not accepted by the network. The protocol enforces an upper bound on the variant, which allows every participant to generate the maximum amount of commitments. The $pubMsg$ is also used in combination with secret value $b$ to form a hash m, which is signed using the private key. The correct signature is verified by a public key and included in the commitment as a secret variable. Any deviation — such as incrementing variant beyond the allowed range or altering secret m results in an invalid ZKP proof or invalid EdDSA signature verification also on the ZKP proof verification level.

### 3.2.6 Eligibility verification without exposure (SSLE fairness)

To prevent nodes from newly creating a pair of keys without a locked stake to manipulate the fairness of legitimate commitments, we incorporate two mechanisms into the ZKP circuit to authenticate a node's eligibility.

- **EdDSA signature binding.** The prover must produce valid EdDSA signatures using their private key for both inputs, m, and epoch number. Since only the keys' owner can generate these signatures, this mechanism prevents adversaries from publishing commitments using artificially created keys. The signature is verified using the prover's public key.
- **Merkle Tree proof.** Nodes share their public keys before the commitment phase begins to form a fixed-depth Merkle tree. Empty places in the Merkle tree are filled with zero value. Each node then computes Merkle paths up to the root. The prover submits in ZKP proof Merkle path, starting leaf (hash of their public key), and path indices, all as secret variables. The Merkle root is a public variable. This publicly available Merkle root is compared to the root computed from the provided Merkle proof. Additionally, we also propose a check during the ZKP verification process to hash the public key and compare it to the provided hashed starting value for Merkle proof.

### 3.2.7 Liveness and accountability for missed slots

Our SSLE-preserving scheme using ZKP conceals the identity of elected leaders until block publication to prevent denial-of-service attacks. The downside of this approach is that we can no longer punish them (by slashing their stake) in case they do not fulfill the obligation of block production. While our implementation does not address this problem, we propose two approaches to fill this gap.

- **Economic disincentive.** Our protocol uses a block reward for every block that has been published. We argue that purposely not publishing a block is not incentivized behavior for block producers because they miss the financial rewards. If the network contains many stakers and active DAG branches, missing out on a block creates a negligible delay in transaction processing if the block production time is set sufficiently low (e.g., 12 seconds).

- **Delayed commitment reveal and punishing slashing.** Nodes that participated in the election for epoch commitments have to reveal their commitments (along with their secret variables) that have not been chosen for any slot. This is only applicable if there is at least one empty slot with an ungenerated block in the epoch. The time window to share those commitments is much larger than for generation. Failure to do so signals malicious behavior, triggering stake-slashing. We cannot differentiate malicious behavior from not publishing commitment. This approach, while more complex and strict, enhances protocol resilience by distinguishing between non-selected participants and deliberate withholders.

## 3.3 Commitments shuffling

Before block producers can determine their assigned slots for the upcoming epoch, all nodes must agree on the same, uniformly randomized ordering of the collected commitments (bucket). To reach this, each node performs the following steps:

1. **Commitment identification and sorting.** Each commitment, comprising its public inputs and ZKP proof, is serialized using Base64 encoding and hashed using Keccak-256. These hashes serve as commit unique identifiers. Nodes then sort the list of commitments by commitment hashes using lexicographical ordering.

2. **Deterministic shuffling.** To achieve unpredictable yet deterministic permutation of the bucket, the Whisk describes Feistelshuffle [36] while utilizing the RANDAO beacon. In our implementation, we abstracted RANDOM and used fixed key strings for shuffling. Moreover, we use shuffle implementation in Go, which swaps positions of commitments in the bucket.

3. **Bucket hash agreement.** During the last round, nodes share the hash of their shuffled bucket to compare if they have the same content. If the consensus is reached, nodes use this commitment in the upcoming epoch. The bucket is then interpreted in the form of a two-dimensional matrix to determine the correct slot by row and column.

The structure of the epoch is illustrated in Fig. 3.2. Within each epoch, the DAG structure of the blockchain is formed inside a grid of a fixed size and utilizing binary tree principles to facilitate block splitting and merging. Blocks are aligned according to the round in which they were produced, a property maintained by strict time synchronization across rounds. The default time value for a round is 12 seconds, but it can be modified. This time window allows nodes first to determine which slots need to be filled with blocks based on the decisions made in the previous round—whether to continue a branch, split it into two, or merge two branches into one. Each node compares its bucket of commitment position during which they are expected to produce a block with computed available slots.

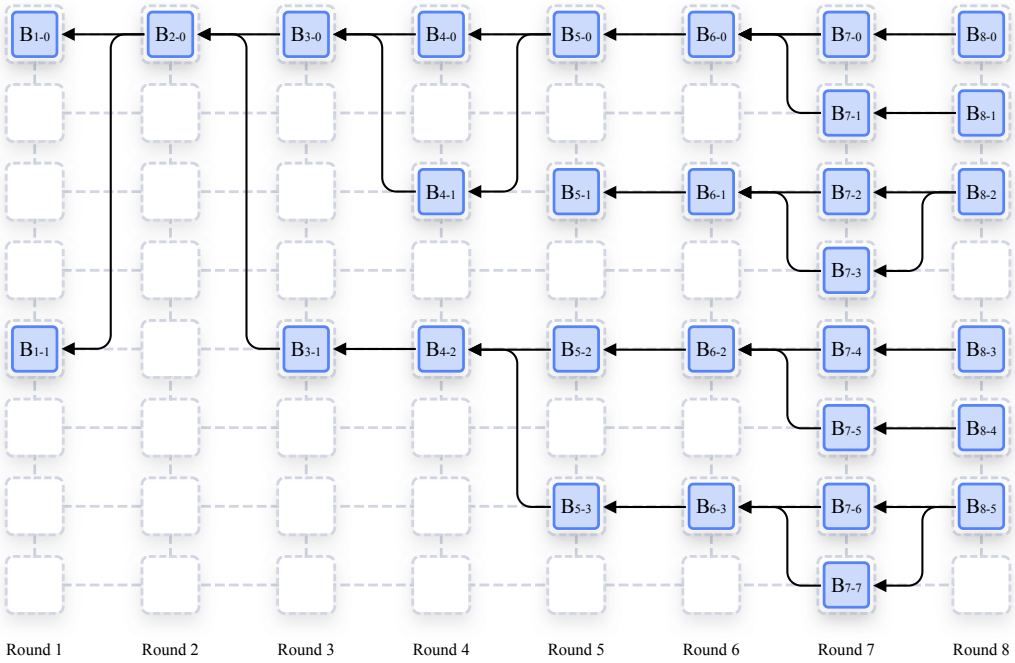



Figure 3.2: Structure of a single epoch. One epoch contains several rounds, and one round contains several slots (vertically). Each slot may or may not contain a block. Time is increasing in the right direction.

The commitment position is then translated to position in the grid in column-major order. Specifically, the position is calculated using the formula: column (round) + a maximum number of parallel branches + row (branch). In our implementation, the default value for a maximum number of parallel branches is eight, though the implementation supports up to 64 branches. Theoretically, this limit could be increased further; however, practical constraints such as the number of rounds per epoch impose limitations. A higher number of rounds requires a greater number of commitments, which could become resource-intensive for a single node to process a significantly higher number of blocks and commitments in one epoch. During the 12-second time window, nodes concurrently produce blocks if they have the right commitments for the correct slots, broadcast them, process incoming blocks, process commitments for the epoch, and handle incoming transactions.

For a block to initiate a branch split in the subsequent round, it needs to exceed a certain threshold in either block size or transaction count. In this work, we only consider amount of transactions. When a branch is triggered to split, transactions associated with the current branch get sorted by their hash. This approach is inspired by [5]. In our approach, a node evaluates a character in transaction hash in its hexadecimal representation based on the current branching depth (see Fig. 3.3 and Fig. 3.4). If the character falls within range 0–7, the transaction is assigned to the first slot (labeled 0); if it falls within range 8–15, it is assigned to the second slot (labeled 1).

The split function follows a specific algorithm. The result of the split always results in one branch that continues the branch from the original slot, and the second slot is computed as a half addition to the current nesting. Furthermore, due to the commitment-to-slot mapping strategy, nodes with valid commitments are not guaranteed to produce blocks, except in the case of the branch at row 0. This structure reduces the likelihood of block

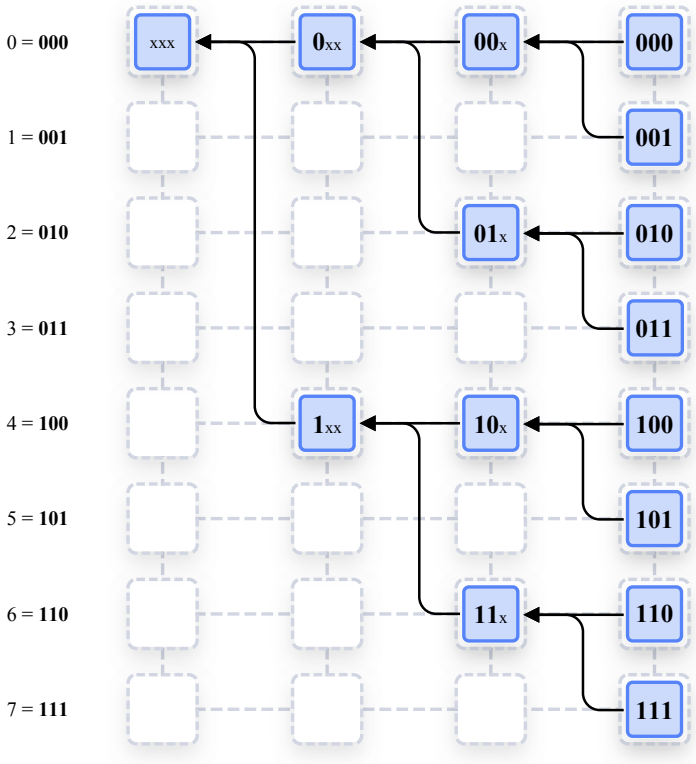



Figure 3.3: Calculation of hash mask for transaction applied in DAG structure. The `x` value states that the bit value is irrelevant to the result.

production in deeper branches, particularly for the second slot positions. Consequently, the reward earned from block production is highly influenced by transaction demand, which is a potential drawback in structured DAG blockchains where the number of blocks in individual rounds can vary dynamically. Nevertheless, provided that the commitment shuffling and randomization processes are unbiased, all nodes have an equal probability of being assigned to any slot in the upcoming epoch.

The transaction load must fall below a defined threshold for merging in both branches to trigger a branch merge. In other words, this means that two blocks contain a sufficient amount of empty space, and continuing in a merged path is more efficient. The selection

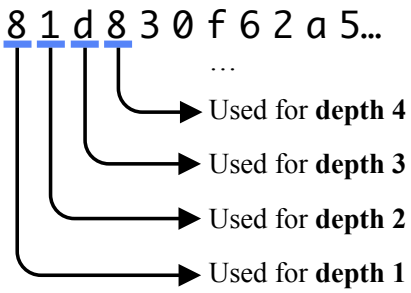



Figure 3.4: Calculation of hash mask for transaction. An individual hexadecimal number is used based on the depth of the slot.

mechanism for merging works the same way as for the split but is applied in reverse order (see Fig. 3.2).

The additional visualizations are display in Fig. 3.5 (a)–(d).

## 3.4 Protocol flow

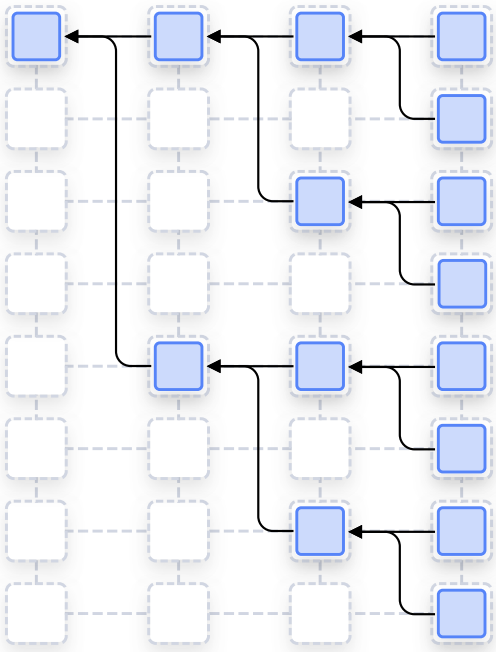
This section describes initial and ongoing protocol flow.

### 3.4.1 Initial protocol flow

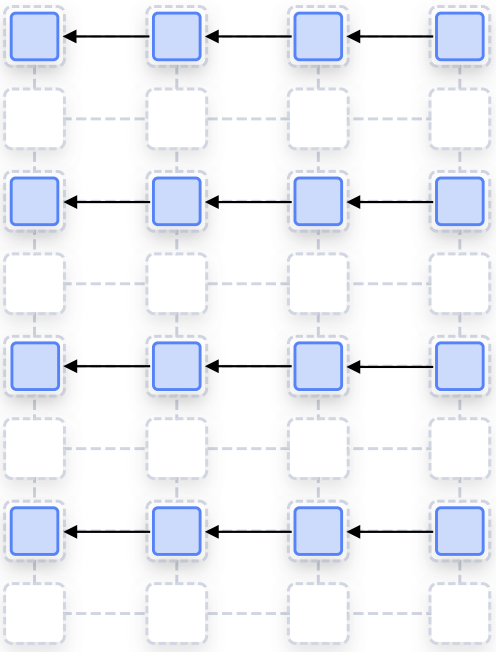
Once the nodes finish their initialization, they create a connection with their peers and prepare the Merkle tree structure of public keys. At this point, they still require the genesis block to start forming a blockchain structure. In conventional implementations, the genesis block is hard-coded inside the implementation. However, in our design, it is provisioned dynamically by a data-process service. This allows us to wait for all nodes to be ready to start at the same state, preventing some nodes from initiating protocol exchanges while others are still forming connections. Additional communication is required to prevent it, and in our approach, it is handled by another service. After the genesis block is processed at all nodes, then every node issues a `BCHAIN_INIT_COMMITMENT` message followed by a `BCHAIN_COMMITMENT` message for the first epoch (see Fig. 3.6). The latter message triggers nodes to start generating commitments. The output of commitment generation is a tuple: public witness, a data structure that contains public variables for the ZKP circuit, and ZKP proof. Public variables are only valid for the provided proof. The provided tuple is then encoded into Base64, encoding, from which is computed Keccak-256 hash.

Additionally, the node stores the commitment's secret variables, $b$, and $sigSec$, which will be disclosed alongside the produced block to allow later verification of the commitment's validity. Nodes then broadcast newly generated and received commitments to their peers, validate each commitment's proof and public variables, and append them to the commitments bucket list data structure. Once all nodes have generated their first-epoch commitments, they exchange the `BCHAIN_INIT_BUCKET_INFO` message. This message is only used for the first set of commitments generation. It signals the preparation for a second round of commitments, which is necessary because the second bucket, to which randomness from block production is used, is empty. Following this, each node issues a `BCHAIN_BUCKET_CHECK` message containing the Keccak-256 hash of the commitments of the bucket filled with commitments.

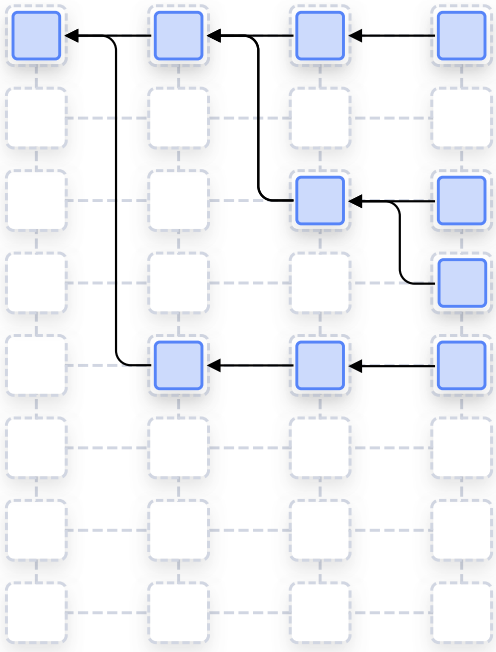
After all nodes finish the bucket hash check, they generate a set of commitments for the second epoch and repeat the bucket hash check again. With two commitment buckets established, nodes exchange a `BCHAIN_SYNC_TIME` message to synchronize their clock to the start of block generation. The time is aligned to the nearest upcoming minute with zero seconds. However, if the node's current time is more than 45 seconds past the minute mark, it delays for five seconds to either synchronize to the closest minute or adopt the timestamp received from peers. At the synchronized start time, nodes begin to generate blocks for the epoch using a set of commitments from the initial bucket. Block data serves as the source of randomness for shuffling commitments in the subsequent epoch. In our implementation, the shuffling is fixed because we have not implemented a RANDAO beacon. Because no prior randomness exists for the initial bucket of commitments, the data from the genesis block are used during the shuffling process.



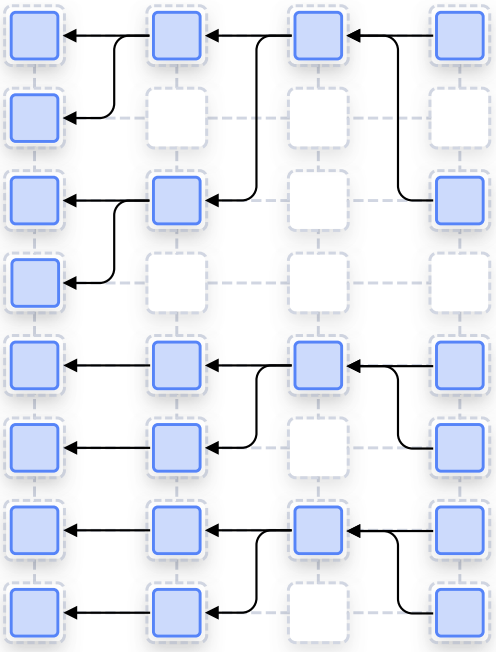
(a) Structure for maximum transaction throughput.



(b) Structure for uniformly distributed transactions.



(c) Structure with a high demand for a specific transaction hash pattern (artificial).



(d) Structure for a random short-term high transaction demand.

Figure 3.5: DAG blockchain structures with different transaction demands.

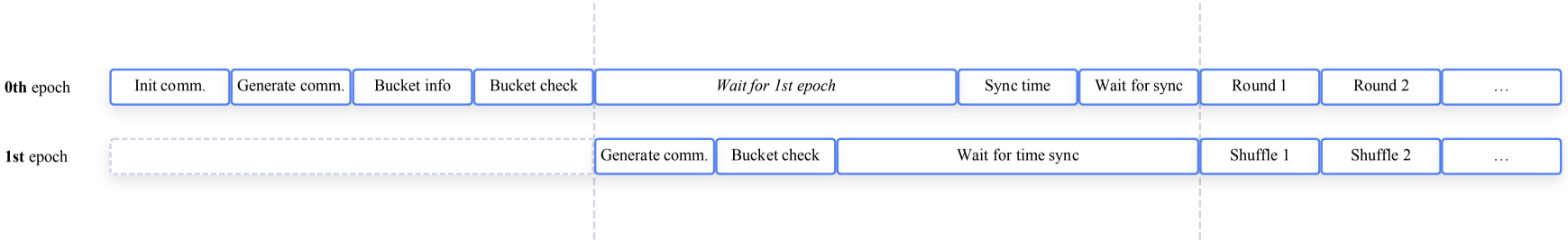



Figure 3.6: Visualization of initial protocol flow.

### 3.4.2 Ongoing protocol flow

After initialization, the protocol proceeds with three overlapping epochs, although only one epoch actively produces a block (see Fig. 3.7). The second epoch uses the data from blocks for shuffling and performs the shuffling. Before the epoch ends, a secure „cooldown“ period, during which no shuffling is allowed to prevent adversaries from predicting future ordering. However, this feature remains unimplemented in our protocol. The third epoch is in the commitment generation phase and then waits for the block generation epoch to finish. At the end of the currently running epoch, roles rotate. The previous shuffling epoch becomes the next block generation epoch, the commitment generation epoch transitions into the shuffling phase, and a new epoch for new commitments is formed.

## 3.5 Node communications

Communication between nodes is ensured using the libp2p library, which relies on the multiaddresses format to encode multiple layers of addressing information into a single path (e.g., `/ip4/127.0.0.1/tcp/12345`). Each node obtains the multiaddresses of its peers from a bootstrapping service and opens bi-directional connections. Within the overlay network, nodes are uniquely identified by integer identifiers (i.e., node ID) of the location from the generated network. The node with the larger ID initiates the connection. For each peer, an incoming read stream buffer is managed by a separate goroutine on a separate thread. Individual messages within the buffer are delimited by newline characters. All messages are serialized in JSON or, if the field contains binary data, they are additionally encoded to hexadecimal or Base64 strings.

To simulate real-world network conditions despite nodes location location on the same physical host, each outgoing message incurs an artificial delay. Just before transmission, the responsible gourtine sleeps for a duration determined by the network topology's geolocation data. This latency is further increased by jitter to make latency more realistic. The network jitter is a variation in the delay of data packages sent over a network, causing irregular arrival times at the destination. The jitter is generated using exponential distribution where $\lambda = 1$ and the output value is equal to milliseconds. To optimize message flow, the sending node checks after the delay if it has already received the same message from the node that is going to receive the message. If yes, it does not send the message.

Whenever a node receives a message from a peer, it decodes it and checks it against its local history. Duplicate messages are discarded; otherwise, the message is appended to the node's history, gossiped to the other peers, and processed based on the type of the message

### 3.5.1 Communication with data-processing service

Nodes communicate with data-processing service using RabbitMQ. Node-specific events such as commitment generation, bucket hash check, and block production are published to the RabbitMQ's exchange „`blockchain_data`“. The data-processing service consumes these events for downstream analysis. Conversely, the data-processing service publishes node commands to nodes with the correct routing key to consume. The routing key is the node ID.

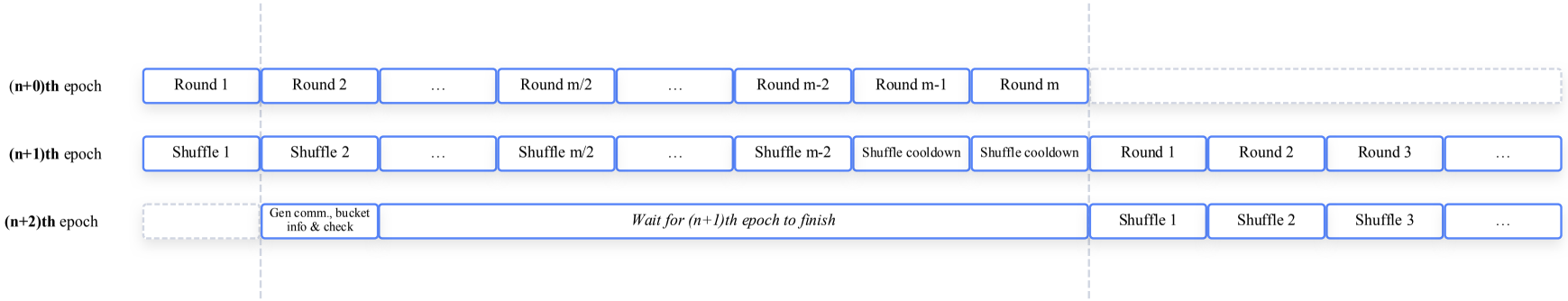



Figure 3.7: Visualization of protocol flow for the running epochs.

# Chapter 4

# Implementation

This chapter presents the architecture tailored for this work and individual services. It also details the network latency dataset and network model used for topology generation.

## 4.1 Architecture

The implementation is ready for a Docker environment (see Fig. 4.1), enabling users to run it independently with significantly reduced complexity. The project only requires Python, Docker[20], and Docker Compose to run. All services will be built in a container environment and integrated into Docker images. Python is required for the initialization script.

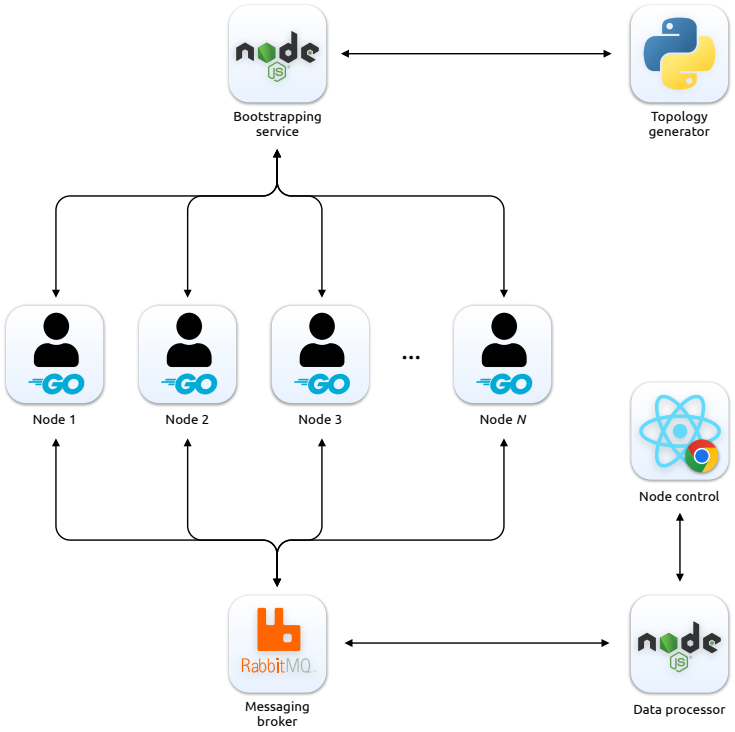



Figure 4.1: Simulation architecture. Individual components represent containers inside Docker environment.

### 4.1.1 Containerized Node Deployment

The idea of a containerized environment is to enable easier management of simultaneously running nodes in the same network with automated configuration using an initialization script. The isolated environment only exposes a web application and data processor services, which can control running nodes using commands sent through RabbitMQ over the internal network.

The implementation uses the required environment variable, which contains the path to the volume storage shared with other services. From this path, the mandatory configuration file is loaded. The configuration contains the setup parameters for the blockchain shared with other nodes and the connection port numbers for Bootstrapping and RabbitMQ services.

## 4.2 Network latency dataset

In order to make the simulation more realistic, we use a dataset containing data on connection latencies from 246 servers worldwide (see Fig. 4.2). Original data are publicly available from Wonderproxy company and contain ping times measured several times on July 19th and 20th, 2020[1], from each server to almost every other server. We calculated an average from all measurements for each location and stored this in a data structure that allows efficient access latencies from one server to all 245 other servers. Finally, data were incomplete, and less than 1% contained no value. We conducted manual measurements on December 31st, 2024, to fill these gaps. In summary, the processed dataset contains 246 server locations, each containing the location's name, geographical position, and latencies to the other 245 locations.

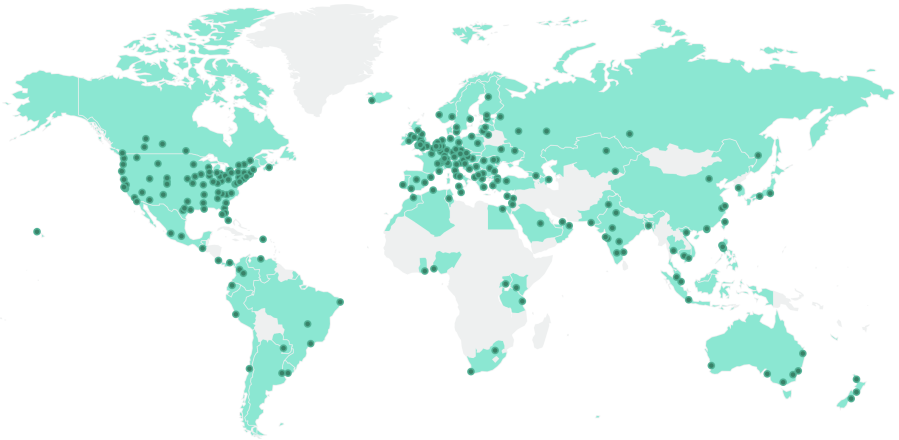


Figure 4.2: Visualization of Wondershare servers in 2025 with 260 servers.

The authors of the ping dataset do not provide the size of the ping message, but it can be estimated in tens of bytes, which is the typical size for such messages. Although the size does not fully correspond to the size of a block and may introduce some additional delay, we have considered the following methods to resolve it. We aim to minimize the size of the block through the use of the DAG structure. In addition, we also introduce an additional random delay at each node (i.e., jitter), which aims to include factors such as block size,

[1] https://wonderproxy.com/blog/a-day-in-the-life-of-the-internet/

network congestion, protocol overhead, and other potential sources of extra delay. In future scenarios where the protocol may require larger blocks, for instance, to operate with smart contracts, we leave this part for future work.

## 4.3 Network model

The network model can be described as a directed graph $G = (V, E, W)$. V is a set of vertices that represents peer-to-peer network nodes. $E$ is a set of edges and stands for the connection between two nodes, and $W$ is a weighted function that represents connection latency. In our work, every node corresponds to a specific location, and the latency value is based on measured ping data (see the Section 4.2). $V$, $E$, and $W$ are further defined as follows:

- $V = \{v_1, v_2, v_3, ..., v_n\}$, where $v_i$ denotes a single node and $n$ denotes the number of nodes in the network.
- $E \subseteq \{\{v_i, v_j\} | v_i, v_j \in V \wedge v_i \neq v_j\}$
- $W : E \to R^+$ is a weighted function that assigns a positive real number $w_{ij}$ to each edge ($v_i$, $v_j$).

Note that real-world communication in the peer-to-peer network between nodes may not be symmetric (e.g., node $v_i$ sends data to $v_j$, but $v_j$ uses its own set that does not include $v_i$ to send data). For simplification, our network models in a directed graph only include edges that are directed on both sides. Thus, the graph has a symmetric property.

## 4.4 Topology generation service

The topology generation service is implemented in Python, and it is based on Project practice 1 [32]. It aims to generate a suitable peer-to-peer network for blockchain consensual protocol simulation. Variables that affect the final peer-to-peer network topology model include the final number of nodes, node degree (i.e., number of peers for a node), and number of hops (i.e., graph diameter in graph theory). The number of hops is highly affected by how well nodes inside the network are connected. Optimally, the peer-to-peer network should have the least average number of hops, enabling the network to communicate more effectively as messages from one node to the whole network are distributed faster. Existing graphs such as de Bruijn, Trie, Chord, CAN, and Butterfly offer various graph diameters. Among these, the most suitable solution is the de Bruijn graph due to its minimal graph diameter, which can be calculated as $logN(k)$ [47]. This graph diameter corresponds to the theoretical minimum for a peer-to-peer network in an optimal scenario, meaning the actual value of the average number of hops will converge to the node degree of the de Bruijn graph.

The results in Project Pracice [32] show that for a network of size 8,000 nodes with a node degree of mean 15.2121, applying the de Bruijn property of $logN(k)$, the average number of hops corresponds to 3.301. The result for the generated network is equal to 3.545, which is comparable to the suboptimal solution.

### 4.4.1 Implementation

The topology generation service requires three inputs to generate a network topology. Firstly, the data for network propagation delay. Original work PP1 used a distribution

based on data gathered from the Bitcoin network. In this work, we are not focusing on the Bitcoin network, and we used connection latency data from 246 real servers deployed worldwide.

The second input is node degree. This input was also produced from the Bitcoin network in the original work. However, this setting is more general and affects not only Bitcoin specifically. For this reason, we kept this original distribution created from data published in [32].

The third input contains a specific number of nodes, which must be specified in the configuration file before starting the simulation environment.

### 4.4.2 Algorithm

The algorithm for network topology generation can be divided into three parts:

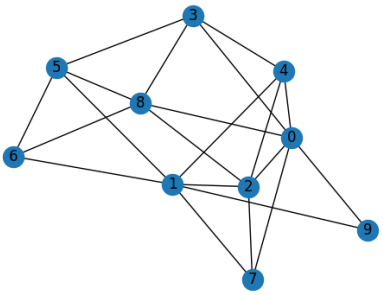
1. **Initialization.** The algorithm starts by loading input data. Specifically, the total node count is set, and the generator for node degree is initialized. Notable variables `END_NODES_TEST` and `PATH_NODES_TEST` are also initialized and will be used in part three.

2. **Nodes loop.** The algorithm add nodes into the graph structure and set their expected number of edges. The actual number of edges may change during the algorithm.

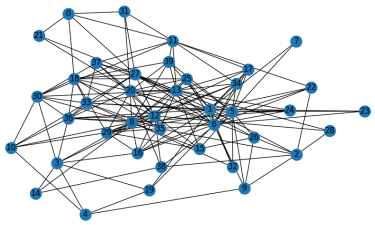
3. **Edges loop.** This part contains multiple nested loops, starting with the outermost loop, which iterates over all nodes in the network. This loop initializes the counters for the number of successfully established edges and attempted edge insertions to zero. The core of the algorithm starts by looping through the expected edges. Consider starting node $S$ (the current node determined by the iteration of the outermost loop). For this node, we randomly select another node $E_i$ as the random ending node (where i denotes the index in the sequence of ending nodes to be tested). We choose two testing nodes for these nodes, $T_1$ and $T_2$, where $T_1 \neq T_2$ and $T_1$ and $T_2$ are different nodes from $S$ and $E_i$. For this pair $(T_1, T_2)$, we calculate the number of hops before and after the path between $S$ and $E_i$ was created. This is done by creating a copy of the current network graph and finding the shortest path using the Dijkstra algorithm. Then we choose another pair of nodes, $T_1$ and $T_2$, and repeat this process several times, limited by the `PATH_NODES_TEST` variable, which is defined as half of total number of nodes in the network in our implementation. From all pairs $(T_1, T_2)$ tests, we compute the average change in the shortest paths across all test pairs, determining the $E_i$ score. We repeat this process for another ending node $E_{i+1}$, where the number of different ending nodes is limited by `END_NODES_TEST`, defined as the total number of nodes in the network minus one. As a result, larger networks with more nodes require testing more potential ending nodes.

   Once all candidate ending nodes have been scored, we sort the list of score results from best to worst and randomly pick an ending node from the top half of the sorted list. By preferring the ending node with the best score, the result would aim for the idealistic approach. Our approach is to generate a suboptimal network that is slightly worse than ideal [32].

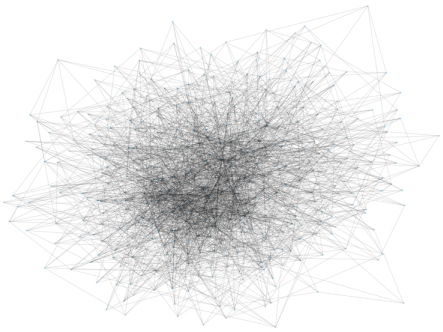
The results produced by our algorithm are shown in Fig. 4.3 and in Table 4.1.



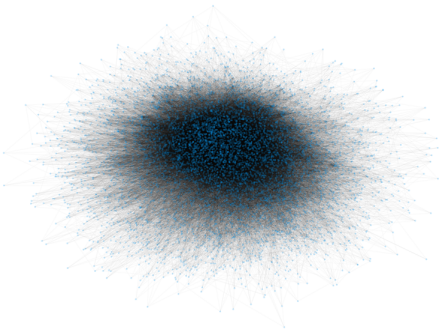

(a) Visualization of generated network with 10 nodes.



(b) Visualization of generated network with 40 nodes.



(c) Visualization of generated network with 400 nodes.



(d) Visualization of generated network with 8000 nodes.

Figure 4.3: Visualizations of generated network using modified version of topology generator.

Table 4.1: Comparison of various network sizes tested on modified version of topology generator used in our work. First metric, number of hops between two compares different values with the ideal value achieved in the de Bruijn graph. The $\tau$ metrics quantify the shortest path duration between two nodes based on the latency distribution. The „-“ denotes missing latency values for networks larger than 246 nodes, the current limit of the network topology generator.

| | 10 | 40 | 400 | 8000 |
|---|---|---|---|---|
| Ideal num. of hops $logN(k)$ | 0.846 | 1.355 | 2.201 | 3.301 |
| Mean num. of hop | 1.484 | 1.781 | 2.566 | 3.545 |
| Mean $\tau$ (seconds) | 0.252 | 0.311 | - | - |
| Min. num. of hops | 1 | 1 | 1 | 1 |
| Min. $\tau$ (seconds) | 0.001 | 0.01 | - | - |
| Max. num. of hops | 2 | 3 | 4 | 5 |
| Max. $\tau$ (seconds) | 0.617 | 0.959 | - | - |
| Std. deviation of num. of hops | 0.494 | 0.498 | 0.593 | 0.622 |
| Std. deviation of $\tau$ (seconds) | 0.107 | 0.157 | - | - |

### 4.4.3 Modifications

The topology generation modifications introduced in our work include:

- Usage of real-world data for latency.
- Generation of additional output - a unique view for each individual node.
- Support for small networks by adding a particular condition.

## 4.5 Bootstrapping service

The bootstrapping service is implemented in JavaScript and uses the WebSocket[2] library to communicate with nodes. The service starts by reading the volume path environment and parsing the configuration file. It requires reading one variable from the configuration — the number of nodes. After this, the WebSocket server starts and waits for nodes to connect. Nodes start to connect by sending a hello message with their multiaddress. The Websocket server responds with a hello message in return and waits until all nodes send the hello message.

After all nodes have established a connection to the WebSocket server and sent a hello message, the server attempts to read the generated network file. At this point, the generated network file might still be in the generation process by the topology service. For this reason, WebSocket waits until the network configuration is ready before proceeding.

Once the configuration is ready, it retrieves all peer information and randomly assigns an identity to each node. After that, every node will be sent information about itself and its peers. Each node needs to confirm this message. After confirmation, the bootstrapping

[2] `https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`

is considered finished. The bootstrapping service schedules its shutdown, and nodes begin operating independently by establishing connections with their peers.

## 4.6 Node implementation

This Section describes implementation details about the node implementation. Further, we show the file structure.

### 4.6.1 Programming Language Selection

Implementing a DAG-based PoS blockchain protocol that integrates zero-knowledge proofs at the consensus layer poses high constraints on programming language selection.

Firstly, zero-knowledge proof systems rely on finite-field arithmetic and elliptic-curve operations, which demand efficient implementation. Zk-SNARK proofs are designed in a way that the generation of proof takes the majority of the time, so that the verification can be done fast. In our protocol design, we are limited by the time required to generate one or more commitments. Commitments are generated and shuffled in parallel to block generation at the same time intervals. Because the DAG structure allows multiple active branches, it demands many more commitments to select from than it would from one branch. For this reason, commitment generation needs to be fast and implemented efficiently.

Secondly, the consensus mechanism in the DAG structure requires concurrent processing of a high volume of transactions, validating blocks, and connecting them with related commitments within a short latency span. Low-level languages enable better utilization of hardware resources (e.g., caches, vector instructions) for resource-needy data structures such as mempool or blockchain.

In summary, we required a programming language with sufficient performance characteristics so that blockchains can handle a lot of processed data and a mempool of unprocessed but validated data. Every full node must be able to generate and validate transactions, commitments, cryptographic proofs, and blocks quickly. Low-level programming languages such as C, C++, Go, and Rust offer faster and more efficient execution than higher-level languages such as JavaScript or Python. Additionally, node implementation should only use the required amount of resources without requiring unnecessary extra overhead. This way, more users can run full nodes and participate in block generation, which benefits decentralization of the protocol.

Gnark library supports different cryptography constructs required in our ZKP application. That is the generation and verification of EdDSA signatures and Merkle proof hash operations. Gnark is implemented in Go, a low-level programming language that also provides optimized and easy-to-use constructs for concurrency programming (e.g., goroutines, channels). For these reasons, we chose Go as a programming language for node implementation and Gnark as a library for ZKP.

### 4.6.2 Source File Structure

The diagram (see Fig. 4.4) presents a high-level file-oriented view of the Go source files comprising the node implementation for the PoS DAG-based blockchain protocol. Since Go is not an object-oriented language and does not contain classes, we have adopted a file-level abstraction.

Individual parts of the implementation are described as follows:

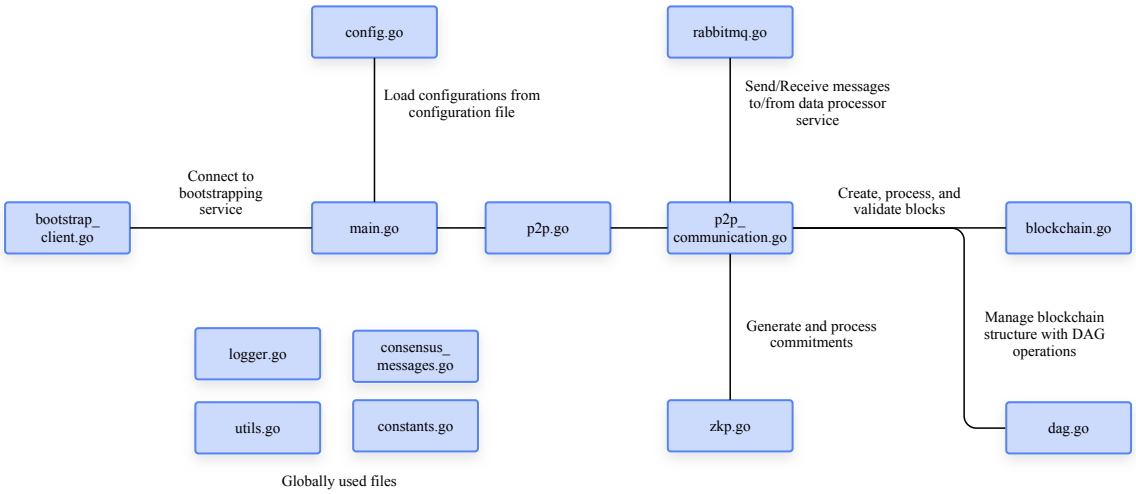



Figure 4.4: Visual representation of source files in node implementation and their relationship.

### `main.go`

The entry point of the program. It initializes the configuration and prepares the zero-knowledge proof keys, followed by the preparation of the peer-to-peer client. Once initialization is complete, the node contacts the bootstrap server and initializes the RabbitMQ connection. After these steps, the node turns into the running state and enters the infinite loop using `select{}`, a blocking statement with no case clauses that keeps the program running.

### `config.go`

Manages configuration of the node. It first reads environment variables that contain volume path, bootstrap server hostname, and RabbitMQ hostname. The node configuration is stored in the volume path and includes the following fields:

- Number of nodes in the network.
- Number of seconds reserved for one round.
- Maximum number of parallel chains in DAG structure (must be a power of two).
- Number of rounds per epoch.
- Maximum (i.e., desired) number of transactions per block.
- Minimum number of transactions in a single block to split.
- Maximum number of transactions in a single block to merge (requires both blocks to be eligible to merge to satisfy this condition).
- Random key used for commitment shuffling.
- Number of entries for the Merkle tree of public keys and its depth. These values are not used because depth is a constant for the arithmetic circuit, and this value must be set during compilation.

- Minimum number of commitments per node.
- Maximum number of commitments per node.
- Number of epochs to simulate.
- Bootstrapping server port.
- RabbitMQ service port.

#### `bootstrap_client.go`

Communicates with bootstrapping using WebSocket. The communication works in several phases during which nodes synchronize. Bootstrapping service is available from the start of the simulation and needs to wait for all nodes to initialize. Every node publishes its peer-to-peer address, available to negotiate connections from other nodes. It receives identification in the network (i.e., unique ID and city) and a set of peers. Every peer entry contains peer-to-peer multiaddress, ID, city, and latency.

#### `p2p.go`

Sets up communication with peers and initializes variables used further in peer-to-peer communication. During initialization, it generates public and private keys used in peer-to-peer communication. This pair of keys differs from the one used in ZKP, where the public key is used for node identification on the consensus layer.

Once the node retrieves its peers and closes the connection with the bootstrapping server, it opens connections to its peers. If the node's ID is greater than the peer's ID, then the node initiates a connection and creates a stream. For a node where its ID is lower, it connects to the stream. Both sides establish read-write connections and store the correct latency to the appropriate stream.

#### `p2p-communication.go`

As the biggest file, process the incoming messages from nodes, broadcast newly created messages, gossip received messages, handle block and commitment generation schedule, and delegates more work with complex tasks (such as ZKP, DAG, blockchain, RabbitMQ) to functions in other files. When the node receives the message, it is first processed by unmarshalling the JSON format. Then, the unique message ID is extracted and stored within the map structure, where the message ID is the key and the value is a node that sent this message. The receiving node then checks if it has already received this message earlier. If yes, it is discarded. Otherwise, it is gossiped to the node's peers that did not share this message with this node. Gossiping is executed as a new goroutine on a separate thread and starts by thread sleep for the time of latency to the peer plus random jitter. When sleep ends, the node again checks if the peer has already received and sent this message during sleep time to prevent sending a message that will be instantly discarded.

The processing of the message is split into several branches based on the message type. When a node is about to create a new message, it generates a new ID that consists of a cryptographically secure random integer and a timestamp to avoid collisions.

Nodes also send RabbitMQ transmissions for notable messages as part of node communication logging.

`blockchain.go`

Manages block processing, validation, and creation. Every block comprises the following fields:

- Hash, which consists of the block order number, Merkle root hash, the block author's public key, and creation timestamp.
- Block order number in the author's blockchain. This number does not impact ordering in the DAG structure and serves mainly for debugging purposes of a single node.
- Creation timestamp.
- Set of processed transactions.
- Merkle root hash of transactions.
- The author's public key.
- Hash of the parent block.
- Hash of the second parent, which is populated only after two branches merge into one.
- Row, column, and depth in the DAG structure.
- Corresponding commitment hash, secret $b$, and secret signature $sigSec$.

When a node receives the block, it performs the following actions:

- Decodes the transaction hash and validates its format. In a real-world scenario, the node would additionally check for sufficient balance and double-spending transactions from the same recipient. In our implementation, this is abstracted.
- Compute the Merkle root from transactions and compare it with the enclosed Merkle root hash.
- Check the valid author's public key format.
- Compute message hash $msg$ from $msgPub$ and $b$.
- Verify EdDSA signature.

If all conditions are met, the node determines the block as valid and stores it inside its local blockchain. It also computes object references for the parent block or blocks for faster access.

Once the block creation function is invoked, the node has already checked and prepared everything necessary for block production. This includes checking if a block can be created at the specific slot, determining its row, column, and depth in the DAG structure, preparing transactions that will be included in this block, and attaching the commitment to the block's slot. During block creation, the node creates an extra transaction that contains the block reward for block production and computes the Merkle root. Moreover, the block hash is calculated, and other necessary fields are attached to the block. The created block is then appended to the local blockchain and shared with the network.

### `dag.go`

Contains functions related to blockchain DAG structure. Once a set of transactions is received, the transaction hash mask is calculated and used to determine the correct branch where the transaction should be included. It also includes functions to determine the correct indices after split and merge actions. Furthermore, it contains a `decideAction` function that takes row and column as arguments for the position of the upcoming slot and produces a decision on the output with linked parent blocks. Based on previous blocks, the output action for the slot can be split, merge, continue, or skip. Continue action denotes continuation in the current branch, without split or merge, and skip denotes that the upcoming action will be empty. This function is called at the start of the round for every slot of the current column. If an action to split was produced, we must distinguish between the first and second generated blocks. The logic handling this specific is executed after deciding the action, during block production, where the block from the split at the bottom position is labeled as `SPLIT_SND`.

### `zkp.go`

Handles the ZKP logic. In the beginning, it reads proving and verifying keys that are pre-generated outside of the implementation for the ZKP circuit. The pair of asymmetric keys is generated using the Twisted Edwards BLS12-381 curve. The public key is used as node identification on the consensus layer, which a node uses to lock its stake and become eligible for block production. This key is included in a generated block and in the Merkle tree of public keys. Furthermore, it is also used in ZKP proof to verify EdDSA signatures.

The private key is used to produce EdDSA signatures and is never shared. Produced commitments contain ZKP proof and public witness (i.e., values for public variables used in the circuit).

Furthermore, this file also handles detailed validation of commitment with the following actions:

- Validate JSON format for the commitment message.
- Validate the content separator between the Base64 of the public witness and the Base64 of the proof.
- Validate the Base64 format for public witness and proof.
- Validate the content of the public witness (every variable should be aligned to 64 bits).
- Validate separators in $pubMsg$ (i.e., the „_“ and „#“).
- Validate protocol version, epoch number, and variant number.
- Validate ZKP proof.

### `rabbitmq.go`

Manages connection to RabbitMQ service using the go-rabbitmq library. It consumes messages with the routing key of the node's network ID. The messages include starting blockchain, receiving transactions, sharing public key, and receiving other public keys to build the public keys Merkle tree. The node also produces messages for RabbitMQ, such as generated or received blocks, generated or received commitments, and bucket information.

`logger.go`

Contains functions for improved logging output, which includes different colors for different logging types (e.g., debug, info, error) and additional time information for each logging message.

`consensus-messages.go`

Defines structures, types, and constants used in peer-to-peer and RabbitMQ communications.

`constants.go`

Defines constants used in the protocol and declares variables initialized from the configuration file.

`utils.go`

Includes additional functions used in node implementation.

## 4.7 RabbitMQ service

RabbitMQ is an open-source messaging broker that enables asynchronous communication between services using AMQP (Advanced Message Queuing Protocol). It acts as a middleman that receives and routes messages between producers (senders) and consumers (receivers).

RabbitMQ is well-suited for a scalable environment and allows features such as message acknowledgments, delivery guarantees, and flexible routing logic using exchanges and bindings.

Producers send messages to exchanges, which route these messages to queues based on defined routing rules. Consumers subscribe to queues and receive delivered messages [61].

Our work uses RabbitMQ as an intermediary to forward logging messages from the nodes to the data processor service and to deliver control messages from the data processor service to individual nodes.

## 4.8 Data processor service

The data processor is a backend service implemented in TypeScript that uses the framework NestJS[3]. It collects data from nodes, stores results in a CSV file, receives commands from the node control service, and forwards them to nodes using RabbitMQ.

The CSV file contains information about blockchain. Specifically, each row contains one that was produced and contains its author, position in DAG structure, number of transactions, hash, order number, merkle root, previous block hashes, and commitment's hash and secret. In the future, a database can extend this service to store collected data more efficiently.

[3] `https://nestjs.com/`

## 4.9 Node control service

The node control service is a frontend service implemented in the React[4] framework, allowing access to an HTML website (see Fig. 4.5) and sending specific messages to individual nodes. The messages available to send are the following:

- Get a number of ready nodes. Once a node establishes a connection with all its peers, it sends the message that it is ready.
- Initiate nodes to share their public key and build a Merkle tree of public keys.
- Share the genesis block and start the blockchain protocol.
- Send transaction pack — a number of transactions specified by the user, with randomly generated hashes.
- Stop the blockchain protocol.

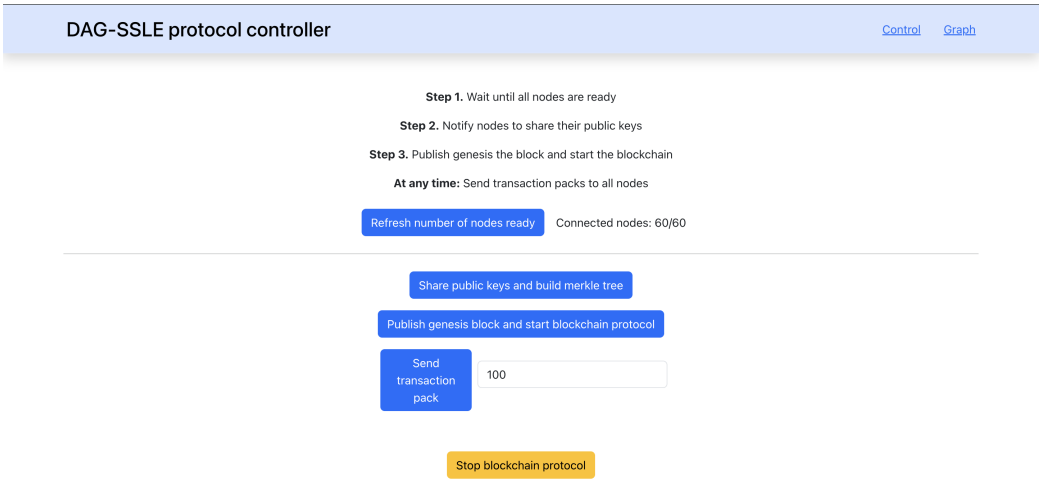



Figure 4.5: Node control service web page with „Control“ tab selected. The website allows sending commands to the data processor that are further passed to RabbitMQ and eventually reach the target nodes.

The website also shows the interactive network visualization in the Graph tab (see Fig. 4.6) if the network topology has already been generated.

## 4.10 Initialization process

The whole implementation of all services requires several commands to be executed in the correct order. We created an initialization script that does everything in the background to simplify the process. The script is by default executed in interactive mode, which prompts the user regarding setup configuration (e.g., number of nodes in the network, number of rounds per epoch, required time for a single round). The prompts also contain a valid range of values, and the default answer contains a value used during the last script execution.

[4] `https://react.dev/`

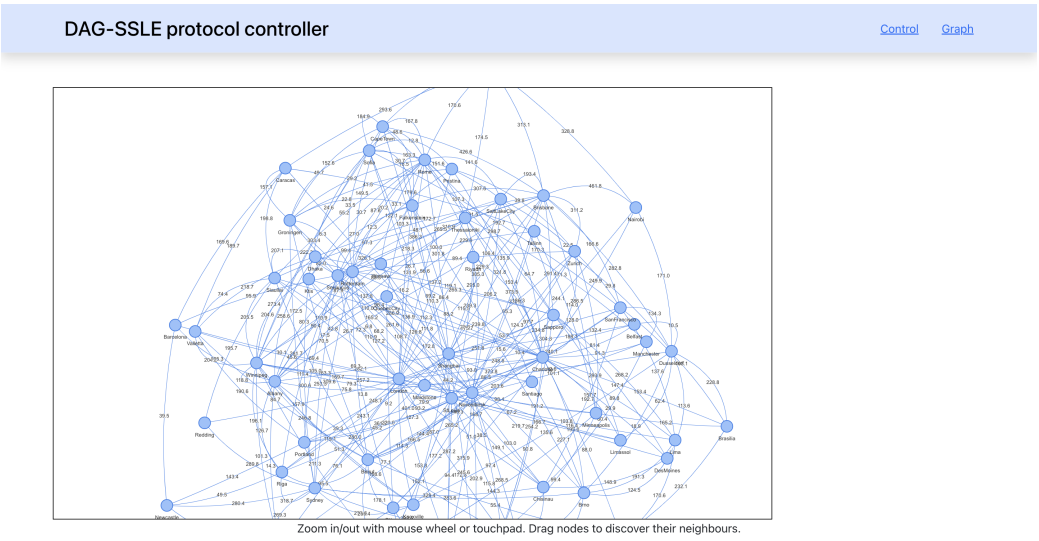



Figure 4.6: Node control service web page with „Graph“ tab selected. The visualized network is identical to the one generated from topology generator.

The one specific prompt is to enter a path in which a `volume` directory will be created. This path will be used to create a folder in the format `volume_DATETIME` which will serve as a shared directory within deployed services. This folder is mounted inside running containers as a volume, and the content is the same as on the hosting machine (not duplicated). This folder will contain the nodes configuration file, proving and verifying keys, and additional files generated by services (i.e., network topology and CSV data file).

The script also allows to be executed in non-interactive mode, which sets all parameters to the values from the last execution.

After all parameters are set, either interactive or non-interactive, the script creates the volume directory, copies ZKP-related keys, and builds Docker images for all services. Because Docker is using caching for building images, this process only needs to be executed once unless the source files of individual services have changed. Then, a Docker Compose YAML file is created, which contains all mandatory services plus individual nodes based on the number of nodes specified for the configuration. Finally, it saves this configuration to remember parameters for subsequent execution, stops existing services related to the project, if there are any, and starts the services with the newly created setup.

# Chapter 5

# Evaluation

This chapter includes two experiments with our implementation of the SSLE-DAG consensus protocol.

Experiments were executed on CPU Intel Xeon Gold 6548N, x86_64 architecture with 60 cores and 60 threads while using 120 GB of RAM. The machine used was hosted on Digital Ocean[1], a cloud service provider.

## 5.1 Experiment with 60 nodes

The first experiment was conducted on a network of 60 nodes to evaluate the protocol's functionality, resilience, fairness, and overall throughput. The parameters used for this experiment are displayed in Table 5.1.

Table 5.1: Parameters used in the experiment with 60 nodes. The maximum number of transactions per block states a maximum number of transactions produced by a blockchain user, plus one transaction as a block reward, added by the author.

| Parameter | Value |
|---|---|
| Number of nodes | 60 |
| Time allocated per round | 8 seconds |
| Maximum number of parallel chains (DAG) | 8 |
| Number of rounds per epoch | 20 |
| Maximum number of transactions per block | 1000+1 |
| Minimum number of transactions in a single block to split | 800 |
| Maximum number of transactions in a single block to split | 275 |
| Minimum number of commitments per node | 1 |
| Maximum number of commitments per node | 3 |
| Number of entries in Merkle tree of public keys | 512 |
| Depth of Merkle tree of public keys | 9 |

All nodes started with 1,000,000 pre-generated transactions, providing a sustained load to process. Immediately after start, blocks began to fill and split, eventually reaching the maximum number of chains in the DAG structure (see Fig. 5.2).

[1] `https://www.digitalocean.com/`

Commitment utilization was 88.89%, meaning that while all parallel chains were active, approximately 11.11% of commitments were discarded. Although the DAG structure was operating at all chains, block production remained probabilistic for individual nodes, resulting in slight variations in block rewards among participating nodes. However, the difference from the mean value remained minor (see Fig. 5.1).

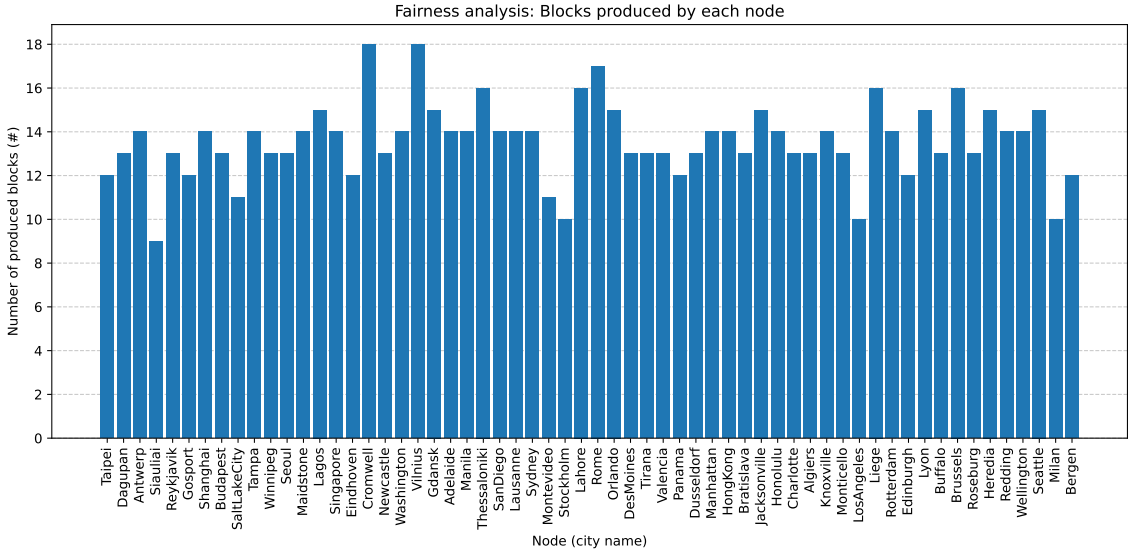



Figure 5.1: Number of blocks produced by individual nodes during the experiment with 60 nodes. Nodes are identified by a city, instead of the public key.

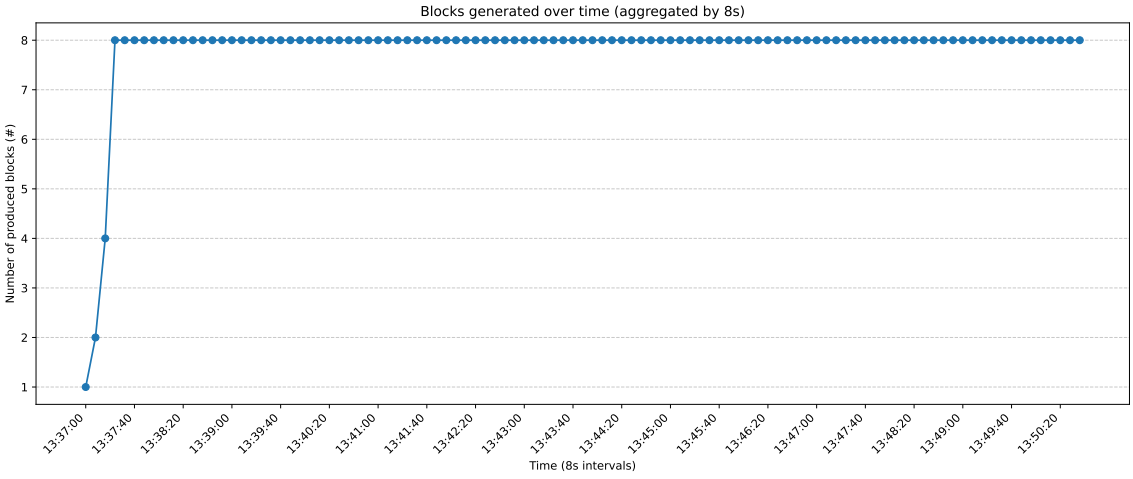



Figure 5.2: Number of generated blocks during the experiment with 60 nodes aggregated in 8-second intervals.

During the simulation, nodes generated 815 blocks across 104 consecutive rounds (each round lasting 8 seconds), which took 13 minutes and 44 seconds (simulation time is equal to the real time). During this period, nodes processed 815,815 transactions, resulting in 990.07 transactions per second (see Table 5.2).

The experiment demonstrated that, even under constant high demand and maximal DAG branching, the SSLE consensus protocol remains functionally correct within a network of 60 nodes communicating and coordinating independently, without any central coordinator.

Table 5.2: Results metrics of the experiment with 60 nodes.

| Metric | Value |
|---|---|
| Number of completed rounds | 104 |
| Number of completed blocks | 815 |
| Minimum blocks produced per node | 9 |
| Maximum blocks produced per node | 18 |
| Mean blocks produced per node | 13.58 |
| Standard deviation of block production | 1.77 |
| Utilization of commitments | 88.89% |
| Mean interval of block generation | 1.01 seconds |
| Total number of processed transactions | 815,815 |
| Total time span of simulation | 13 minutes 44 seconds |
| Average transaction throughput (TPS) | 990.07 transactions/second |
| Peak transaction throughput (TPS) | 1001 transactions/second |

## 5.2 Experiment with 40 nodes

The second experiment involved fewer nodes but focused on higher block-production rates (see Table 5.3). The round duration was halved to four seconds, and the epoch length was doubled to 40. The simulation also ran longer, resulting in visible branch merges (see Fig. 5.4), and after processing all available transactions, it dropped to one branch.

The utilization of commitments has been lowered to 50%, which resulted in a negligible change in the fair amount of block production (see Fig. 5.3).

Overall, this configuration achieved a higher transaction throughput than in the experiment with 60 nodes (see Table 5.4). Even with only a four-second round interval, nodes were capable of sustaining the demand of 8 parallel chains.

## 5.3 Discussion

Although the implementation supports networks of up to 246 nodes, we performed our experiments on a smaller scale because when establishing a connection with more nodes, the communication becomes unstable. In our tests, a network of 99 nodes remained reliable. After exceeding this size, already established connections started to break, causing communications errors. This limitation is on the operating system's network stack, so scaling past 99 nodes requires further adjustments on the system level. In addition, networks operating with more than 60 nodes rarely started showing connection instability. For this reason, we decided to conduct experiments with 60 and 40 nodes to ensure stable results.

Furthermore, our simulated throughput may exceed what a real-world implementation would achieve. By abstracting validation of sufficient balance and double-spend validation,

Table 5.3: Parameters used in the experiment with 40 nodes. The maximum number of transactions per block states a maximum number of transactions produced by a blockchain user, plus one transaction as a block reward, added by the author.

| Parameter | Value |
|---|---|
| Number of nodes | 40 |
| Time allocated per round | 4 seconds |
| Maximum number of parallel chains (DAG) | 8 |
| Number of rounds per epoch | 40 |
| Maximum number of transactions per block | 1000+1 |
| Minimum number of transactions in a single block to split | 800 |
| Maximum number of transactions in a single block to split | 275 |
| Minimum number of commitments per node | 1 |
| Maximum number of commitments per node | 16 |
| Number of entries in Merkle tree of public keys | 512 |
| Depth of Merkle tree of public keys | 9 |

the implementation becomes faster, so the difference between ten and a thousand transactions per block is minimal. On the other side, our block data structure contains the majority of required validation, including the previous block hash check and the Merkle root hash validation. The block processing amount plays a critical role in configuration because a block production rate that is too high could result in missing commitments prepared for the next epoch.

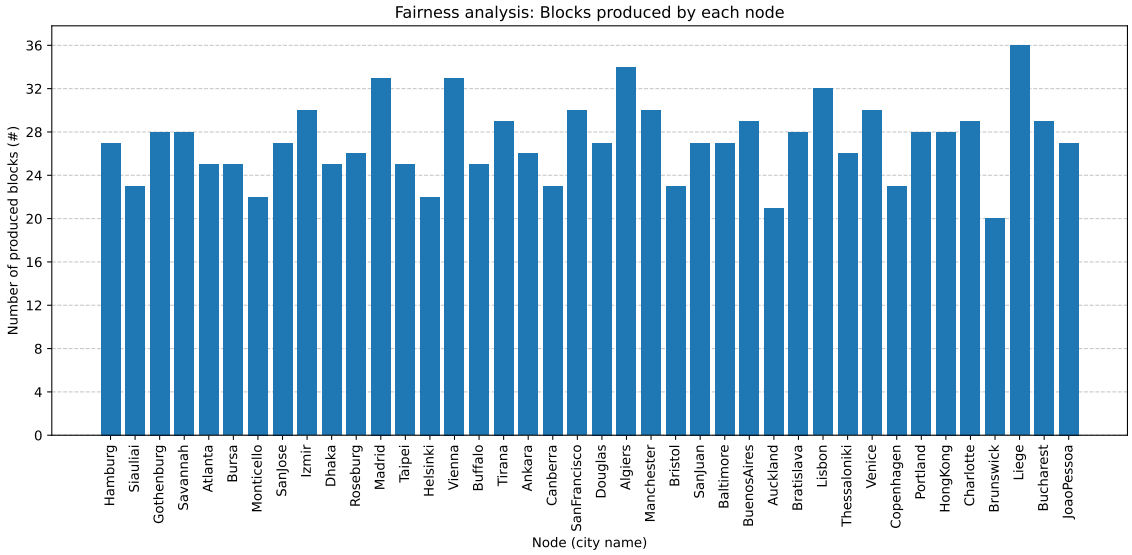



Figure 5.3: Number of blocks produced by individual nodes during the experiment with 40 nodes. Nodes are identified by a city, instead of the public key.

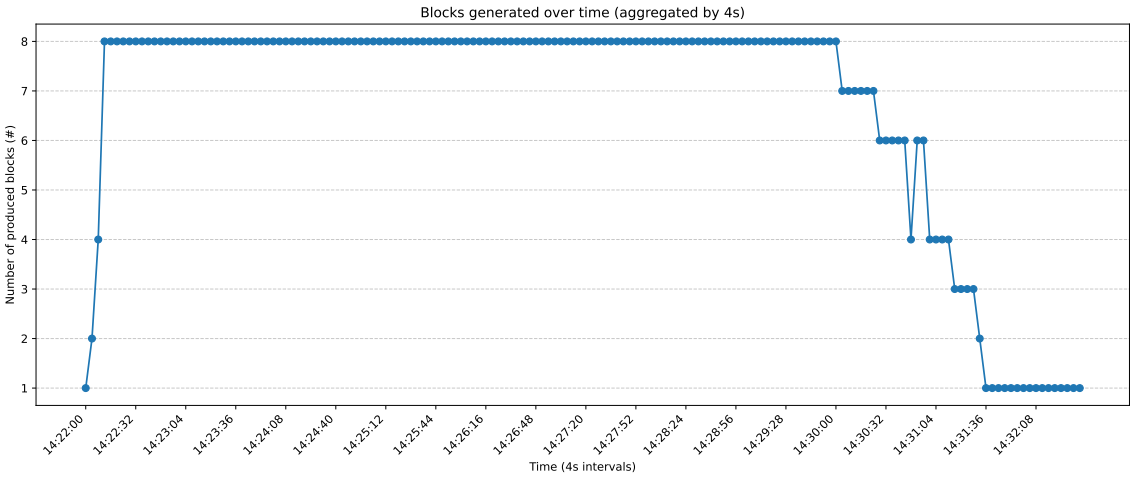



Figure 5.4: Number of generated blocks during the experiment with 40 nodes aggregated in 4-second intervals.

Table 5.4: Results metrics of the experiment with 40 nodes.

| Metric | Value |
|---|---|
| Number of completed rounds | 161 |
| Number of completed blocks | 1,086 |
| Minimum blocks produced per node | 20 |
| Maximum blocks produced per node | 36 |
| Mean blocks produced per node | 27.15 |
| Standard deviation of block production | 3.60 |
| Utilization of commitments | 50% |
| Mean interval of block generation | 0.59 seconds |
| Total number of processed transactions | 1,024,403 |
| Total time span of simulation | 10 minutes 40 seconds |
| Average transaction throughput (TPS) | 1600.63 transactions/second |
| Peak transaction throughput (TPS) | 2002 transactions/second |

# Chapter 6

# Conclusion

In this work, we studied existing consensus protocols, comparing the strengths and weaknesses of PoW and PoS approaches, and narrowed our focus to PoS-based protocols. We examined the existing DoS vulnerabilities in the PoS consensus layer, specifically for Ethereum. We reviewed existing cryptographic schemes — VRF, HE, ZKPs that are used to mitigate the DoS vulnerability in Polkadot and Algorand in the SLE application.

We further adopted the SSLE paradigm, a strict form of SLE, which is currently being researched and developed for Ethereum and aims to ensure only a single selection of leader per election. Based on these ideas, we designed our commitment scheme based on ZKPs (specifically zk-SNARKS) and integrated it into our consensus protocol, which implements network, consensus, and data layers in detail.

For the network layer, we leveraged real-world latency measurements from 246 geolocated servers. These modified data are used in our topology generator script to generate a realistic, suboptimal peer-to-peer network with the desired number of nodes and peers.

We also focused on DAG-based blockchains, specifically on the research-based protocol Sycomore, which served as a baseline for our version of a flexible DAG-based blockchain structure that utilizes multiple chains under high demand and reduces the number of chains when the number of transactions to process is low. This structure is used in our work instead of the traditional, one-chain structure.

To establish a connection between nodes and assign their identities with their peers, we developed a bootstrapping service that takes the generated network topology with a desired number of nodes, connects to nodes using WebSocket, and delivers each of them their identity and their peers. Nodes are then capable of communication, independent of any controlling element.

To monitor and control the running simulation, we implemented a web interface that communicates with the backend service directly connected to nodes through RabbitMQ, a messaging broker service.

To simplify the installation and deployment process, we also developed an initialization script in Python, the only file the user executes to set up and start the consensus protocol. The script also allows for entering interactive mode, which hints at the required values for a valid configuration and thus enables starting the protocol simulation with minimal knowledge of the underlying system. The script reduces the installation hurdles by building all services inside a Docker environment, which are then containerized.

We also deployed two experiments to test our proposed protocol's correctness, resiliency, and transaction throughput. The experiments were conducted on networks of 60 and 40 nodes. The results showed fair distribution of block rewards among participants and av-

erage throughput of 990.07 TPS for the first experiment and 1,600.63 TPS for the second experiment. The second experiment also demonstrated that the protocol deployed with 40 nodes can process eight parallel chains, where a single round lasts only four seconds.

Our implementation can be further extended by modifications in the operating system to allow stable simulation of more than 100 nodes. Furthermore, it can be expanded for deployment on multiple physical machines to simulate even bigger networks. Finally, the implementation lays the groundwork for a production-grade blockchain client.

# Bibliography


[1] Acar, A.; Aksu, H.; Uluagac, A. S. and Conti, M. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery, july 2018, vol. 51, no. 4. ISSN 0360-0300. Available at: https://doi.org/10.1145/3214303.

[2] Albrecht, M.; Grassi, L.; Rechberger, C.; Roy, A. and Tiessen, T. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity* Cryptology ePrint Archive, Paper 2016/492. 2016. Available at: https://eprint.iacr.org/2016/492.

[3] Algorand. *Algorand consensus* online. 2021. Available at: https://developer.algorand.org/docs/get-details/algorand_consensus/. [cit. 2025-05-15].

[4] Alpturer, K. and Weinberg, S. M. Optimal RANDAO Manipulation in Ethereum. In: Böhme, R. and Kiffer, L., ed. *6th Conference on Advances in Financial Technologies (AFT 2024)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, vol. 316, p. 10:1–10:21. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-345-4. Available at: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2024.10.

[5] Anceaume, E.; Guellier, A.; Ludinard, R. and Sericola, B. Sycomore: A permissionless distributed ledger that self-adapts to transactions demand. In: IEEE. *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018, p. 1–8.

[6] Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K. et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In: *Proceedings of the Thirteenth EuroSys Conference*. New York, NY, USA: Association for Computing Machinery, 2018. EuroSys '18. ISBN 9781450355841. Available at: https://doi.org/10.1145/3190508.3190538.

[7] Bellés Muñoz, M.; Whitehat, B.; Baylina, J.; Daza, V. and Muñoz Tapia, J. L. Twisted edwards elliptic curves for zero-knowledge circuits. *Mathematics*. MDPI, 2021, vol. 9, no. 23, p. 3022.

[8] Bellore, A. *Birthday Attacks, Collisions, And Password Strength* online. 2021. Available at: https://auth0.com/blog/birthday-attacks-collisions-and-password-strength/.

[9] Bellés Muñoz, M.; Isabel, M.; Muñoz Tapia, J. L.; Rubio, A. and Baylina, J. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing*, 2023, vol. 20, no. 6, p. 4733–4751.

[10] Ben Sasson, E.; Chiesa, A.; Garman, C.; Green, M.; Miers, I. et al. *Zerocash: Decentralized Anonymous Payments from Bitcoin* Cryptology ePrint Archive, Paper 2014/349. 2014. Available at: https://eprint.iacr.org/2014/349.

[11] Boneh, D.; Eskandarian, S.; Hanzlik, L. and Greco, N. *Single Secret Leader Election* Cryptology ePrint Archive, Paper 2020/025. 2020. Available at: https://eprint.iacr.org/2020/025.

[12] Botrel, G.; Piellard, T.; Housni, Y. E.; Kubjas, I. and Tabaie, A. *Consensys/gnark: v0.12.0*. Zenodo, 2024. Available at: https://doi.org/10.5281/zenodo.14728739.

[13] Brown, D. R. Sec 1: Elliptic curve cryptography. *Certicom Research*, 2009, vol. 2.

[14] Buterin, V. *Secret non-single leader election* online. 16. Jan 2022. Available at: https://ethresear.ch/t/secret-non-single-leader-election/11789. [cit. 2025-05-17]. Ethereum Research.

[15] Chen, J. and Micali, S. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*. Elsevier, 2019, vol. 777, p. 155–183.

[16] Christ, M.; Choi, K.; McKelvie, W.; Bonneau, J. and Malkin, T. Accountable Secret Leader Election. In: Böhme, R. and Kiffer, L., ed. *6th Conference on Advances in Financial Technologies (AFT 2024)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, vol. 316, p. 1:1–1:21. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-345-4. Available at: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2024.1.

[17] Churyumov, A. Byteball: A decentralized system for storage and transfer of value. online, 2016. Available at: https://byteball.org/Byteball.pdf.

[18] Cormen, T.; Leiserson, C.; Rivest, R. and Stein, C. *Introduction To Algorithms*. MIT Press, 2001. Mit Electrical Engineering and Computer Science. ISBN 9780262032933. Available at: https://books.google.sk/books?id=NLngYyWFl_YC.

[19] Digiconomist. *Bitcoin Energy Consumption Index* online. 2017. Available at: https://digiconomist.net/bitcoin-energy-consumption. [cit. 2025-05-09].

[20] Docker, I. Docker. *Lınea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[21] Dong, M. *Benchmarking ZKP Development Frameworks: the Pantheon of ZKP* online. 2023. Available at: https://ethresear.ch/t/benchmarking-zkp-development-frameworks-the-pantheon-of-zkp/14943. [cit. 2025-05-09].

[22] Douceur, J. R. The sybil attack. In: Springer. *International workshop on peer-to-peer systems*. 2002, p. 251–260.

[23] Edgington, B. *A technical handbook on Ethereum's move to proof of stake and beyond* online. 2025. Available at: https://eth2book.info/. [cit. 2025-05-14].

[24] Feng, S.; He, J. and Cheng, M. X. Security Analysis of Block Withholding Attacks in Blockchain. In: *ICC 2021 - IEEE International Conference on Communications*. 2021, p. 1–6.

[25] Freitas, L.; Tonkikh, A.; Bendoukha, A.-A.; Tucci Piergiovanni, S.; Sirdey, R. et al. Homomorphic sortition–single secret leader election for pos blockchains. *Cryptology ePrint Archive*, 2023.

[26] Goldreich, O. *Foundations of Cryptography: Volume 1*. USA: Cambridge University Press, 2006. ISBN 0521035368.

[27] Grandjean, D.; Heimbach, L. and Wattenhofer, R. Ethereum proof-of-stake consensus layer: Participation and decentralization. In: Springer. *International Conference on Financial Cryptography and Data Security*. 2024, p. 253–280.

[28] Groth, J. On the size of pairing-based non-interactive arguments. In: Springer. *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. 2016, p. 305–326.

[29] Guiso, L.; Sapienza, P. and Zingales, L. The role of social capital in financial development. *American economic review*. American Economic Association, 2004, vol. 94, no. 3, p. 526–556.

[30] Habib, G.; Sharma, S.; Ibrahim, S.; Ahmad, I.; Qureshi, S. et al. Blockchain technology: benefits, challenges, applications, and integration of blockchain technology with cloud computing. *Future Internet*. MDPI, 2022, vol. 14, no. 11, p. 341.

[31] Hladký, T. *Simulator for Verifying the Properties of DAG-Based Consensus Protocols*. Brno, CZ, 2022. Bachelor thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/24643/.

[32] Hladký, T. *Modeling the Bitcoin-like peer-to-peer network for the blockchain simulator*. Brno, CZ, 2023. Project practice. Brno University of Technology, Faculty of Information Technology.

[33] 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.

[34] Johnson, D.; Menezes, A. and Vanstone, S. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*. Springer, 2001, vol. 1, p. 36–63.

[35] Josefsson, S. and Liusvaara, I. *Edwards-Curve Digital Signature Algorithm (EdDSA)* RFC 8032. RFC Editor, january 2017. Available at: https://doi.org/10.17487/RFC8032.

[36] Kadianakis, G. *Whisk: A practical shuffle-based SSLE protocol for Ethereum* online. 2022. Available at: https://hackmd.io/@asn-d6/HyD3Yjp2Y. [cit. 2025-05-17].

[37] Katz, J. and Lindell, Y. *Introduction to Modern Cryptography, Second Edition.* 2ndth ed. Chapman & Hall/CRC, 2014. ISBN 1466570261.

[38] King, S. and Nadal, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *Self-published paper, August*, 2012, vol. 19, no. 1.

[39] Kumar, G. Denial of service attacks – an updated perspective. *Systems Science & Control Engineering*, october 2016, vol. 4, p. 285–294.

[40] Lamport, L.; Shostak, R. and Pease, M. The Byzantine generals problem. In: *Concurrency: the works of leslie lamport.* 2019, p. 203–226.

[41] Lantz, L. and Cawrey, D. *Mastering Blockchain.* O'Reilly, 2021. ISBN 9781492054702.

[42] Lee, C. *Litecoin* online. 2011. Available at: https://litecoin.org/. [cit. 2025-05-17].

[43] Li, L. Mitigating Challenges in Ethereum's Proof-of-Stake Consensus: Evaluating the Impact of EigenLayer and Lido, october 2024.

[44] Li, W.; Feng, C.; Zhang, L.; Xu, H.; Cao, B. et al. A Scalable Multi-Layer PBFT Consensus for Blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 2021, vol. 32, no. 5, p. 1146–1160.

[45] Lindell, Y. *Secure Multiparty Computation (MPC)* Cryptology ePrint Archive, Paper 2020/300. 2020. Available at: https://doi.org/10.1145/3387108.

[46] Liu, Y.; Zhang, L. and Zhao, Y. Deciphering bitcoin blockchain data by cohort analysis. *Scientific Data.* Nature Publishing Group UK London, 2022, vol. 9, no. 1, p. 136.

[47] Loguinov, D.; Casas, J. and Wang, X. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. *IEEE/ACM Transactions on Networking*, 2005, vol. 13, no. 5, p. 1107–1120.

[48] Lucas, B. and Páez, R. V. Consensus Algorithm for a Private Blockchain. In: *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication.* July 2019, p. 264–271.

[49] Maller, M.; Bowe, S.; Kohlweiss, M. and Meiklejohn, S. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security.* 2019, p. 2111–2128.

[50] McCorry, P. *Basics of hash function and their usage in cryptocurrencies* online. 2022. Available at: https://mirror.xyz/0xaFaBa30769374EA0F971300dE79c62Bf94B464d5/nufhx7EoEPwW9Zoe03rwPapL-AerI4sdDglTK4fN4lM. [cit. 2025-05-17].

[51] Menezes, A.; Katz, J.; Oorschot, P. van and Vanstone, S. *Handbook of Applied Cryptography*. CRC Press, 1996. Discrete Mathematics and Its Applications. ISBN 9781439821916. Available at: https://books.google.sk/books?id=MhvcBQAAQBAJ.

[52] Merkle, R. C. *Secrecy, authentication, and public key systems.* Stanford, CA, USA, 1979. Dissertation. AAI8001972.

[53] Micali, S.; Rabin, M. and Vadhan, S. Verifiable random functions. In: IEEE. *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*. 1999, p. 120–130.

[54] Nagy Ábel; Tapolcai, J.; Seres, I. A. and Ladóczki, B. *Forking the RANDAO: Manipulating Ethereum's Distributed Randomness Beacon* Cryptology ePrint Archive, Paper 2025/037. 2025. Available at: https://eprint.iacr.org/2025/037.

[55] Nakamoto, S. *Bitcoin: A peer-to-peer electronic cash system* online. 2009. Available at: http://www.bitcoin.org/bitcoin.pdf. [cit. 2025-05-17].

[56] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication FIPS PUB 186-5. U.S. Department of Commerce, February 2023. Available at: https://doi.org/10.6028/NIST.FIPS.186-5.

[57] Newhedge. *Newhedge - Bitcoin live dashboard* online. 2021. Available at: https://newhedge.io/terminal/bitcoin. [cit. 2025-05-17].

[58] Nguyen Van, T.; Le, T.-D.; Nguyen Anh, T.; Nguyen Ho, M.-P.; Nguyen Van, T. et al. A System for Scalable Decentralized Random Number Generation. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW)*. 2019, p. 100–103.

[59] Noether, S. Ring SIgnature Confidential Transactions for Monero. *IACR Cryptology ePrint Archive*, 2015, vol. 2015, p. 1098.

[60] Pettinari, P. and al. et. *Starky* online. 2022. Available at: https://ethereum.org/en/developers/docs/networking-layer/. [cit. 2025-05-09].

[61] Pivotal Software, Inc.. *RabbitMQ: Open Source Message Broker* https://www.rabbitmq.com/. 2007. [cit. 2025-05-17].

[62] Popov, S. The tangle. *White paper*, 2018, vol. 1, no. 3.

[63] Raikwar, M. and Gligoroski, D. Dos attacks on blockchain ecosystem. In: Springer. *European conference on parallel processing*. 2021, p. 230–242.

[64] Salen, R. and al. et. *Starky* online. 2024. Available at: https://github.com/0xPolygonZero/plonky2/tree/main/starky. [cit. 2025-05-09].

[65] Shahriar Hazari, S. and Mahmoud, Q. H. Improving transaction speed and scalability of blockchain systems via parallel proof of work. *Future internet*. MDPI, 2020, vol. 12, no. 8, p. 125.

[66] Sompolinsky, Y.; Wyborski, S. and Zohar, A. PHANTOM GHOSTDAG: A Scalable Generalization of Nakamoto Consensus: September 2, 2021. In:. New York, NY, USA: Association for Computing Machinery, 2021, p. 57–70. ISBN 9781450390828. Available at: https://doi.org/10.1145/3479722.3480990.

[67] Son, D. H.; Quynh, T. T. T. and Minh, L. Q. RANDAO-based RNG: Last Revealer Attacks in Ethereum 2.0 Randomness and a Potential Solution. *CoRR*, 2024, abs/2403.09541.

[68] Tovanich, N.; Soulié, N.; Heulot, N. and Isenberg, P. The Evolution of Mining Pools and Miners' Behaviors in the Bitcoin Blockchain. *IEEE Transactions on Network and Service Management*, 2022, vol. 19, no. 3, p. 3633–3644.

[69] Williams, A. Zero-Knowledge Proofs and Their Role within the Blockchain. *Communications of the ACM*. ACM New York, NY, USA, 2024, vol. 67, no. 7, p. 6–7.

[70] Wood, D. G. Join-Accumulate machine: a mostly-coherent trustless supercomputer. online, 2025.

[71] Wood, G. Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 2016, vol. 21, no. 2327, p. 4662.

[72] Wood, G. et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014, vol. 151, no. 2014, p. 1–32.

[73] Yaga, D.; Mell, P.; Roby, N. and Scarfone, K. Blockchain Technology Overview. *ArXiv*, 2018, abs/1906.11078. Available at: https://api.semanticscholar.org/CorpusID:69842399.

[74] Yu, G. Simple Schnorr signature with Pedersen commitment as key. *Cryptology ePrint Archive*, 2020.

[75] Zergity. *Unbiasable Randomness with VDF* online. 2023. Available at: https://hackmd.io/@Zergity/UnbiasableRNG. [cit. 2025-05-17].

# Appendix A

# Contents of the included storage media

Files in the root folder are organized as follows:

```
/
├── README.md
├── master-thesis-xhladk15.pdf
├── bootstrap-server ..... bootstrapping service
├── data-processor ..... data processor service
├── init.py ..... initialization script (main entry)
├── node-control ..... node control service
├── node-implementation ..... blockchain node implementation
├── requirements.txt ..... used Python modules
├── setup.yaml ..... last configuration used in init.py
├── topology-generator ..... network topology generator service
├── volume_experiment_60 ..... data gathered from experiment with 60 nodes
├── volume_experiment_40 ..... data gathered from experiment with 40 nodes
└── zkp-circuit-keys ..... pre-generated proving and verifying keys used by Gnark
```