BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PROTECTION TECHNIQUES OF DOS ON THE BLOCK PRODUCERS IN ETHEREUM

TECHNIKY OCHRANY PROTI DOS NA PRODUCENTY BLOKOV V ETHEREU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ KRAJČÍ**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. IVAN HOMOLIAK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

162896

Institut:	Department of Intelligent Systems (DITS)
Student:	**Krajčí Tomáš**
Programme:	Information Technology
Title:	**Protection Techniques of DoS on the Block Producers in Ethereum**
Category:	Security
Academic year:	2024/25

Assignment:

1. Get familiar with blockchains, smart contracts, Ethereum 2.0 PoS, and its consensus mechanism with respect to DoS on the block producers.
2. Study techniques of avoiding DoS and compare their theoretical features.
3. Implement at least 2 techniques for avoiding DoS in Ethereum proof of concept simulation.
4. Evaluate the performance and cost of the implemented technique.
5. Discuss possible extensions and potential drawbacks.

Literature:

- Gavin Wood, Ethereum Yellow paper, 2024 https://ethereum.github.io/yellowpaper/paper.pdf
- Ethereum, Secret leader election, 2024, https://ethereum.org/en/roadmap/secret-leader-election/
- Vitalik Butterin, Single and non-single proposer selection: introduction and context, 2022, https://ethresear.ch/t/secret-non-single-leader-election/11789

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

Supervisor:	**Homoliak Ivan, doc. Ing., Ph.D.**
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.10.2024

Abstract

The goal of this thesis is to explore cryptographic techniques to mitigate denial-of-service vulnerabilities on Ethereum block proposers by hiding leader identities. It implements and evaluates two Single Secret Leader Election protocols: WHISK (using SNARK-based shuffling) and Homomorphic Sortition (based on TFHE). Experimental results show that WHISK offers practical performance for integration into Ethereum, while Homomorphic Sortition provides stronger privacy at a higher computational cost.

Abstrakt

Cieľom tejto práce je preskúmať kryptografické techniky na zmiernenie zraniteľností typu denial-of-service (DoS) na producentov blokov v Ethereu pomocou skrytia identity navrhovaného lídra. Práca implementuje a vyhodnocuje dva protokoly pre jednorazový tajný výber lídra (Single Secret Leader Election – SSLE): WHISK (založený na premiešavaní pomocou SNARK dôkazov) a Homomorphic Sortition (využívajúci plne homomorfné šifrovanie – TFHE). Experimentálne výsledky ukazujú, že WHISK poskytuje praktický výkon vhodný na integráciu do Etherea, zatiaľ čo Homomorphic Sortition ponúka vyššiu úroveň súkromia za cenu vyššej výpočtovej náročnosti.

Keywords

Denial of Service (DoS), Ethereum, Block Producers, Protection Techniques, Security, Blockchain, Whisk, Homomorphic Sortition, Secret single leader election (SSLE)

Klíčová slova

Odmietnutie Služby (DoS), Ethereum, Producent Blokov, Ochranné Techniky, Bezpečnosť, Blockchain, Whisk, Homomorphic Sortition, Secret single leader election (SSLE)

Reference

KRAJČÍ, Tomáš. *Protection Techniques of DoS on the Block Producers in Ethereum.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Ivan Homoliak, Ph.D.

Protection Techniques of DoS on the Block Producers in Ethereum

Declaration

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

.....

Tomáš Krajčí

May 12, 2025

Contents

List of Figures

Chapter 1

Introduction

Blockchain technology has introduced a paradigm shift in how distributed systems operate, enabling trustless, transparent, and secure exchange of value and information. At the heart of this innovation lies the concept of smart contracts, self-executing programs that run on decentralized infrastructure, eliminating the need for intermediaries and automating agreement enforcement.

Ethereum, a leading smart contract platform, initially relied on Proof-of-Work (PoW), but has since transitioned to Proof-of-Stake (PoS) in its Ethereum 2.0 upgrade. Although PoS brings efficiency and scalability improvements, it also introduces new challenges related to the public visibility of validator assignments.

One such vulnerability is the risk of Denial-of-Service (DoS) attacks targeting future block proposers. By knowing who will propose a block in advance, adversaries can disrupt the protocol by preemptively disabling these validators. This thesis explores Single Secret Leader Election (SSLE) protocols as a cryptographic solution to this problem, focusing on the Whisk and Homomorphic Sortition constructions.

1.1 Contributions

- Proof of concept, a modular C++ implementation of the Whisk protocol, including tracker generation, DLEQ proof of ownership, verifiable shuffling using SNARKs (via *libsnark*), and public proof verification.
- Proof of concept, a fully encrypted implementation of the Homomorphic Sortition protocol, using the TFHE scheme to execute all core operations: stake comparisons, selection logic, and PRF evaluation over ciphertexts.

  To the best of our knowledge, this is the first working implementation of the Homomorphic Sortition protocol as described in [20], realized using TFHE for circuit-level privacy-preserving leader selection.
- A side-by-side analysis of both protocols regarding cryptographic assumptions, privacy guarantees, and performance trade-offs.

1.2 Organization

The structure of this thesis is organized as follows:

- **Chapter 2** introduces the fundamental principles of blockchain technology, covering its structure, consensus mechanisms, and categorization into permissionless and permissioned networks.
- **Chapter 3** focuses on Ethereum, its architecture, the transition to Proof-of-Stake (PoS), and related concepts such as smart contracts, gas, and validator roles.
- **Chapter 4** analyzes denial-of-service (DoS) attacks, particularly those targeting Ethereum's block producers. It presents historical incidents, possible attacker strategies, and theoretical analysis of WHISK and Homomorphic Sortition.
- **Chapter 5** details the implementation of the two SSLE protocols: WHISK and Homomorphic Sortition. It describes their design, cryptographic components, and integration logic. Performance metrics are evaluated through instrumented test runs.
- **Chapter 6** shows comparative evaluation of both protocols in multiple dimensions, including efficiency, privacy, and applicability to Ethereum. Discusses their limitations and possible future improvements.
- **Chapter 7** concludes the thesis by summarizing the contributions and key insights gained from the study of privacy-preserving leader election protocols.

Chapter 2

Blockchain

This chapter provides an overview of blockchain technology, explaining its core components, mechanisms, and fundamental principles that make it a secure and transparent distributed ledger.

2.1 What is blockchain?

A blockchain is a distributed ledger consisting of blocks that contain cryptographically signed transactions and links between these blocks. Each block (except the very first one) includes a cryptographic link (hash value) of the previous block, as shown in Figure 2.1 [33]. Due to this, as new blocks are created, modifying older blocks becomes increasingly more difficult. Any changes would propagate to subsequent blocks, making alterations easily detectable. To further protect against tampering, blockchains utilize a consensus mechanism that ensures that any modification requires the majority approval of the participants. In addition, each new block is replicated in every copy of the ledger in the network. This design aims to maintain the integrity and trustworthiness of the blockchain.

Ledger

A ledger is a collection of transactions. Throughout history, written ledgers have been used to track the exchange of goods and services. In modern times, digital ledgers have replaced old-fashioned pen and paper. However, they are often owned and operated by a trusted centralized third party on behalf of the user community [33]. In contrast, blockchain is a decentralized, digital, and distributed ledger that allows transparent and secure information sharing among the peer-to-peer network (P2P) [27].

Key Features

Blockchain technology is built on a foundation of unique features distinguishing it from traditional record-keeping systems.

- Decentralization - Blockchain operates without a central authority, distributing control across the network's participants.
- Transparency - All transactions are visible and verifiable by network participants.
- Immutability - Data recorded in a blockchain is tamper-proof, as changes to one block require altering all subsequent blocks.

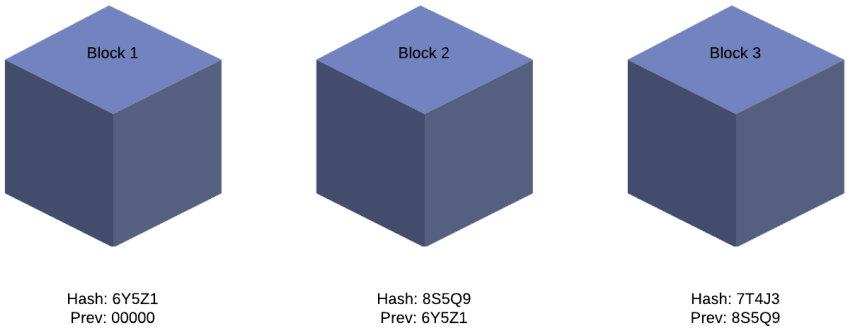

Figure 2.1: Blockchain illustration.

- Security - Blockchains are cryptographically secure, ensuring that data contained in the ledger has not been tampered with.

Blockchain technology operates without relying on centralized authorities or third-party verification, as transactions are validated through mutual confirmation and consensus among network participants. However, not all blockchains implement these features in a similar way, as their design and use cases may vary.

Blockchain gained widespread attention due to the Bitcoin cryptocurrency, introduced in 2008 by Satoshi Nakamoto's groundbreaking paper [23] that describes a decentralized payment system. Although Bitcoin was the first implementation of blockchain technology, blockchain itself has broader applications beyond cryptocurrencies. It can be used for tasks such as managing electronic medical records, issuing digital certificates, tracking supply chains, enabling voting systems, and more.

Blockchain is not always the ideal solution for all scenarios, despite its innovative features. Belotti et al. [1] provides a decision framework to evaluate whether a blockchain or a traditional ledger is more appropriate according to specific requirements. The decision framework highlights that blockchain suitability depends on the context and purpose of its implementation.

2.2 Peer-to-Peer Network (P2P)

The term „peer-to-peer“ refers to the type of architecture where the participants are both the consumers and the providers of resources and services. Having both roles, which means both the client and the server, enables the participants to send and receive information and services from one another. In other words, participants are peers in a peer-to-peer network.

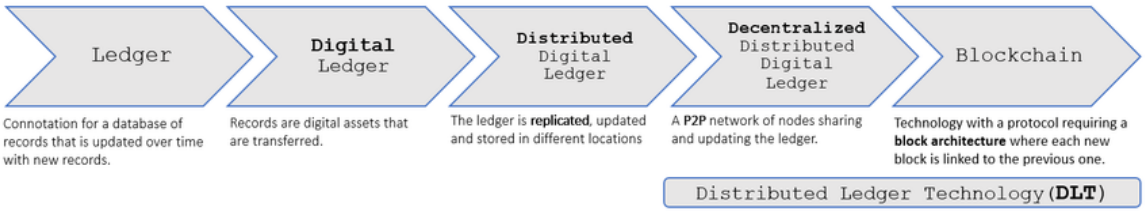

Figure 2.2: Evolution from traditional ledger to blockchain [1].

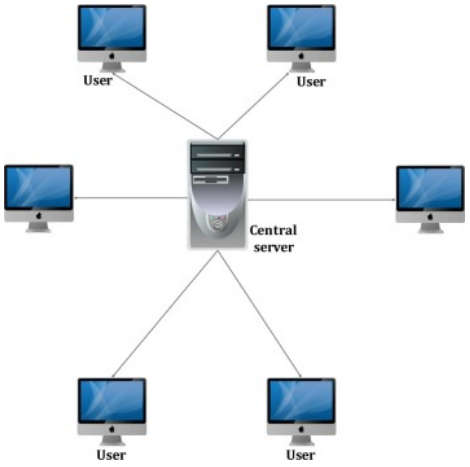

Figure 2.3: Centralized network-server based.

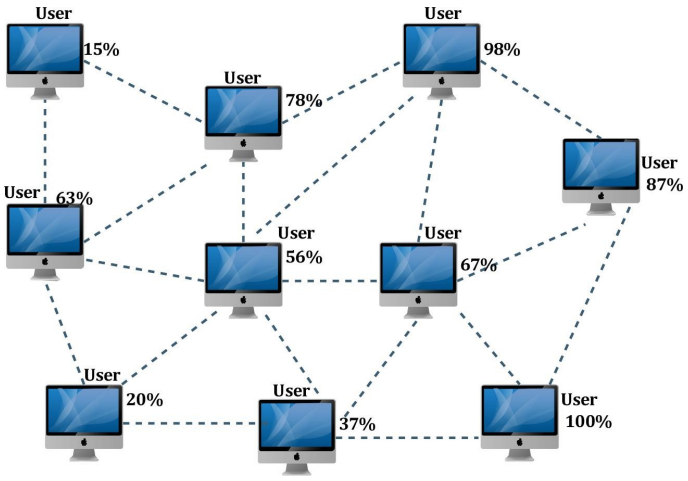

Figure 2.4: Decentralized network - P2P.

Figure 2.5: Networks side by side [26].

Creating an interconnected collection of all nodes in which the entire load is distributed among all participants. However, peers are distributed and decentralized. The main benefit of using P2P networks is data/file sharing. In the decentralized P2P network, if one of the peers fails to operate, the other peers assume the role and perform the required function [26]. See Figure 2.5 for the difference between a centralized and a decentralized network.

Blockchain technology relies on a peer-to-peer (P2P) network to maintain its decentralized structure. Sharing and validating data without the need for a central authority. Transactions are broadcast to the network, where nodes work together to verify their authenticity and reach a consensus on the blockchain's state. This decentralized communication ensures transparency and fault tolerance. The P2P network is fundamental to the blockchain's ability to distribute and synchronize data securely across all participants in real-time.

2.3 Blockchain Categorization

Blockchain networks are categorized by their permission model, which determines who can participate in the network and publish blocks. Two primary types are permissionless and permissioned.

Permissionless

Permissionless blockchain networks are open to anyone who can publish blocks, read the blockchain, and submit transactions without prior authorization. These networks are often open-source software, meaning anyone who wishes to download and join them can do so. However, the openness of permissionless networks also introduces the risk of malicious intent by users who attempt to affect and manipulate the system by publishing invalid or disruptive blocks. To prevent this, permissionless blockchain networks often utilize a multiparty agreement or 'consensus' system (see Section 2.6) that requires users to expend or maintain resources when attempting to publish blocks. The consensus system prevents malicious users from effortlessly subverting the system [33]. These mechanisms impose

high costs on malicious actors trying to undermine the network while incentivizing honest behavior and ensuring the blockchain's integrity by rewarding publishers of valid, protocol-compliant blocks with native cryptocurrency or transaction fees.

Permissioned

Permissioned blockchain networks are systems where only authorized users (by some authority) can publish blocks, depending on the implementation, which may or may not restrict access to reading the blockchain or creating transactions. Since these networks offer controlled access, they are suitable for use in organizations that require tighter control or collaboration. The permissioned blockchain can be operated using open-source or closed-source software.

Permissioned blockchain networks can have the exact traceability of digital assets as they pass through the blockchain, and the same distributed, resilient, and redundant data storage system as permissionless blockchain networks [33]. They also rely on consensus mechanisms for block publishing, but these methods are typically more resource-efficient and less costly to maintain than permissionless networks. This efficiency comes from the fact that participants are required to establish their identity, creating a level of trust among those maintaining the network because all members are authorized participants.

2.4 Blocks

A block in a blockchain is a composite structure comprising two parts: the Header and Transactions (data). It should be noted that every blockchain implementation can define its data fields. However, this is the most common implementation. [33]

Block Header

- The block number, also known as the block height.
- The hash value of the header of the previous block.
- A hash representation of block data.
- A timestamp.
- The size of the block.
- The nonce value. For blockchain networks that utilize mining, this number is manipulated by the publishing node to solve the hash puzzle. Other blockchain networks may or may not include it or use it for a purpose other than solving a hash puzzle.

Block Data

- A list of transactions and ledger events included within the block.
- Other data may be present, such as signatures or consensus-related data.

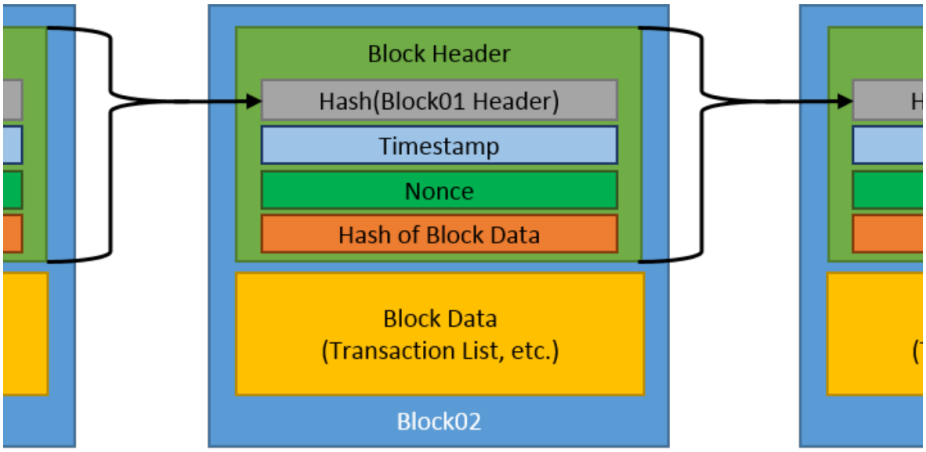

Figure 2.6: Generic chain of blocks [33].

Merkle Tree

The Merkle Tree (patented in 1989 [25]) is a cryptographic hash tree structure that hierarchically organizes and verifies data, such as transactions. Transactions, represented as leaf nodes, are hashed using a cryptographic hash function. These hashes are then paired and hashed iteratively to form parent nodes, continuing this process until a single hash remains, known as the Merkle root. If the number of transactions is odd, the last transaction is duplicated and hashed with itself to maintain pairing consistency. Example Merkle Tree in Figure 2.7

In terms of blockchain, this hash, stored in the block header, acts as the cryptographic fingerprint of all transactions within the block. Merkle Tree's primary purpose in blockchain is to ensure the integrity, immutability, and verifiability of transaction data.

2.5 Transactions

Transactions represent the fundamental mechanism through which users interact within a blockchain network. Transaction details contain sender and receiver information and the amount to be transferred. It is considered the smallest building block in the block system that holds the information, records, etc. [26]. Although commonly associated with financial exchanges, transactions are not strictly limited to transferring value. They can also include recording data, transferring digital assets, or executing code.

Every transaction, once validated, is placed in a new block, which is added to the transaction ledger and linked to the previous one. This results in an update of the system state and users' local copy of the blockchain. Whenever a user aims to interact with another in the network, one or multiple transactions are created, propagated, validated, and confirmed by the network. Each blockchain-based system differs in how the 'transaction journey' steps are carried out. This journey starts when the transaction is created and ends when the transaction is recorded in the blockchain [1].

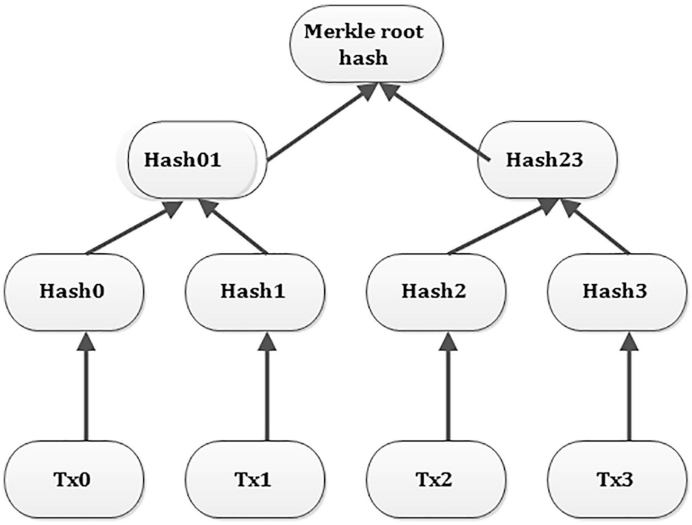

Figure 2.7: Example Merkle Tree with transactions Tx0-Tx3 [26].

Transaction Journey

The journey of the transaction consists of five crucial steps, as follows [1]:

- Creation - Each blockchain adopts a predefined data structure that determines certain benefits and drawbacks. Transactions specify the origin, destination, and conditions for redeeming the „object of the transfer“ (e.g., digital assets). These conditions can range from simple scripts to complex smart contracts. It is up to the sender to define these according to the specified transaction structure.
- Propagation - Transaction (or eventually the block) is broadcast to validating peers (nodes). Efficient transaction broadcasting is critical for processing speed, with communication protocols optimized for performance while resisting attacks.
- Validation - The most critical step, where transactions in blocks undergo the different stages of the consensus mechanism envisaged to be considered valid and, therefore, executable. Once validated, the block is added to the blockchain, updating its state.
- Propagation - After validation, the block is propagated across the network so all nodes can update their copies of the blockchain.
- Confirmation - Transactions are finalized only after validation and publication in the blockchain. The consensus procedure has to come to an end; that is, the nodes have to agree on a single chain of blocks.

Transaction Signing

Before a transaction is propagated to the blockchain network, it must be signed by the sender to ensure its authenticity, integrity, and irrefutability. This process is a fundamental part of the **Creation** step in the transaction journey. Signing a transaction involves cryptographic methods using a pair of keys: the sender's private key and public key.

When a user creates a transaction, its details are hashed to generate a unique representation of the transaction. The sender then uses their private key to sign this hash, creating

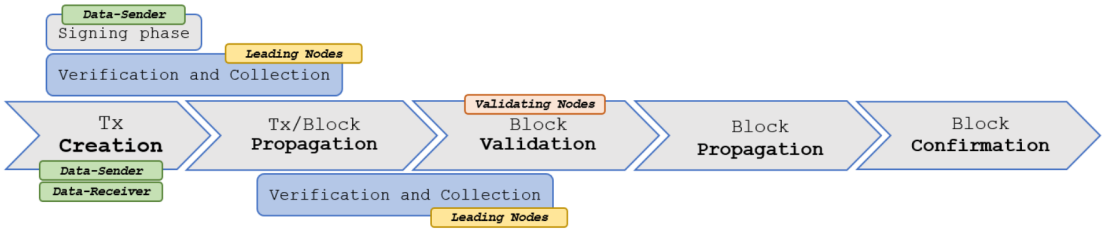

Figure 2.8: Journey of transaction[1].

a digital signature. The digital signature and the sender's public key are attached to the transaction to allow nodes in the network to validate it.

During the **Validation** step, network nodes verify the transaction by decrypting the digital signature using the sender's public key and comparing the decrypted hash with the hash of the transaction data. If the two match, the transaction is confirmed as authentic, untampered with, and originating from the rightful sender. This signing mechanism ensures the security and trustworthiness of transactions before they undergo propagation, validation, and confirmation in the blockchain network.

2.6 Consensus Mechanism

Since blockchain works using a decentralized network model, a trusted third party or central authority does not exist. Therefore, a consensus mechanism determines which node publishes the next block or ensures that the nodes agree on which block to add as the new block.

Why is this important? For permissionless blockchain networks, there are generally many publishing nodes competing simultaneously to publish the next block. They usually do this to win cryptocurrencies and/or transaction fees. Note that legal remedies may be used for permissioned blockchain networks if a user acts maliciously [33]. Therefore, each node is likely motivated by the desire for financial gain, not the well-being of other nodes or the network. Why would a user propagate a block that another user is attempting to

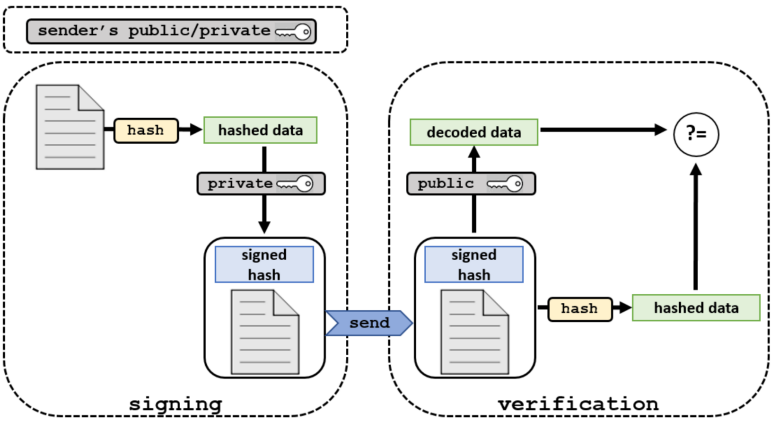

Figure 2.9: Illustration of signing of transaction [1].

publish? Who resolves the conflicts when multiple nodes publish a block simultaneously [33]?

Blockchain technologies use consensus mechanisms to enable users to work together without the need to trust each other fully. These consensus mechanisms vary according to the blockchain design:

- **Proof of Work (PoW)** - Requires nodes (miners) to solve computational puzzles, rewarding the successful miner with cryptocurrency and transaction fees (e.g., Bitcoin) 2.6
- **Proof of Stake (PoS)** - Elects validators based on their staked cryptocurrency, offering rewards for validating blocks and penalizing malicious activity (e.g., Ethereum 2.0) 2.6
- Other mechanisms, such as **Delegated Proof of Stake (DPoS)**, **Proof of Authority (PoA)**, and **Byzantine Fault Tolerance (BFT)**

When multiple nodes simultaneously publish valid blocks, consensus mechanisms include conflict resolution strategies to maintain consistency. For example, in PoW-based blockchains, the longest chain rule ensures that the chain with the most cumulative computational work becomes the valid one (the longest chain of blocks). This reduces the likelihood of permanent forks in the blockchain.

In permissionless blockchains, incentives play a critical role. Honest behavior is rewarded, while malicious actions may result in penalties, ensuring that participants act in their financial interest, which aligns with the network's integrity. On the other hand, permissioned blockchains often rely on legal frameworks to address disputes and enforce accountability among pre-approved participants, reducing the reliance on complex consensus protocols.

Proof of Work (PoW)

In the Proof-of-Work (PoW) mechanism, a user publishes the next block by being the first to solve a computationally intensive puzzle. The solution to this puzzle is the "proof" they have performed their work [33]. This puzzle requires a significant computational effort to be solved, but is, in comparison, very easy for others to verify. Typically, the puzzle involves finding the hash value of the block header of the next block that is below a specified target. Achieved by iterating through different nonce values. Nodes that solve this computational puzzle are rewarded to increase the incentive, usually by rewards such as cryptocurrency or transaction fees.

The difficulty of this puzzle is dynamically adjusted to regulate the block-creation rate, typically requiring a hash with a specific number of leading zeros. For example, Bitcoin, which uses the proof of work model, adjusts the puzzle's difficulty every 2016 blocks to influence the block publication rate to be around once every ten minutes [33]. These difficulty adjustments aim to ensure that no entity can take over block production. However, with these adjustments, the puzzle-solving computations also increase their energy consumption.

Once a block is solved, it is sent to full nodes to verify that the new block satisfies the puzzle requirement. It is simple and requires only one hash [33]. If it does, it is added to the blockchain of the full nodes and quickly distributed across the network of participating nodes.

Each individual puzzle is independent of previous or future puzzles, ensuring that miners (nodes) abandon unsolved work once a valid block is received and build on the newly created block.

Proof of Stake (PoS)

Proof of Stake (PoS) is a consensus mechanism in which the likelihood of publishing a new block is proportional to the amount of cryptocurrency („stake“) a user has invested in the network. The idea beyond the mechanism is that users with more commitments are unlikely to attack the blockchain [26]. Users stake their assets by locking them in the network. Once staked, the cryptocurrency is generally no longer capable of being spent. Proof of stake blockchain networks use a user's stake as a determining factor for publishing new blocks [33]. The staked currency incentivizes them to act in the system's best interest. Therefore, the higher the amount of cryptocurrency the user has staked, the higher the probability that they are the publisher of the next block.

Resource-intensive computations, such as those required in Proof of Work, which require significant time, electricity, and processing power, are unnecessary in this consensus mechanism. Because this model is more resource-efficient, some blockchain networks choose not to offer block creation rewards. Instead, these systems distribute all the cryptocurrency upfront to users, eliminating the need for continuous coin generation. In such cases, block publishers typically earn rewards in the form of transaction fees provided by users.

2.7 Blockchain technology summary

In conclusion, a blockchain is a distributed, decentralized, and immutable ledger that records transactions in blocks linked together using cryptographic hashes. The key components of blockchain technology include asymmetric cryptography, digital signatures, and cryptographic hashes, all of which ensure security, transparency, and data integrity. Communication within the blockchain is facilitated through a peer-to-peer (P2P) network, enabling decentralized collaboration among participants. Verification and validation of transactions are achieved through a consensus mechanism, an algorithm that ensures agreement among network nodes without the need for a central authority.

Chapter 3

Ethereum

This chapter serves as a brief introduction to Ethereum, a decentralized blockchain platform. Explores the foundation of Ethereum, how it operates, and its underlying structures.

3.1 What is Ethereum

Ethereum is a decentralized open-source blockchain platform that enables the creation and execution of smart contracts (3.2). Unlike Bitcoin, which primarily serves as a digital currency, Ethereum offers a programmable blockchain powered by the Ethereum Virtual Machine (EVM), allowing developers to build and deploy applications without intermediaries. Initially secured by Proof-of-Work (PoW), Ethereum has transitioned to a more efficient Proof-of-Stake (PoS) consensus mechanism through Ethereum 2.0, improving scalability, energy efficiency, and security.

Ethereum's Origin

Vitalik Buterin conceived Ethereum in late 2013, and it was publicly announced in January 2014, with an official launch on July 30, 2015. Vitalik envisioned a blockchain that could go beyond being just a digital currency. His idea was to create a platform capable of programmable smart contracts and decentralized applications, allowing developers to build systems on top of a basic framework without creating a separate blockchain for each new project [13].

Ethereum was developed through collaboration with notable contributors, including Gavin Wood, who authored the Ethereum Yellow Paper detailing its technical specifications [31], and Joseph Lubin, who helped establish ConsenSys, an organization focused on Ethereum-based projects.

The pioneering vision transformed Ethereum into a versatile blockchain platform, becoming the foundation of many innovations, such as decentralized finance (DeFi) and non-fungible tokens (NFTs).

Ethereum Virtual Machine (EVM)

The Ethereum virtual machine, EVM for short, is a decentralized virtual environment capable of running code consistently and securely across all nodes in Ethereum [18]. As a run-time environment, EVM enables the execution of smart contracts deterministically, ensuring that the same input equals the same output every time. EVM is a quasi-Turing

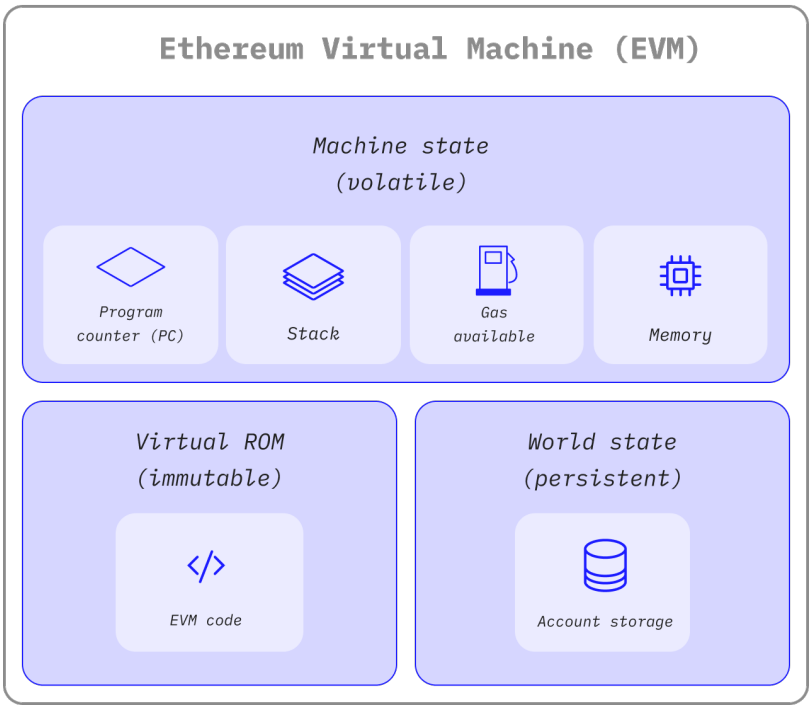

Figure 3.1: Illustration of EVM [19].

complete machine that can execute any computation given enough resource called gas (See Section 3.3).

Due to its ability to execute smart contracts, Ethereum is not a pure distributed ledger like Bitcoin; instead, it is more of a state machine. In this case, the state is a large data structure, a modified Merkle Patricia Trie 3.3, which holds not only the accounts and their balances, but also the current state of the Ethereum machine. This state can change block from block according to pre-defined rules defined by the EVM.

3.2 Smart contracts

A smart contract is a self-executing code stored and running on the blockchain. These contracts are designed to automatically enforce and execute agreements based on predefined conditions in a deterministic way. By eliminating the need for intermediaries, smart contracts aim to reduce errors and enhance reliability in contractual agreements. They follow the logic „if this then that“ and are guaranteed to execute according to the rules defined by their code, which cannot be changed once created [17]. Smart contracts can receive, store, and send funds and even interact with other smart contracts.

Nick Szabo first introduced the concept of smart contracts in the 1990s, envisioning contracts embedded in code to execute predefined rules automatically [29]. Ethereum brought this idea to life by integrating smart contracts into a decentralized blockchain environment.

Vending machine analogy

A simple analogy used to showcase smart contracts is a vending machine. Typically, vending machines are programmed so that specific inputs result in predetermined outputs.

Typical usage of the vending machine:

1. Selection of product
2. The vending machine displays the price
3. You pay the price
4. The vending machine checks if u paid the right amount
5. You receive your product

This analogy highlights the deterministic and automated nature of smart contracts. Once the inputs meet the conditions, the output is guaranteed without the need for a trusted intermediary or manual intervention.

Life Cycle of Smart Contract

Smart contracts operate in four main steps:

1. Contract creation - A developer writes the contract in a high-level programming language like Solidity and deploys it to the blockchain.
2. Interaction - Users send transactions to the contract's address to trigger its functions.
3. Execution - The contract evaluates the inputs and executes predefined logic if conditions are met.
4. Storage - Results are stored immutably on the blockchain, ensuring transparency and reliability.

In Ethereum, all smart contracts deployed are immutable. An immutable contract means that once a contract is created and deployed, it cannot be modified, although it can be deleted. Also, on Ethereum, smart contracts are decentralized, meaning no single machine controls the contract. In fact, all nodes on the Ethereum network have the same contract with the same state.

Use cases for Smart Contracts

The use cases for smart contracts have been growing, ranging from payments and decentralized finance to supply chain and crowdfunding. I want to introduce a few:

- Decentralized Finance (DeFi) - Automated lending, trading, and yield farming protocols
- Crowdfunding - A contract that will only unlock funds once certain goals are achieved.
- Supply chain management - Transparent tracking of goods and payment settlements.
- Payments - Enabling automated, secure, and transparent payment systems.

Smart vs Traditional contracts

Smart contracts significantly improve traditional contracts by utilizing automation, decentralization, and transparency. Traditional contracts often require a third party to facilitate the agreement, which can introduce delays, costs, and risks. Smart contracts are faster. They execute within seconds of the conditions being met. They are a more cost-efficient alternative while also being reusable. As traditional contracts are often tailored to a specific individual, the re-usability comes in the form that anyone who meets the conditions can execute this smart contract. Their use of blockchain technology enhances security and ensures deterministic transactions and trustworthiness.

Smart contracts offer several advantages and disadvantages when used for transaction automation. One of the biggest positives is that they are fully automated, reducing the need for intermediaries and manual intervention. This leads to a faster, more cost-efficient, and precise execution of these agreements. Their deterministic nature ensures that the outcome is always predictable when the conditions for their execution are met. However, they are not without their share of negatives. Software bugs can expose vulnerabilities, while protocol changes can disrupt operations. The lack of clear regulations and tax guidelines creates uncertainty for businesses and individuals. In general, the benefits of smart contracts make them a powerful tool for modern digital transactions. However, we can fully harness their potential only by being aware of and addressing the vulnerabilities.

3.3 Ethereum's Blocks

Based on the information discussed in Section 2.4 about blocks in general blockchains. Let me introduce the unique components of Ethereum blocks that facilitate its smart contract functionality and state management. The key elements are the following [32].

- **State root** - It is a hash value of the root node that represents the current state of the Ethereum blockchain system, e.g., account balance, contract storage, contract code, and account nonce of all accounts.
- **Transaction Root** - Represents all transactions in the block. Nodes can use this to efficiently verify that a particular transaction exists in the block.
- **Beneficiary** - An Ethereum address that receives the priority fees (transaction tips) from transactions in the block. In PoS, the block proposer (validator) earns the fees for proposing and finalizing the block.
- **Receipts Root** - Stores receipts (outcomes) of all transactions, including gas usage and logs. This allows for efficient verification of transaction execution results.
- **Logs Bloom** - It is a filter that provides an efficient way to find a specific log or event within the block.
- **Withdrawals Root** - Tracks withdrawals from the block.
- **prevRandao** - Used for randomness generation
- **Gas Limit** - Limit to the amount of work each block can do, ensuring the network's capacity is enough.

- **Gas Used** - The amount of gas used by transactions within the block.
- **Base Fee Per Gas** - The minimum amount of gas burned per unit of gas used in a transaction.

Gas

I have introduced an unfamiliar term, gas, in the environment of Ethereum, which is used as a form of fee. In Ethereum, each computational action has a set „gas“ price. Your gas fees are the total cost of the actions in your transaction [15]. Whenever a user decides to send a transaction or execute a smart contract, the gas fee is paid for the processing.

Why is gas needed? Gas is a critical element for executing transactions and keeping Ethereum secure. Gas is used to prevent malicious actors from overwhelming the network. Because computations cost gas, spamming or flooding Ethereum with expensive transactions is financially penalized. Gas works as a way to limit the number of transactions possible at one time, preventing the network from being overwhelmed while keeping it always accessible.

Now we know what gas is, but how is it calculated? The calculation consists of a few parts. Base fee, priority fee, and units of gas used. Every block has a base fee that acts as a reserve price. This part is burned and removed from the circulation once a block is established. The priority fee or tip is an incentive for the block validators to choose your transaction to include in the block. Without this, it would be viable for the validators to produce empty blocks. The higher the tip, the higher the chance for the validators to pick your transaction. This creates a competition for inclusion in a block. Units of gas represent the difficulty of action. More computation equals more gas fees. Executing smart contracts will cost more than executing a simple transaction.

Merkle Patricia Trie

The Merkle Patricia Trie is a modification of the Merkle Tree 2.4, where instead of nodes representing the entire hash, one node represents just one part of the hash. This allows the structure to represent the correct path and cryptographically verify if the data is valid in the first place. In other words, it keeps the blockchain valid by combining the structure of a standard Merkle tree with the structure of a Radix tree. Since all search and sorting algorithms on Ethereum must be filtered through this stringently correct database, the accuracy of the information is guaranteed [5].

3.4 Accounts and Transactions

An account in Ethereum is a fundamental concept. It is an entity with an ETH balance that facilitates transactions. All accounts are stored in a big table that is part of the EVM state. Accounts can be user-controlled or deployed as smart contracts [10]. As stated before, accounts come in 2 types: Externally owned accounts (EOA) and Contract accounts. Both types can receive, hold, or send ETH or tokens and interact with smart contracts.

However, EOA accounts are controlled by the user using a pair of cryptographic keys (public and private). EOA accounts can initiate transactions, which contract accounts cannot do. These accounts can only be initiated by an incoming transaction that enables their smart contract code. When it comes to the creation of accounts, EOA accounts are free; in contrast, deploying a contract account has a cost because it uses network storage.

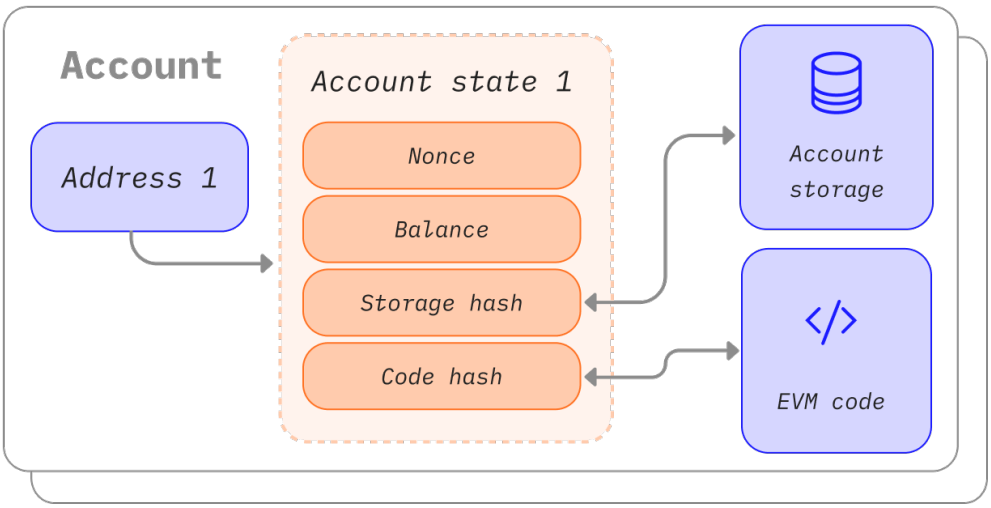

Figure 3.2: Illustration of the account on Ethereum [10].

In addition, transactions sent to a contract account can trigger code execution. This ability to execute code upon receiving a transaction allows contract accounts to perform actions such as token transfers, contract creation, and interaction with other smart contracts.

What is the structure of an Ethereum account? An Ethereum account has four fields [10].

- **Nonce** - The number of transactions sent from this account for the EOA or the number of contracts created by the contract-owned account. Why is this useful? Only one transaction with a given nonce can exist and be executed, meaning repeated re-execution of transactions is impossible.
- **Balance** - The current balance of the account
- **codeHash** - This field is a reference for the contract code in the EVM. The code that gets executed once a transaction calls the account. This applies only to contract accounts; the EOA account has this field as an empty string.
- **storageRoot** - The hash of the storage content of the account. By default, empty.

On Ethereum, a transaction is a cryptographically signed instruction from an externally owned account. An instruction to update the current state of the Ethereum network, which needs to be broadcast to the entire network [11]. Once the request has been broadcast, a validator must execute it and propagate this state change to the network. A transaction includes necessary components such as the recipient address, the amount of ether to be transferred, the gas limit, the maximum fees the proposer is willing to pay, and the data to invoke a function in a smart contract. Each transaction needs to be signed using the sender's private key to ensure that the transaction has come from the sender and is not a dishonest action.

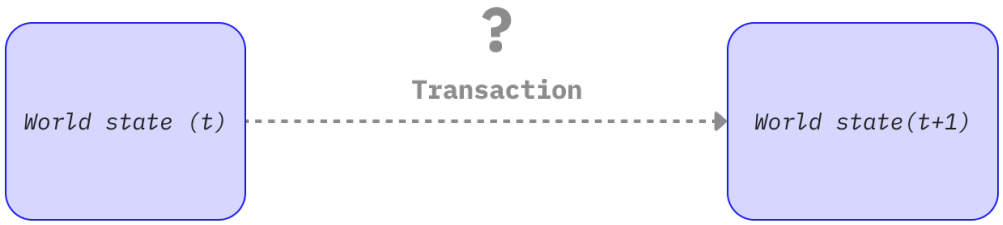

Figure 3.3: Illustration of a transaction on Ethereum [11].

3.5 Ethereum 2.0 and PoS

Ethereum 2.0, or Eth2, is a set of upgrades to Ethereum with the primary goal of improving Ethereum's scalability, security, and sustainability. The most significant change introduced with Ethereum 2.0 is the shift from a proof-of-work to a proof-of-stake consensus mechanism. I have covered what PoS is in Section 2.6. However, let us be a bit more specific about the Ethereum network. Users who want to become validators must stake at least 32 ETH or be part of a stake pool with at least 32 ETH in a smart contract on Ethereum. From then on, each validator will be responsible for checking the validity of new blocks shared across the network. Each validator is incentivized to validate transactions by receiving block rewards and transaction fees. Ethereum implements a mechanism called „slashing" to deter dishonest behavior, where validators lose part of their stake if they act maliciously. This shift to PoS significantly reduces Ethereum's energy consumption, making it more resource-efficient and environmentally friendly.

Except for depositing the ETH required to become a validator, each validator has to run three separate pieces of software: an execution client, a consensus client, and a validator client [14]. Once the ETH is deposited, the user joins a queue to become a validator. This queue limits the rate of acquisition of new validators on the network. Once a validator is active, it receives new blocks from its peers. The block's signature is checked, and each transaction is re-executed to confirm its validity. Afterward, the validator sends a vote (an attestation) to favor this block across the network.

Unlike PoW, where block production is determined by mining difficulty, PoS operates on a fixed time schedule. Time is divided into slots and epochs. One slot is 12 seconds, and one epoch is 32 slots. Each slot gets a validator that is responsible for proposing a block and propagating it to the network, while a committee of validators is randomly chosen for each slot, with the task of determining the validity of the proposed block. The division into committees is necessary to keep the network load manageable. All validators participate in every epoch, but not in every slot.

The transition of Ethereum 2.0 to PoS has also improved accessibility by lowering hardware requirements and allowing more involvement through stake pools. It has also introduced the concept of finality. Each transaction has finality when it is part of a block that would require a large amount of ETH to be burned to change. In Ethereum 2.0, this is implemented using „checkpoint" blocks. Each epoch (32 slots or 6.4 minutes) begins with a checkpoint block. Validators vote on pairs of checkpoints; when a pair receives two-thirds of the total staked ETH as votes, the last checkpoint becomes justified, and the previous justified checkpoint becomes finalized.

Being a validator has its rewards, but also its risks. It has potential avenues for malicious activity and sabotage. Ethereum implements strict penalties for validators who fail to fulfill their responsibilities or behave inappropriately. However, the rewards are as attractive.

RANDAO mechanism

RANDAO (Random Number DAO) is a decentralized mechanism used in Ethereum to generate secure and unbiased randomness in its PoS consensus system. RANDAO works in two phases. In the first phase, validators submit a commitment to a randomly generated number, ensuring that it remains concealed. In the second phase, validators reveal their commitment to their numbers, which are usually combined using an XOR operation to produce a final random number. Randomness plays a critical role in Ethereum; It is used in assignment validation, block proposer selection, and other operations requiring randomness. Its decentralized nature ensures that no single entity can manipulate this randomness.

Chapter 4

Protection Techniques against DoS

This chapter introduces what DoS is, how it affects block producers on the Ethereum network, and what techniques are used to combat it.

4.1 Denial of service (DoS)

What is a DoS attack? A denial-of-service attack attempts to overload a website or network, in the case of Ethereum, its block producers. The attack aims to degrade the network's performance or even make it inaccessible. The typical result of a successful attack is the loss in availability of a part or all of the system [24].

DoS attacks are a significant threat and can easily be directed at networks, systems, and services. The type of attack depends on the vulnerability being exploited. For example, some may overload a network's bandwidth, disabling its ability to send or receive network traffic. Others may exhaust a server's processing capabilities. The severity of the attack depends heavily on the specific target, the attack's duration, and the defendant's response. Regardless of the method used, a successful attack will leave the targeted system unreliable and unresponsive. On Ethereum, the attack on a selected block proposer can result in them not proposing a block, missing out on reward fees, and even getting fined for doing so.

Types of DoS Attacks

Denial-of-Service (DoS) attacks come in various forms; surprisingly, some can even be unintentional. In this section, I will explore different ways in which DoS attacks can be executed. First, we will discuss distributed denial-of-service (DDoS) attacks. Although the goal remains the same as that of a standard DoS attack - disrupting the availability of a target. DDoS attacks differ in their execution. Instead of a single attacker, DDoS attacks leverage multiple compromised devices, known as a botnet, to coordinate and overwhelm the target, making mitigation significantly more challenging. See figure 4.1.

1. **Volume-Based Attacks**

   Volume-based attacks flood the network with excessive network traffic, consuming its bandwidth and making it unusable. The magnitude is measured in bits per second (Bps). For example, **UDP flood** and **ICMP flood**. UDP floods send a barrage of UDP packets to random ports on the server, making it difficult to handle all these requests. ICMP floods (Ping attacks) operate similarly by overwhelming the server with echo requests.

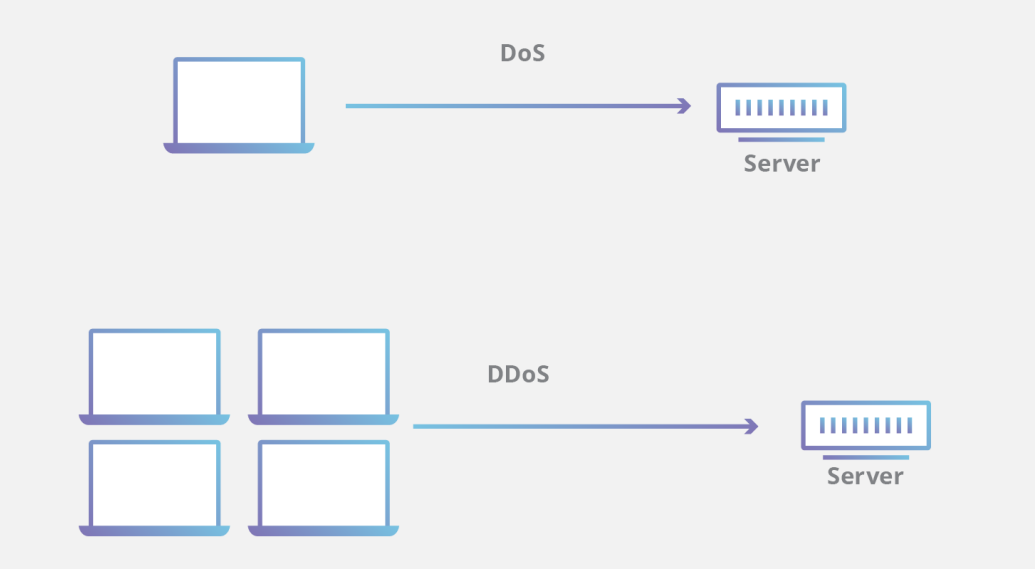

Figure 4.1: Illustration of DoS and DDoS attacks [8].

2. **Protocol Attacks**

   In protocol attacks, the attackers exploit weaknesses in network protocols. This type of attack consumes actual server resources or those of intermediate communication equipment, such as firewalls. The magnitude is measured in packets per second (Pps). Examples are **SYN floods** and **Ping of death**. In SYN floods, attackers send numerous TCP connection requests but never finalize the TCP handshake, leaving the server overwhelmed with incomplete connections. The Ping of Death involves sending oversized or malformed packets that can crash vulnerable systems.

3. **Application layer Attacks**

   Instead of targeting the network or the protocol layer, these attacks focus on overloading specific web applications and services. These attacks use seemingly legitimate and innocent requests to crash the web server. The magnitude is measured in requests per second (Rps). Examples include **HTTP floods** and **Slowloris**. HTTP flood, a large number of legitimate-looking requests are sent to a web server, consuming processing power and slowing performance. Slowloris keeps many connections to the server open with incomplete HTTP requests, blocking new incoming traffic.

4. **Reflective Attacks**

   In reflective attacks, the attacker sends requests to third-party servers with the victim's IP address, causing the servers to send massive responses to the victim unknowingly. Examples are **DNS reflection** and **NTP reflection**. In a DNS reflection attack, the attackers use open DNS resolvers to send large amounts of data to the target. In an NTP reflection attack, the attacker exploits Network Time Protocol (NTP) servers to generate amplified traffic.

5. **Resource Exhaustion Attacks**

These attacks focus on exhausting system resources, such as CPU, RAM, or disk space, rather than network bandwidth. Such as memory leaks or exploits of the ARP table.

4.2 DoS approaches on Ethereum

The prior section provided an overview of Denial-of-Service (DoS) attacks in general; however, Ethereum's distinct architecture and consensus approach make it vulnerable to specific threats. The open validator schedule and financial incentives introduce new potential attack targets for malicious actors. The following subsections delve into how attackers might strategically employ DoS methods to disrupt consensus, slow block creation, and alter transaction ordering within Ethereum's Proof-of-Stake mechanism.

Time-Sensitive Attacks

Attackers may perform targeted Denial-of-Service (DoS) attacks on specific validators scheduled to propose blocks. By flooding the network or the validator's infrastructure at critical times, the attacker can cause the validator to miss their slot, disrupting the consensus process. This strategy can be particularly advantageous if the attacker is the subsequent proposer, allowing them to capitalize on the missed block. This concept is discussed in the context of validator sniping, where validators attempt to „snipe“ block proposals from others to capture associated rewards [6].

MEV (Maximal Extractable Value) Exploits

Maximal Extractable Value (MEV) refers to the profit validators or miners can extract by reordering, including or excluding transactions within a block. Attackers can utilize DoS techniques to suppress competing validators, thus monopolizing the ability to order transactions to maximize their MEV extraction. For example, by DoS-ing validators that handle high-MEV transactions, an attacker can position themselves to capture these profits [21].

4.3 History of DoS on Ethereum

Since its inception, Ethereum has faced several Denial-of-Service attacks that have tested its resilience and significantly improved its protocol and security.

1. **The DAO Attack (June 2016)**

   In June 2016, the Decentralized Autonomous Organization (DAO), an Ethereum-based capital fund, suffered a major attack due to a vulnerability in its smart contract code. The attacker exploited this flaw to recursively withdraw funds, siphoning approximately 3.6 million ether. Although primarily a smart contract exploit, the incident led to network congestion, affecting Ethereum's performance. The community response was a contentious hard fork to reverse the theft, which resulted in the creation of Ethereum Classic (ETC) [2].

2. **Late 2016 DoS Attacks**

Despite the cost (in gas) required to mount the traditional DoS attack on the network, there was a string of attacks between late 2016 and early 2017. The attackers have been able to exploit bugs in blockchain clients to bypass the cost. Still, in particular, the developers for these clients and the Ethereum Foundation have been able to fix the problems quite rapidly [2].

3. **Eclipse Attack on Ethereum's Peer-to-Peer Network (2018)**

   In 2018, researchers demonstrated an Eclipse Attack on Ethereum's peer-to-peer network. In this scenario, an attacker isolates a node by monopolizing its peer connections, controlling the information it receives. This can lead to various exploits, including double spending and selfish mining. The Ethereum community responded by implementing changes to the peer selection process, making such attacks more difficult to execute [22].

4. **Speculative Denial-of-Service Attacks (2023)**

   In 2023, a study introduced new speculative DoS attacks on Ethereum, such as „ConditionalExhaust“ and „MemPurge.“ These attacks exploit the decoupling of computational work from transaction fees, allowing adversaries to burden nodes with excessive computations without proportional costs. The research highlighted the need for refined transaction fee mechanisms and resource metering to mitigate such vulnerabilities [34].

In Ethereum's PoS consensus mechanism, all upcoming block producers are publicly available. This means that attackers could identify which validators are supposed to propose a block and target them, resulting in the block not being proposed. Who is this beneficial for? An example could be a block proposer in the n+1 proposer slot attacking the current proposer in the slot n. Since the previous block was not proposed, the upcoming slot $n+1$ can take the transactions from the block that was not successfully proposed. Claiming the reward for their block and the prior block. [16].

There are multiple possible solutions to this problem. I hope to explore some of these in the coming parts.

4.4 Single secret leader election (SSLE)

Single Secret Leader Election refers to a class of cryptographic protocols designed to select a leader so that only the chosen participant learns of their selection and only at the appropriate time. These protocols ensure the process is private, fair, and verifiable, even in adversarial environments. Typically, each validator commits to some form of hidden identifier or secret input, and the protocol transforms these commitments, through cryptographic shuffling, homomorphic processing, or other privacy-preserving methods, into a result that reveals the leader to himself alone. This design minimizes the risk of targeted denial-of-service attacks, preserves unpredictability, and ensures proportional fairness (e.g., stake-weighted selection) without compromising participant privacy.

Before diving into specific SSLE protocols, I would like to introduce two key terms: vouchers and proofs.

In an SSLE protocol, a voucher is a publicly available output that all participants in the leader election agree upon. It serves as a shared reference that confirms the outcome of the election. On the other hand, proof refers to a private value the selected participant receives that enables them to verify their own selection. This proof allows the participant to recognize that it is their turn to propose a block while maintaining secrecy until they choose to reveal it.

Each SSLE protocol must satisfy the following properties [20]:

- **Uniqueness**

  All honest participants in the process of selecting the leader will agree on the same voucher. There is a specific participant whose proof can be successfully verified as legitimate. Additionally, even if an attacker knows this valid proof, the chances of creating a different proof that tricks another participant into accepting it as valid are extremely low. [20]

- **Fairness**

  The selection of the leader is based on the amount of stake they hold. More stake means a higher chance of being selected as a leader.

- **Unpredictability**

  Unless the proof is revealed, no one (including attackers) can predict the leader with better accuracy than their stake proportion allows.

- **Termination**

  If all honest participants follow the protocol, each of them will eventually receive a valid proof and a voucher.

These four properties collectively ensure that SSLE protocols provide robust, fair, and secure mechanisms for leader selection in blockchain consensus systems. *Uniqueness* prevents conflicting leader selections, maintaining consistency between all participants. *Fairness* promotes decentralization by ensuring that the chance for a participant to be elected is proportional to their stake. *Unpredictability* enhances security by preventing adversaries from determining the leader before they reveal themselves. Finally, *termination* guarantees that the process always reaches a valid result, preventing deadlocks or indefinite delays.

The following sections will examine different SSLE implementations and how they achieve these guarantees.

4.5 WHISK: A Practical Shuffle-Based SSLE Protocol for Ethereum

Whisk is an implementation of the SSLE protocol with the aim of enhancing the security and efficiency of leader selection, trying to plug the information leak that occurs during leader election. The primary objective of Whisk is to keep the identity of the leader secret until it is time for them to propose a block. Whisk is a modified version of the SSLE from DDH and the shuffle scheme from the paper „Single Secret Leader Election“ by Boneh et al. [3].

High-level overview

Firstly, the beacon chain picks a set of election candidates. For an entire day, the block proposers shuffle this list of candidates. After the shuffling has been completed, the final order of this list determines the block producers for the following day. During shuffling, we do not shuffle the block producers, but we randomize the commitments (see Section 4.5) that correspond to them, ensuring privacy and unlinkability. The election winners open their commitment to prove that they won the election. [3]

The selection of leaders using the WHISK protocol flows in specific events or phases; I would like to introduce them and go into more detail later. For this purpose, we will assume that the bootstrapping of the WHISK protocol has already been done.

The flow of the WHISK protocol:

1. **Candidate selection event**

   We select a set of candidates from the entire pool of validators.

2. **Shuffling phase**

   The selected candidates are shuffled and randomized. As illustrated in Figure 4.2.

3. **Cool-down phase**

   This phase exists because we need to wait for our randomness beacon, RANDAO, to build enough entropy, so its output is unpredictable.

4. **Proposer selection event**

   The block proposers for the next day are selected from the previously selected and shuffled set of candidates.

5. **Block proposal phase**

   The winners of the selection propose their blocks.

Commitment Scheme

For the shuffling to work correctly, a commitment scheme is in place. A scheme that allows a special type of cryptographic commitment that third parties can randomize while keeping it unlikable to the original version. However, the owner must still be able to recognize and prove ownership of their commitment. This commitment is tied to the owner‘s identity; this way, only they are allowed to open it. Finally, it supports zero-knowledge proofs (ZKPs), which can be opened and proved multiple times without exposing unnecessary information.

To achieve this, WHISK uses a tuple-based commitment scheme. The owner (Alice) commits to a long-term secret k using a tuple (rG, krG), known as **tracker**. Here, rG is a base commitment chosen at random, and krG ensures that the secret k is cryptographically embedded in the commitment. During the shuffling, another participant (Bob) can randomize Alice‘s tracker by multiplying both elements by a random secret z, creating a new tuple $(zrG, zkrG)$. This ensures that the new version of the tracker remains unlikable to the original version, and Alice can still recognize and prove ownership. How does she prove it? When it’s necessary, she can use **discrete logarithm proof (DLOG NIZK):** $k(zrG) = zkrG$. This way, k is not revealed, but Alice can prove that she knows the corresponding secret. Furthermore, to bind the commitment to Alice’s identity, she registers a deterministic commitment $com(k) = kG$, which is stored in Alice‘s validator record. This

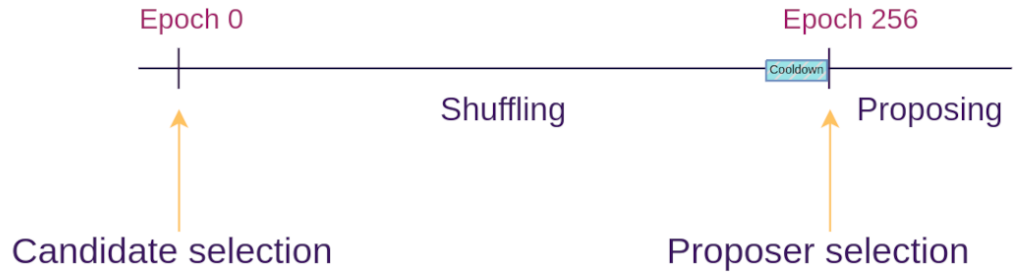

Figure 4.2: Illustration of the shuffling of block proposers for a whole day[9].

ensures that only she can open her commitment and that no two validators can use the same $\boldsymbol{k}$. [3, 9]

Protocol flow

As mentioned, the WHISK protocol is divided into events/phases. Now, we will dive deeper into WHISK to understand how proposing works and how candidates are selected. Let us assume that all validators have registered trackers to simplify this explanation.

The protocol starts with candidate selection, the beacon chain uses public randomness from RANDAO to select 16,384 (2^{14}) random trackers from the whole set of validators. The selected trackers are placed on the candidate list.

After the candidate list has been populated for the entire day (8192 slots), block proposers shuffle this candidate list, 128 trackers at a time, using their private randomness. How safe is this? The shuffling block proposers must also provide zero-knowledge proofs (ZKPs) that their shuffling has been performed honestly. For more information on how shuffling works, see section 4.5.

After the shuffling phase has finished (a day has passed), the proposer selection begins, RANDAO is used to populate a proposer list. The proposer list is an ordered list of 8,192 (2^{13}) winning trackers representing the proposers for the upcoming 8192 slots. The shuffling phase stops one epoch before the proposer selection phase. This is the cooldown phase; it exists, so malicious shufflers cannot manipulate the shuffling based on the upcoming RANDAO results.

Finally, for the next 8,192 slots (equivalent to a full day), we systematically assign the trackers from the proposer list to the Beacon Chain slots in sequential order.

This means that the proposal and shuffling phases last 8192 slots. As a result, Whisk operates in a continuous day-long pipeline, ensuring that by the time the current proposal phase concludes, the shuffling phase has already completed and finalized the 8,192 proposers for the following day [30].

Shuffling phase

I want to elaborate on how the shuffling works, as I have glossed over it previously. The shuffling of the candidate list of 16,384 trackers using only small shuffles, the size of 128 trackers at a time, is not an easy feat. Whisk uses an algorithm called the Feistel shuffle.

The Feistel shuffle lasts the entire shuffling phase of 8192 slots. The time is split into 64 rounds of 128 steps (slots) each. The set of 16,384 candidates is represented as a 128x128 shuffling matrix, each cell containing a number between 0 and 16,383.

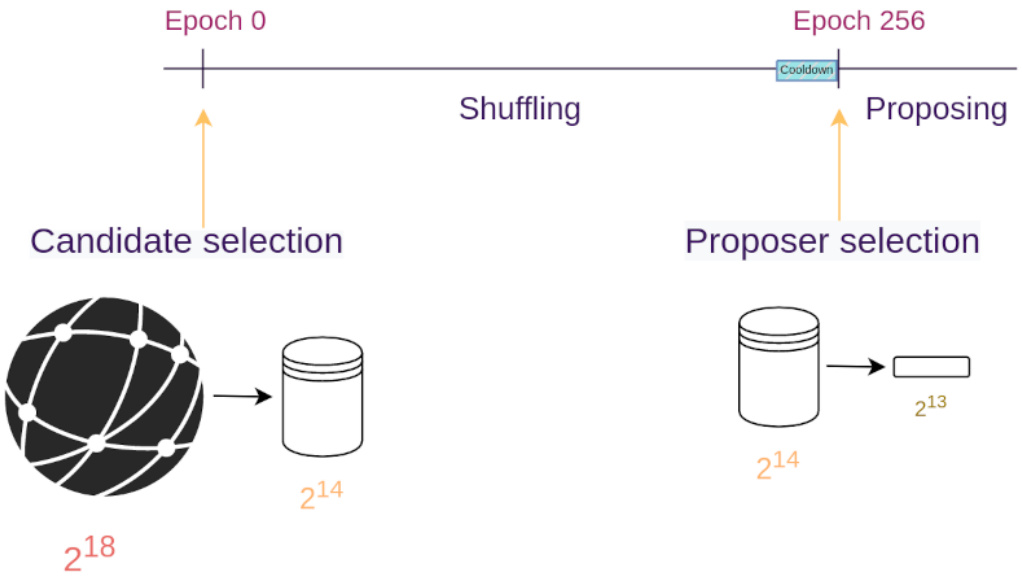

Figure 4.3: Illustration of WHISK protocol flow [30].

Every round, one row is shuffled (first round, first row). After 128 shuffles, a round is concluded, and a dispersion transformation is performed on the shuffling matrix.

This process involves treating each (x, y) coordinate in the matrix as plain text and modifying it through a single round Feistel mapping. This transformation rearranges each (x, y) coordinate into a corresponding (y, s) coordinate one-to-one. The F in question is a bijective non-linear mapping function; in the case of Feistel shuffle, it is $y \to y^3 \bmod 128$.

During the shuffling phase, the validators apply a permutation and randomization to each row of the shuffling matrix using private randomness.

To ensure the integrity of the shuffle, validators must provide zero-knowledge proofs demonstrating that they have performed the process honestly. These proofs verify that the output is a valid permutation of the input and that the original trackers have been properly randomized. Without these proofs, a malicious validator could manipulate the shuffle by replacing all trackers with their own or inserting invalid ones.

Additionally, to prevent adaptive shuffling attacks, validators are required to commit to their permutation in advance. When publishing a block, they must register their permutation commitment and are obligated to use that same permutation the next time they publish a block. Zero-knowledge proofs are again used to enforce compliance with this rule.

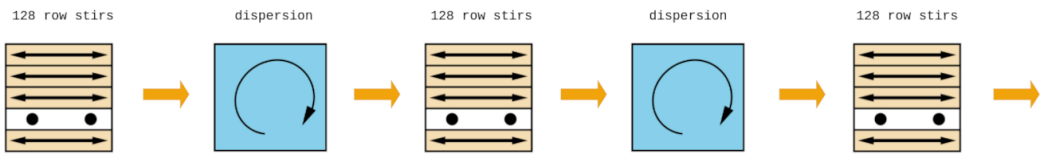

Figure 4.4: Illustration of the Feistelshuffle [30].

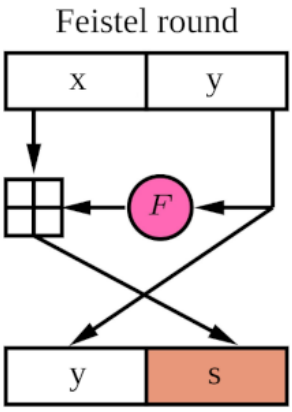

Figure 4.5: Illustration of the Feistel transformation [30].

Bootstrapping

Or how does one mount Whisk on a blockchain network? So far, I have assumed the Whisk protocol has been mounted and that all validators have trackers and commitments associated with them.

Once Whisk is bootstrapped, it starts in an insecure state where it is possible to predict future proposers. Why is this? Because, while bootstrapping Whisk, the beacon chain initializes all validators with dummy predictable commitments [30]. As blocks are proposed, validators can register a secure commitment for themselves. This way, the whisk‘s insecure state gradually shifts into a secure, steady state.

Overhead

How much would implementing the Whisk protocol cost? Let us look at the Whisk‘s overhead for 300k validators in terms of block and state size and computational overhead imposed on validators.

State size overhead

The overhead of Whisk in BeaconState is 45.5MB for 300k validators [9]. In detail:

- a candidate list -> 16,384 trackers $(16,384 * 96 = 1.57MB)$
- a proposer list -> 8,192 trackers $(8,192 * 96 = 0.78MB)$
- a tracker for each validator (96 bytes per validator) (28.8 MB for 300k validators)
- a $com(k)$ for each validator (48 bytes per validator) (14.4 MB for 300k validators)

Currently, the BeaconState is approximately 50MB in size [12]. Adding Whisk would roughly double the BeaconState size for 300k validators. In more detail, each validator currently adds 139 bytes to the state, but with Whisk it will add 283 bytes [9]. The good thing is that not all of these costs must be repeated in each round. The tracker and $com(k)$ of each validator (43.2MB for 300k validators) never change once they get registered. This, in turn, allows implementations to save space by keeping a reference to them in memory that can be used across multiple state objects. [9]

Block size overhead

The overhead of Whisk in the BeaconBlockBody is 16.5 kilobytes [9]. In detail:

- a list of shuffled trackers (128 trackers) ($128 * 96 = 12,288$ bytes)
- the shuffle proof ($4,272$ bytes)
- one fresh tracker (96 bytes)
- one $com(k)$ (48 bytes)
- a registration DLEQ proof (192 bytes)

The BeaconBlockBody contains data specific to the shuffling process. The data is extensive because it contains the shuffled trackers and cryptographic proofs.

The overhead of Whisk in the BeaconBlock is 192 bytes [9]. In detail:

- an opening DLEQ proof (192 bytes)

The BeaconBlock overhead contains only minimal proof-related data needed for block verification. It gets stored in the block header, ensuring that validators can verify the shuffle without needing all the details of the block body.

Computational overhead

The main part of the computational overhead for Whisk is proving and verifying the zero-knowledge proofs involved.

Based on a proof-of-concept (completed in January 2022 [9]) using an old version of arkworks-rs (zexe) and the BLS12-381 curve, the computational overhead for the average laptop is:

- Shuffling and proving the shuffle takes about 880ms (done by block proposers)
- Verifying the proof takes about 21ms (done by validators)

Since this is a PoC, a production-ready implementation should provide a 4x-10x performance boost.

4.6 Homomorphic Sortition Protocol

The homomorphic sorting protocol is an SSLE protocol with the same objective as the WHISK protocol. Select leaders at random and proportionally to stake, without revealing individual stake values or eligibility status until voluntarily disclosed. Homomorphic sorting is based on threshold fully homomorphic encryption (ThFHE) [20].

Threshold fully holomorphic encryption (ThFHE)

Threshold Fully Homomorphic Encryption (ThFHE) is a type of encryption that lets you do calculations on data without ever needing to see the original values. This means that sensitive information, like someone's stake in a system, can stay hidden while still being used in computations. ThFHE is special because it also splits the ability to decrypt the

final result among several participants. No one person can decrypt the data alone; instead, a group must work together to unlock the information. This adds a strong layer of security and decentralization. In the Homomorphic Sortition protocol, ThFHE allows participants to privately check if they've been selected as a leader, and only the selected one can later prove it. All this happens without exposing any private data.

High-level overview

The protocol evaluates a random x, then sums the total stake to normalize the stake of each participant into a range (scaled stake). It then selects the participant whose stake range includes x; this ensures that the higher your stake, the more likely you will be selected, but the process remains private and verifiable. The Homomorphic Sortition protocol assumes a few things as global knowledge. A stake distribution S, an encrypted random seed q, and a list T of encrypted random numbers, which are called tickets [20]. The protocol could be divided into 3 phases:

1. **Eligibility determination phase**

   In the first phase, each participant privately checks whether they've been selected as a leader for the current round. This is done by generating a shared random number and comparing it to their stake, which has been scaled relative to the total stake of the system, while keeping values encrypted. The result is that only the selected participant learns they've been chosen, without revealing their stake or the selection process to others.

2. **Leader Selection phase**

   Once a participant is privately selected, the protocol uses encrypted operations to extract their stake, ticket, and ID. These values are then used to generate a cryptographic proof (called a voucher) showing that the participant was fairly selected without revealing their identity or stake. Only the selected participant can produce this proof.

3. **Proof Sharing and Verification phase**

   The selected participant shares a partially decrypted version of their voucher with the network. Other participants verify these shares and combine them; once enough valid shares are collected, the voucher is fully decrypted, and the leader's selection can be publicly verified, without exposing private information.

Phase 1: Eligibility Determination

In this phase, a pseudorandom value x is derived from a public seed q and the current round r. Afterward, a vector U is created as a cumulative stake vector, where $U[i]$ represents the sum of all encrypted stakes from the first participant up to the i-th participant (i.e., $U[1] = S[1], U[2] = S[1] + S[2]$ and so on). This enables the protocol to later identify the selected participant based on a generated encrypted value, the x.

In addition to the cumulative vector, the sum of the total stake in the system is calculated, which is precisely the last element of the cumulative vector U, represented as m.

Thereafter, each participant's stake in the system is scaled (see Figure 4.6 picture (a)) in order to make their chance at becoming the leader proportional to their stake. The

scaling is done based on the formula $Z[i] = (S[i] * \delta)/m$. $S[i]$ is participants own stake and δ represents a fixed numeric constant defined by the system.

Finally, the protocol compares the random number x with each participant's scaled stake range (all encrypted) to determine which participant's range contains x (Figure 4.6, picture (b)) [20]. That participant is privately marked as the selected leader, but no one else can see this.

Phase 2: Leader Selection

In the second phase, the protocol uses the encrypted selection result of the previous step to determine which participant has been chosen as the leader. A new vector E is created, E being a binary vector, containing only zeros and ones, containing only a single one at the position corresponding to the selected participant and zeros elsewhere. This selection process is performed entirely under encryption, ensuring the leader's identity remains private.

Using the selection vector E, the protocol uses homomorphic operations to extract the encrypted stake of the participant S_r, ticket t_r, and the identifier i_r (Figure 4.6, picture (c)). These are used to create a special value called a voucher, which acts as a cryptographic receipt proving that the participant was fairly selected.

To create the voucher, the participant evaluates a pseudo-random function (PRF) using their ticket t_r and the round number r and then hashes the result with their ID i_r. This voucher v_r will be shared in the next phase as a selection proof (Figure 4.6, picture (d)). All values remain encrypted and private, just like in the previous phase.s

Phase 3: Proof Sharing and Verification

In the final phase, the selected participant reveals the proof of eligibility by sharing a partially decrypted version of their voucher v_r with the network. This is done in the form of a partial decryption, ensuring that no single participant can fully expose the voucher or impersonate the leader.

Upon receiving these partial decryptions, other participants verify each share for correctness. If valid, the share is added to a growing set of decryption shares Λ_r. Once enough valid shares have been collected, that is, when the total stake associated with the contributors exceeds the fault threshold of the system $s_f + 1$, the entire voucher v_r can be reconstructed and decrypted.

This decrypted voucher acts as a public, verifiable proof that the leader was selected correctly, without revealing the participant's stake, ticket, or internal random values. The process maintains privacy for all non-selected participants and ensures that only valid leaders are recognized.

Overhead and Limitations

While Homomorphic Sortition provides strong privacy guarantees through the use of threshold fully homomorphic encryption (4.6), it also introduces notable computational and communication overheads.

Looking at the communication overhead, in each round of the Homomorphic Sortition protocol, the only messages that participants exchange are partial decryptions of a cryptographic proof, a voucher, which serves as evidence that a participant was selected as the leader. These vouchers are compact, roughly equivalent to a few cryptographic hashes. The total volume of data exchanged in a round depends on the number of participants involved;

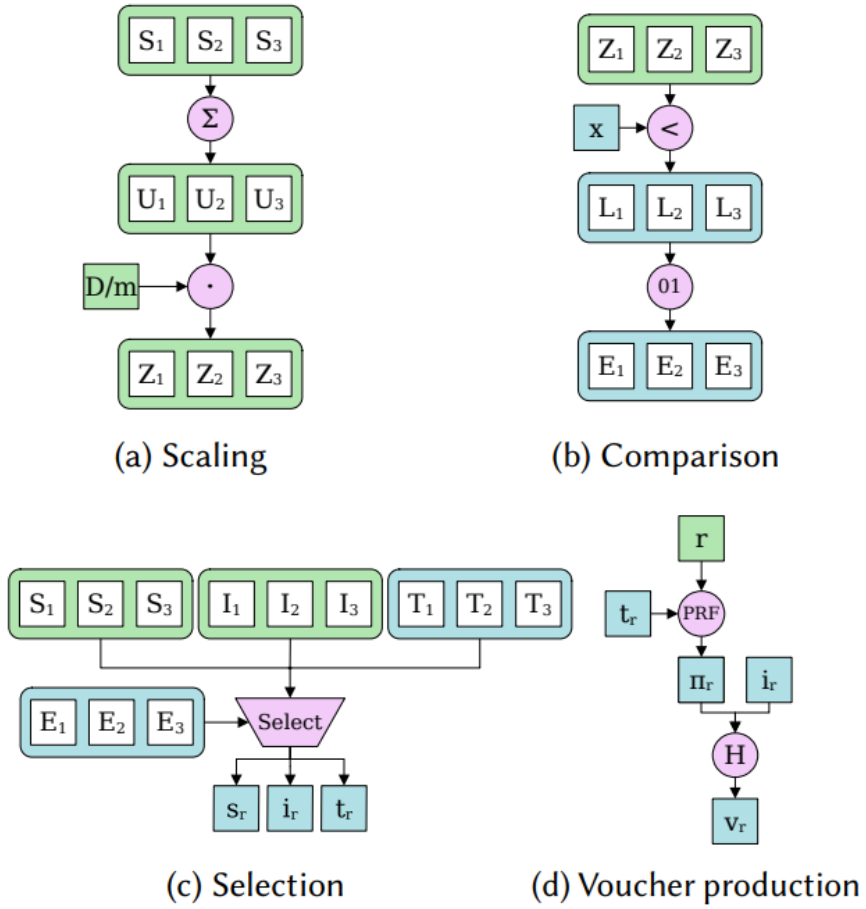

Figure 4.6: SSLE components of Homomorphic Sortition [20].

specifically, with n participants, the total communication load scales proportionally to n^2 multiplied by a small constant. Importantly, only a single round of message exchange is required per election, making the protocol relatively lightweight in terms of network usage. This design keeps communication overhead low and ensures that the protocol remains efficient even as the number of participants increases.

However, on the computational side, the costs are significantly higher. The protocol's performance depends heavily on the complexity of the underlying homomorphic circuits, which may involve thousands or even millions of logic gates. Although this approach preserves privacy by allowing computation on encrypted data, it introduces substantial computational overhead. The protocol must perform several intensive operations, such as comparing encrypted numbers, selecting values, evaluating pseudorandom functions (PRFs), and hashing, while the data remains encrypted. According to [20], a binary encryption approach like TFHE would require approximately n^2 operations, which scale poorly as the number of participants increases. Alternatively, arithmetic encryption schemes like BGV offer a more compact representation, but still involve a considerable computational depth, estimated at around 16 layers. Computational depth refers to the number of sequential encrypted multiplications that can be performed before the ciphertext becomes too noisy

to decrypt correctly and is a key factor in determining the feasibility and efficiency of FHE-based protocols.

Another critical aspect of the protocol performance is the overhead introduced by threshold decryption. Thanks to the authors of [4], who have developed a general approach to adding threshold functionality that allows splitting the secret key into a number n of shares for holomorphic schemes based on LWE (learning with errors), such that full decryption can only be performed when at least $t \leq n$ valid partial descriptions are combined, each using a separate key share[20]. According to the same study, the additional overhead caused by a threshold scheme arises during the setup and decryption phases. However, the efficiency of homomorphic operations remains unchanged by adding the threshold, meaning that the operations, which are the most expensive component, remain unaffected.

Chapter 5

Implementation of SSLE Protocols: Whisk and Homomorphic Sortition

This chapter presents the practical realization of the two analyzed Single Secret Leader Election (SSLE) protocols: Whisk and homomorphic sortition. It describes the architectural structure of the proof-of-concept implementations and the specific cryptographic components used. The goal is to demonstrate how these protocols can be translated from theoretical design into functional software systems capable of performing secure and privacy-preserving leader elections in decentralized networks.

5.1 Overview

These implementations aim to provide verifiable and privacy-preserving methods for leader election in decentralized systems, particularly in the context of Proof-of-Stake blockchains. Both protocols were implemented as proof-of-concept (PoC) systems focusing on cryptographic correctness, modularity, and extensibility.

The entire codebase is written primarily in C++, with some early components exposed to Python via Pybind11 for easier prototyping. Two major cryptographic libraries were used:

- `libsnark` (by SCIPR Lab) for zero-knowledge SNARKs in the Whisk protocol, using the pairing-friendly elliptic curve BN128.
- OpenFHE for fully homomorphic encryption (TFHE scheme) in the Sortition protocol.

Despite differences in the underlying primitives, both implementations share a common architectural approach.

- A central protocol runner simulates a round of the election.
- Cryptographic circuits or gadgets are written as reusable C++ modules.
- Timing and performance measurements are collected using scoped timers to evaluate feasibility.

Each protocol focuses on a different trust model:

- Whisk emphasizes the public verifiability of correct shuffling using SNARK proofs.

- Homomorphic Sortition emphasizes input privacy, with only the winner revealed and all other information encrypted throughout the process.

The following sections should provide a more detailed insight into each protocol's implementation.

5.2 Whisk Protocol Implementation

The Whisk protocol enables a verifiable shuffle of encrypted leader election credentials, ensuring unlinkability and correctness through zero-knowledge proofs. The developed PoC implementation reflects the specification from the original SSLE paper, with all core components being implemented in C++ using the libsnark and libff libraries. The protocol was modularized into components that handle the creation of trackers, proof of ownership, secure shuffle, and proof verification.

Tracker Format and Commitments

Each participant generates a tracker represented by a pair of elliptic curve points (rG, krG), where:

- r is a secret nonce for randomization
- k is the participant's secret key
- G is the group generator on the BN128 curve

This tracker is bound to a commitment kG, stored separately to verify identity ownership without revealing k. The use of this tuple ensures both identity-binding and unlinkable shuffling.

DLEQ Proof of Ownership

To prove ownership of a tracker, each participant constructs a Discrete Logarithm Equality (DLEQ) proof, demonstrating knowledge of the same secret scalar k in both public keys kG and krG, without revealing the scalar itself. This zero-knowledge proof is realized as a non-interactive *Sigma protocol* [1], using the *Fiat–Shamir heuristic* [2] to derive a challenge deterministically from a hash of public inputs. The protocol structure is based on the identification scheme described in Section 2 of Schnorr's work [28]. It follows the DLEQ construction introduced by Chaum and Pedersen to prove the discrete logarithm equality between group elements with respect to different generators [7]. The correctness of this Sigma protocol structure and its application in blind signature proofs involving equal discrete logarithms across different generators is formally shown in Proposition 3.2 of the Chaum–Pedersen construction [7].

[1] A Sigma protocol is a 3-move interactive proof where a prover demonstrates knowledge of a secret without revealing it: commitment, challenge, and response.

[2] The Fiat–Shamir heuristic transforms an interactive proof into a non-interactive one by replacing the verifier's random challenge with a hash of the public input and commitment.

Tracker Shuffle using SNARKs

The core of the Whisk protocol lies in securely and privately shuffling the list of trackers. A SNARK proof is used to guarantee that:

- The shuffled list is a permutation of the input list
- Each output tracker is a scaled version of its corresponding input (i.e., re-randomized as $(zrG, zkrG)$)

This is accomplished by modeling the shuffle as a row-wise transformation, where each row contains 128 tracker tuples. The shuffle proof circuit is constructed using libsnark, and a verifier function validates the proof.

Circuit Design: Permutation and Scaling Gadgets

Two key gadgets were designed and implemented:

- **Permutation Gadget:** Enforces that each output row is a permuted version of the input, using binary indicator variables and permutation matrices
- **Scaling Gadget:** Ensures that for each pair (rG_i, krG_i) and (rG'_j, krG'_j), there exists a scalar z such that $rG'_j = zrG_i$ and $krG'_j = zkrG_i$

Integration and Testing

The WHISK protocol was implemented as a standalone runner orchestrating all major phases of the protocol. Figure 5.1 illustrates the whole pipeline, from tracker creation to block proposal.

1. Generate trackers for all participants
2. Produce and verify ownership proofs (DLEQ)
3. Perform 64 rounds of tracker shuffling
4. Generate and verify SNARK proofs for each shuffle round
5. Select proposers and confirm ownership using DLEQ proofs

Each stage was instrumented with timers to evaluate performance. DLEQ proofs were generated for every tracker and verified for each proposed block. SNARK proofs were constructed and verified for each row-level shuffle.

On average, DLEQ proofs were generated in 2.15 ms and verified in 0.85 ms. SNARK proof generation and verification each required approximately 2.15 ms per row, while SNARK key generation took around 100 ms per round. These results confirm that the main computational overhead lies in the shuffle phase.

Table 5.1 summarizes the average cost of the core cryptographic operations. Due to the fixed circuit structure, the SNARK-related costs remain constant per shuffle row. The linear scaling occurs only in the number of rounds or rows processed.

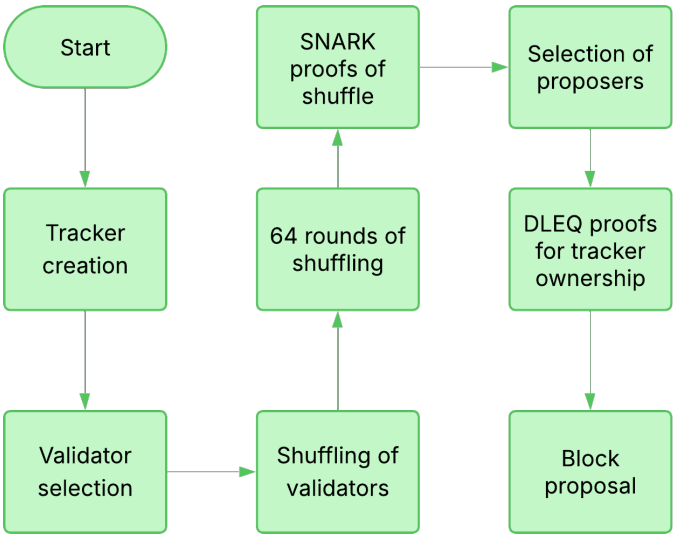

Figure 5.1: Illustration of the WHISK protocol pipeline.

Table 5.1: WHISK Protocol Performance Summary (per-row shuffle)

Component	**Average Time (s)**
DLEQ Proof Generation	0.002153
DLEQ Verification	0.000846
SNARK Keygen (per row)	0.100637
SNARK Proof Generation (per row)	0.002153
SNARK Verification (per row)	0.000846
Total DLEQ Proofs	305,663
Total SNARK Calls (Rows Shuffled)	8192

5.3 Homomorphic Sortition Protocol Implementation

The homomorphic sortition protocol was implemented as a standalone C++ module, complementing the SNARK-based Whisk system by addressing the same problem through an entirely different cryptographic framework. This approach leverages the fully homomorphic TFHE encryption scheme to enable secure computation over encrypted data, preserving privacy throughout the selection process.

At a high level, the protocol homomorphically computes a pseudorandom function (PRF) output, which is then compared against scaled stake thresholds to determine the winning participant. All comparisons and conditional selections are performed using TFHE logic gates, ensuring the intermediate values remain encrypted and private. This section describes the key components of the implementation, with emphasis placed on performance-focused simplifications, such as limiting the ciphertext domain to 64-bit values and minimizing the circuit depth to ensure feasible execution times.

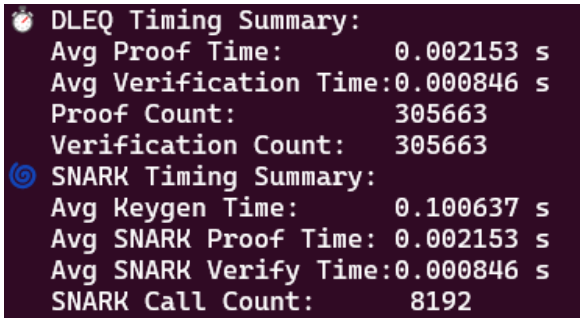

Figure 5.2: Performance output showing WHISK protocol.

Stake Representation and Scaling

Each participant is initialized with a known plaintext stake value, which defines their selection weight. These values are used to compute a cumulative stake array, which is then scaled into a fixed-size numeric domain that matches the PRF output range. The scaled thresholds are calculated as cumulative proportions of the total stake, scaled by a configurable constant δ. The resulting array Z is used to define subinterval boundaries for each participant. During the protocol round, the homomorphically computed PRF output is compared against these boundaries to determine which participant owns the corresponding subinterval. This approach enables proportional selection based on stakes without encrypting the stakes themselves.

PRF and Encrypted Participant Selection

The protocol computes a pseudorandom output using a Feistel-style PRF evaluated homomorphically over encrypted inputs to determine the leader in each round. The PRF takes as input the encrypted round number and a global seed shared by all participants, producing a 128-bit encrypted output vector. This output is then homomorphically compared with the scaled cumulative stake boundaries using the *tfheLessThan* circuit. Each comparison yields an encrypted bit that indicates whether the PRF output falls below the participant's threshold. The results are stored in a vector L representing a masked selection signal. To identify the first participant for whom the condition $x < z_i$ is true, the *tfheFirstOne* circuit is applied to L. This produces a one-hot encrypted selection vector E, which activates only the winning index. This method ensures that selection is proportional to stake, computed entirely over encrypted data, and reveals no information about non-selected participants.

Winner Extraction and Voucher Generation

Once the encrypted selection vector E is computed, it is used to extract the encrypted data of the selected participant. This includes their ticket, index, and stake, all represented as 64-bit ciphertext vectors. Conditional selection is performed using the *csel* circuit, which takes E and a list of encrypted values (e.g., tickets) and returns only the one corresponding to the active index. After selection, the ticket and the winner's index are decrypted locally to finalize the round result. The ticket is then combined with the encrypted round number to calculate a new PRF output, called π_r, which serves as a private proof of selection. To allow others to verify the result, a final voucher is constructed by hashing the encrypted PRF output π_r with the winner's index using the *tfheHash* function. The result is an

encrypted digest v_r, which acts as a verifiable commitment to the selected participant and the randomness used in the round. This voucher can be verified by recomputation, ensuring the integrity of the selection without revealing the underlying PRF value or compromising privacy.

Decryption and Verification

The protocol proceeds with the verification phase after selecting the winning participant and generating the voucher v_r. The chosen participant locally decrypts the encrypted ticket and index using their private key, allowing them to recompute the PRF output $\pi_r = PRF(ticket, round)$. This allows them to confirm their selection without revealing any sensitive information.

To enable public verification, the participant broadcasts the encrypted voucher v_r and π_r, along with their encrypted index. Other participants use these to independently recompute the hash $H(\pi_r||index)$ and compare it against the claimed voucher. The *verify_voucher* function performs this verification by concatenating π_r and the encrypted index, then applying the same *tfheHash* function used during selection.

Participants who successfully validate the voucher contribute confirmations. Once a threshold number of confirmations is reached (for example, $t > N/3 + 1$), the round is considered completed. This mechanism provides strong consistency and soundness guarantees without requiring decryption of any intermediate values, preserving privacy while ensuring public verifiability.

Integration and Testing

The protocol was orchestrated in a standalone C++ runner that:

1. Encrypts the number and seed of the round and evaluates a homomorphic PRF.
2. Scales plaintext stake values to cumulative thresholds.
3. Compares the encrypted PRF output to these thresholds using *tfheLessThan*.
4. Identifies the selected participant via *tfheFirstOne*.
5. Extracts the winner's encrypted ticket, index, and stake using the *csel* circuit.
6. Recomputes a PRF using the ticket and round, and hashes the result to produce a verifiable voucher.
7. Decrypts the winner's values and verifies correctness using the *verify_voucher* function.

Tables 5.2 and 5.3 summarize the measured execution time of a single round of the Sortition protocol with 20 and 100 participants, respectively. The most computationally expensive components are the encrypted comparison (`tfheLessThan`) and conditional selection (`csel`) circuits. Although both depend on the number of participants, the scaling is not strictly linear. For example, increasing the participant count by a factor of five led to a roughly 4.7× increase in `tfheLessThan` time, and only a 1.9× increase in `csel`, indicating opportunities for optimization.

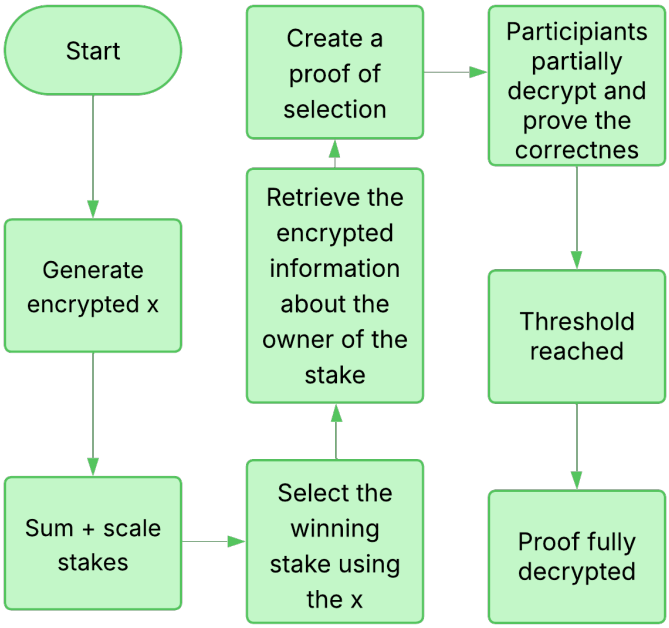

Figure 5.3: Illustration of the Sortition protocol pipeline.

The PRF and hash circuits (`tfhePRF` and `tfheHash`) also contribute significantly to the total runtime, but their scaling is less tightly coupled to the count of participants. The `tfheFirstOne` circuit runs only once per round and remains constant in cost.

The total protocol runtime increased from 1386.837 s to 4897.942 s as the participant count rose from 20 to 100. This suggests that while the protocol scales with the size of the validator set, not all components grow proportionally, and some exhibit sublinear behavior due to reused computation or parallelism.

While the protocol demonstrates functional correctness and verifiability, it exhibits a steep scaling overhead as the participant count increases. Optimizations such as circuit batching, parallelization, or approximate selection could reduce this cost in future implementations. Timing instrumentation was added via scoped timers to evaluate the feasibility of each step.

Table 5.2: Performance Summary of a Sortition Round (20 Participants)

Component	**Total Time (s)**	**Call Count**
`tfhePRF` (seed + ticket)	267.846	2
`tfheLessThan`	399.040	20
`tfheFirstOne`	1.710	1
`csel` (index, ticket, stake)	407.866	192
`tfheHash`	189.124	20
Total Round Time	**1386.837**	–

Table 5.3: Sortition Protocol Performance Summary (100 Participants)

Component	Total Time (s)	Call Count
`tfhePRF` (seed + ticket)	263.115	2
`tfheLessThan`	1875.249	100
`tfheFirstOne`	8.736	1
`csel` (index, ticket, stake)	789.628	192
`tfheHash`	201.335	100
Total Protocol Time	**4897.942**	–

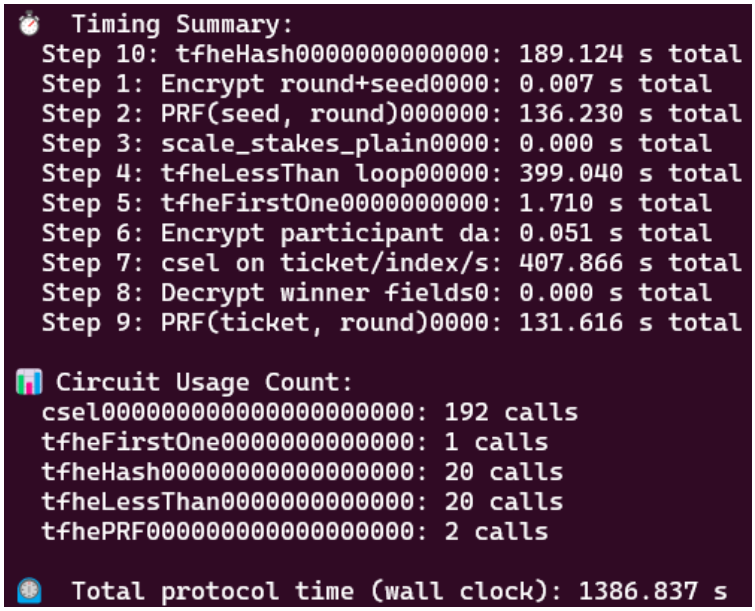

Figure 5.4: Performance output of the Sortition protocol (20 participants).

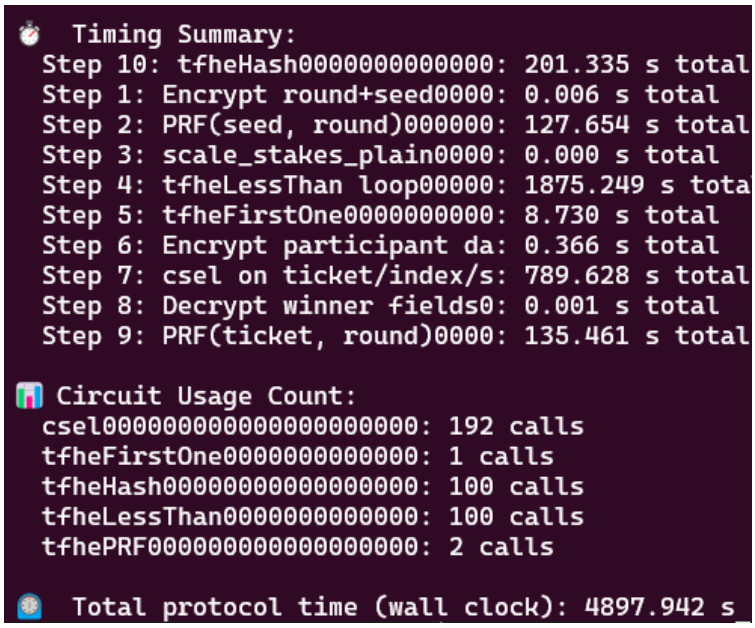

Figure 5.5: Performance output of the Sortition protocol (100 participants).

Chapter 6

Discussion and Comparative Analysis

This chapter summarizes the design trade-offs, strengths, and weaknesses of the two Single Secret Leader Election (SSLE) protocols explored in this thesis: **WHISK** and **Homomorphic Sortition**. It also discusses implementation limitations and outlines directions for future work.

6.1 Protocol Trade-offs and Suitability

The two protocols represent opposite ends of the privacy–performance spectrum. WHISK prioritizes efficiency and real-world applicability through verifiable SNARK-based shuffling, while Homomorphic Sortition offers stronger privacy guarantees using encrypted logic, albeit at significantly higher computational cost.

6.2 Protocol Comparison and Evaluation

Cryptographic Assumptions:

WHISK is built upon elliptic curve cryptography (ECC) and zero-knowledge SNARKs, assuming the hardness of discrete logarithms and the soundness of zk-SNARK constructions (e.g., Groth16 over pairing-friendly curves like BN128). In contrast, Homomorphic Sortition is implemented using the TFHE encryption scheme, which enables bit-level logic operations on encrypted data. TFHE is grounded in lattice-based cryptographic principles and is considered post-quantum secure. However, this implementation does not explicitly construct or rely on the underlying Learning With Errors (LWE) problem. While TFHE allows for strong privacy guarantees, it introduces significantly higher computational overhead than primitives based on elliptic curves.

Privacy Guarantees:

Whisk provides unlinkability through shuffled trackers and SNARK proofs and selective disclosure using DLEQ proofs, but relies on publicly shuffled and committed credentials. On the other hand, Homomorphic Sortition operates entirely over encrypted values, ensuring full input and output privacy until the final decryption phase. Only the selected participant is revealed; no intermediate comparison results or stake information is leaked.

Table 6.1: Qualitative Comparison of WHISK and Homomorphic Sortition

Feature	WHISK	Homomorphic Sortition
Privacy Guarantee	High (shuffle-based unlinkability)	Very High (full FHE-based confidentiality)
Identity Revelation	Only when block is proposed	Only after voucher decryption
Proof Mechanism	SNARKs (libsnark, Groth16)	Threshold Fully Homomorphic Encryption (TFHE)
Computation Cost	Moderate (2.15 ms per row)	Very High (4900 s for 100 participants)
Communication Overhead	Moderate (proofs per shuffle)	Medium (encrypted voucher and PRF broadcast)
Ethereum Compatibility	Ready for zk-rollups or beacon chain integration	Currently not feasible on-chain; suitable for off-chain or post-quantum contexts
Post-Quantum Security	No (based on elliptic curves)	Yes (based on lattice assumptions)

Performance and Efficiency:

Whisk benefits from SNARK proofs that allow for fast public verification and small footprints on the chain. Its performance bottleneck lies in proof generation, which is computationally intensive but efficient when reused across multiple rounds. Homomorphic Sortition trades off succinctness for privacy: TFHE operations are slower, and circuit depth must be minimized to remain tractable. However, TFHE allows for constant-depth logic and avoids a trusted setup.

In summary, Whisk is more practical in systems prioritizing verifiability and performance with moderate privacy, while Homomorphic Sortition is suited to applications where maximum privacy and post-quantum security outweigh computational cost.

Applicability to Ethereum

WHISK is well aligned with Ethereum's current zk-friendly ecosystem. It enables verifiable shuffling of proposer identities off-chain, with small on-chain proof commitments. If WHISK were deployed in a leader election or committee-selection layer of Ethereum (e.g., via L2 or Beacon Chain), the main trade-off would be slight time delays during SNARK proof generation and the need for circuit commitment verification on-chain. These delays are predictable and small (under 10ms per row), making integration feasible.

In contrast, homomorphic sortition is currently not compatible with the Ethereum execution model. TFHE-based logic is not natively verifiable on-chain, and the ciphertext size and computational cost would exceed the gas limits. However, it remains a strong candidate for future post-quantum systems or high-security, off-chain coordination layers. With slight adjustments and the implementation of SLP as described in [20], it could be possible, which would require further studies.

6.3 Key Findings

- WHISK delivers fast, publicly verifiable leader election with partial privacy and efficient scaling. Its reliance on fixed-size SNARK circuits makes performance predictable.

- Homomorphic Sortition achieves complete confidentiality of inputs and comparisons, but suffers from heavy computational overheads that limit its practical deployment today.
- In terms of cryptographic security, WHISK relies on classical assumptions (discrete logs) and SNARK proofs, while Sortition operates with fully homomorphic encryption, offering stronger future-proofing capabilities.

6.4 Limitations and Future Work

While both implemented protocols successfully demonstrate key design principles for Single Secret Leader Election, they also carry specific limitations that open opportunities for future improvements.

- The Sortition implementation used 64-bit scaled domains for stake thresholds and PRF output instead of the full 128-bit as specified in the original paper.
- TFHE execution remains slow on CPU, limiting Sortition's throughput to experimental use or offline rounds.

In terms of future work, several directions could improve both protocols and bring them closer to practical deployment. For WHISK, a natural next step would be integrating the protocol into a simulated blockchain environment or testnet to evaluate end-to-end behavior and on-chain verification costs. While the current implementation demonstrates feasibility, further improvements could enhance flexibility and trust assumptions.

In contrast, Homomorphic Sortition requires more substantial optimization. The current proof of concept confirms that fully encrypted leader selection is possible, but the computational cost remains prohibitively high for real-time or large-scale use. Future improvements could involve circuit-level optimizations, transitioning from the TFHE scheme to more performance-efficient alternatives such as the BGV scheme, and leveraging hardware acceleration. Additionally, implementing Secret Leader Permutation (SLP), as described in the original proposal by Freitas et al. [20], would significantly extend the functionality by enabling the election of non-repeating leaders for multiple consecutive rounds.

Finally, both protocols could benefit from more extensive benchmarking under varying network conditions and larger participant sets. Combining techniques from both paradigms, succinct ZK proofs with partially encrypted computation, may also lead to hybrid designs that balance performance, verifiability, and confidentiality more effectively.

Chapter 7

Conclusion

This thesis has presented the design, implementation, and comparative evaluation of two distinct single secret leader election (SSLE) protocols. **WHISK** and **Homomorphic Sortition**. Each approach provides privacy-preserving leader selection for decentralized systems, but is grounded in fundamentally different cryptographic paradigms: SNARKs based on elliptic curves versus lattice-based homomorphic encryption.

7.1 Summary of Contributions

The main contributions of my thesis are as follows.

- Proof of concept, a modular C++ implementation of the Whisk protocol, including tracker generation, DLEQ proof of ownership, verifiable shuffling using SNARKs (via *libsnark*), and public proof verification.
- Proof of concept, a fully encrypted implementation of the Homomorphic Sortition protocol, using the TFHE scheme to execute all core operations: stake comparisons, selection logic, and PRF evaluation over ciphertexts.

  To the best of our knowledge, this is the first working implementation of the Homomorphic Sortition protocol as described in [20], realized using TFHE for circuit-level privacy-preserving leader selection.
- A side-by-side analysis of both protocols regarding cryptographic assumptions, privacy guarantees, and performance trade-offs.

7.2 Closing Remarks

This thesis has explored two fundamentally different approaches to Single Secret Leader Election (SSLE), each grounded in a unique cryptographic paradigm. The SNARK-based WHISK protocol demonstrates that efficient, publicly verifiable leader election is feasible within the constraints of existing blockchain ecosystems such as Ethereum. In contrast, the TFHE-based Homomorphic Sortition protocol pushes the boundaries of privacy, enabling fully encrypted selection logic at the cost of significantly higher computational overhead.

These two approaches reflect a broader design spectrum: one optimized for performance and verifiability, the other for maximal confidentiality and post-quantum resilience. While

WHISK is ready for integration in real-world systems, Homomorphic Sortition is a prototype for what may become necessary in future privacy-critical or quantum-aware environments.

As the need for secure, transparent, and privacy-preserving coordination grows, SSLE protocols will remain a vital building block for decentralized consensus.

Bibliography

[1] Belotti, M.; Bozic, N.; Pujolle, G. and Secci, S. A Vademecum on Blockchain Technologies: When, Which and How. *IEEE Communications Surveys & Tutorials*, july 2019, PP, p. 1–1.

[2] Bennett, E. *History of Denial-of-Service (DoS) Attacks on Ethereum.* 2024. Available at: https://gist.github.com/ethanbennett/7396bf3f61dd985d3426f2ee184d8822.

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

[4] Boneh, D.; Gennaro, R.; Goldfeder, S.; Jain, A.; Kim, S. et al. *Threshold Cryptosystems From Threshold Fully Homomorphic Encryption.* 2017. Available at: https://eprint.iacr.org/2017/956. https://eprint.iacr.org/2017/956.

[5] Buterin, V. *The Beige Paper: A Simplified Technical Description of Ethereum* https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf. 2018.

[6] Bürgel, S. Proof-of-Stake Validator Sniping Research. *HOPR*, March 2021. Available at: https://medium.com/hoprnet/proof-of-stake-validator-sniping-research-8670c4a88a1c.

[7] Chaum, D. and Pedersen, T. P. Wallet databases with observers. In: Springer. *Annual International Cryptology Conference.* 1992, p. 89–105.

[8] Cloudflare. *What is a Denial-of-Service (DoS) attack?* 2024. Available at: https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/.

[9] Community, E. R. *WHISK: A Practical Shuffle-Based SSLE Protocol for Ethereum.* 2022. Available at: https://ethresear.ch/t/whisk-a-practical-shuffle-based-ssle-protocol-for-ethereum/11763.

[10] Ethereum Foundation. *Accounts | ethereum.org.* 2023. Available at: https://ethereum.org/en/developers/docs/accounts/.

[11] Ethereum Foundation. *Transactions.* 2023. Available at: https://ethereum.org/en/developers/docs/transactions/.

[12] Ethereum Stack Exchange. *How are BeaconBlocks and BeaconStates interconnected?* 2022. Available at: https://ethereum.stackexchange.com/questions/136406/how-are-beaconblocks-and-beaconstates-interconnected. Accessed: 2025-03-13.

[13] Foundation, E. *Ethereum: Now Going Public* https://blog.ethereum.org/2014/01/23/ethereum-now-going-public. 2014.

[14] Foundation, E. *Proof-of-Stake (PoS) Consensus Mechanism*. 2023. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[15] Foundation, E. *Ethereum Gas and Fees*. 2024. Available at: https://ethereum.org/en/gas/.

[16] Foundation, E. *Ethereum Roadmap: Secret Leader Election*. 2024. Available at: https://ethereum.org/en/roadmap/secret-leader-election/.

[17] Foundation, E. *What are Smart Contracts?* https://ethereum.org/en/smart-contracts/. 2024.

[18] Foundation, E. *Ethereum Developer Documentation* https://ethereum.org/en/developers/docs/. N.d.

[19] Foundation, E. *Ethereum Virtual Machine (EVM)* https://ethereum.org/en/developers/docs/evm/. N.d.

[20] 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, Paper 2023/113. 2023. Available at: https://eprint.iacr.org/2023/113.

[21] Heimbach, L.; Vonlanthen, Y.; Villacis, J.; Kiffer, L. and Wattenhofer, R. Deanonymizing Ethereum Validators: The P2P Network Has a Privacy Issue. *ArXiv preprint arXiv:2409.04366v1*, 2024. Available at: https://arxiv.org/pdf/2409.04366v1.pdf.

[22] Marcus, Y.; Heilman, E. and Goldberg, S. *Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network* Cryptology ePrint Archive, Paper 2018/236. 2018. Available at: https://eprint.iacr.org/2018/236.

[23] Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System* https://bitcoin.org/bitcoin.pdf. 2008.

[24] (NCSC), N. C. S. C. *Denial of Service (DoS) Guidance Collection*. 2024. Available at: https://www.ncsc.gov.uk/collection/denial-service-dos-guidance-collection.

[25] Popa, R. A. *Merkle Trees: CS 261 Computer Security Fall 2015 Lecture Notes*. 2015. Available at: https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf.

[26] Rajasekaran, A. S.; Azees, M. and Al Turjman, F. A comprehensive survey on blockchain technology. *Sustainable Energy Technologies and Assessments*, 2022, vol. 52, p. 102039. ISSN 2213-1388. Available at: https://www.sciencedirect.com/science/article/pii/S2213138822000911.

[27] Sachdeva, S.; Fatehaj, L.; Tan, S.; James, J.; Ajayi, O. et al. A Blockchain based Framework for Secure and Decentralized Energy Trading in a Community. In: *2023 IEEE International Conference on Industrial Technology (ICIT)*. 2023, p. 1–8.

[28] Schnorr, C.-P. Efficient identification and signatures for smart cards. In: *Advances in Cryptology—CRYPTO'89*. Springer, 1991, p. 239–252.

[29] Szabo, N. *Smart Contracts* https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. 1997.

[30] Toulme, A.; Drake, J.; Feist, D.; Herold, G.; Khovratovich, D. et al. *Whisk: A Practical Shuffle-Based SSLE Protocol for Ethereum* https://hackmd.io/@asn-d6/HyD3Yjp2Y. 2020.

[31] Wood, G. *Ethereum: A Secure Decentralised Generalised Transaction Ledger* https://ethereum.github.io/yellowpaper/paper.pdf. Ethereum Foundation, 2014.

[32] Wood, G. *Ethereum: A Secure Decentralised Generalised Transaction Ledger, EIP-150 Revision* https://ethereum.github.io/yellowpaper/paper.pdf. 2014.

[33] Yaga, D.; Mell, P.; Roby, N. and Scarfone, K. Blockchain Technology Overview. *CoRR*, 2019, abs/1906.11078. Available at: http://arxiv.org/abs/1906.11078.

[34] Yaish, A.; Qin, K.; Zhou, L.; Zohar, A. and Gervais, A. *Speculative Denial-of-Service Attacks in Ethereum* Cryptology ePrint Archive, Paper 2023/956. 2023. Available at: https://eprint.iacr.org/2023/956.

Appendix A

Code Structure of Whisk and Homomorphic Sortition Implementations

The source code for this thesis includes two main protocol implementations: **Whisk** (SNARK-based) and **Homomorphic Sortition** (TFHE-based). Both C++ implementations are located under the `native/` directory, which also contains third-party libraries. The project structure is as follows:

`native/whisk/` — C++ implementation of the Whisk protocol using libsnark:

- `shuffle_proof.cpp/.hpp` — SNARK circuit for proving correct row-level shuffling of trackers.
- `permutation_gadget.hpp`, `scaling_gadget.hpp`, `row_shuffle_gadget.hpp` — Constraint gadgets for permutation, scaling, and shuffle SNARK logic.
- `dleq_proof.cpp/.hpp` — Discrete log equality proof for tracker ownership.
- `tracker.cpp/.hpp` — Tracker structure and associated cryptographic helpers.
- `utils.cpp/.hpp` — Utility functions such as random generation and hashing.
- `bindings.cpp` — Pybind11 bindings exposing SNARK-related functions to Python.
- `tests/` — Unit tests for SNARK components.

`whisk/` (Python) — Python orchestration layer for the Whisk protocol:

- `protocol.py` — High-level logic and execution flow of the Whisk protocol.
- `run_simulation.py` — Entrypoint script to launch protocol simulations.
- `feistelshuffle.py` — Implements the Feistel-based row shuffle used before SNARK proving.
- `participant.py`, `beacon.py`, `block.py` — Models for participants, beacon chain, and blocks.
- `timing.py` — Performance logging for various phases of the protocol.

`native/sortition/` — C++ implementation of the Homomorphic Sortition protocol using TFHE (OpenFHE):

- `circuits.cpp/.hpp` — Implements homomorphic circuits: comparator, selector, PRF, first-one detection, and hash.
- `protocol.cpp/.hpp` — Core logic for encrypted PRF evaluation, comparison, and winner selection.
- `participant-fhe.cpp`, `main.cpp` — Participant logic and the main execution entry point.
- `timing.hpp` — Scoped timers and timing collector for benchmarking.
- `tests/` - Test suite to verify the correctness of the TFHE circuits.

`common/` — Shared Python utilities (currently used primarily by Whisk):

- `network.py` — Utilities for simulating peer communication and participant identifiers.
- `utils.py` — Common helpers like bit-level operations and formatting.

`native/` — Also includes third-party cryptographic libraries and tools:

- `libsnark/`, `openfhe-development/`, `pybind11/` — Cloned or submodule-linked external dependencies.
- `build/` — CMake build directory.

Each implementation is modular, testable, and supports standalone execution. The Python and C++ components are tightly integrated via bindings to enable flexible simulation and benchmarking.