Analysis of the BNB Smart Chain Consensus Mechanism

Bachelor Thesis

Daniele De Jeso
from
Bern, Switzerland

Faculty of Science, University of Bern

June 18, 2025

Prof. Christian Cachin
Michael Senn
François-Xavier Wicht
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Binance, one of the largest crypto exchanges, and its blockchain, the BNB Smart Chain (BSC), are among the most successful blockchain platforms of recent years. At the center of its functionality is the consensus mechanism Proof of Staked Authority (PoSA),a hybrid approach that integrates proven concepts from the established consensus mechanisms Proof of Stake and Proof of Authority.
This bachelor's thesis contains a theoretical introduction to distributed systems, consensus, cryptographic methods, and technical details of the BSC and its architecture. It is followed by a source code-based analysis of the block production process, PoSA protocol, involved system components, and the underlying security assumptions. The analysis uses pseudo code that follows the logic of the Go programming code used in production. Particular attention is paid to selecting validators and the underlying deterministic shuffling algorithm. The results show how the BSC fast finalization is achieved but also expose risks in terms of fairness in validator selection and decentralization.

Acknowledgments

I would like to thank my supervisors, Michael Senn and François-Xavier Wicht, for their time, support, and especially for the regular meetings, which provided valuable discussions and feedback that helped me improve my work. I also wish to thank Professor Christian Cachin for the opportunity to conduct my thesis in his research group.

Contents

Chapter 1

Introduction

Over the past several years, blockchain technology has undergone significant growth and increasing popularity among the broader public and the scientific community. This development has led to the emergence of various consensus algorithms, each designed to fulfill specific security, efficiency, and decentralization requirements. While the Proof of Work (PoW) mechanism, originally established by Bitcoin, has faced increasing criticism, particularly due to its high energy consumption, more energy-efficient alternatives, such as Proof of Stake (PoS) and its further developments, are now gaining prominence.

One of these developments is the Proof of Staked Authority (PoSA) mechanism, which combines the stake-based selection process of PoS with the delegated trust model of Proof of Authority. It thus promises high efficiency while maintaining decentralization. However, combining these two concepts raises concerns regarding security, fairness, and resistance to manipulation. Binance, one of the world's largest cryptocurrency exchanges, developed PoSA specifically for their own blockchain platform, BNB Smart Chain (BSC).

Despite the widespread use of the BSC, particularly among cryptocurrency traders, the technical background of the underlying consensus mechanism has been largely undocumented to date. The official Binance documentation contains only superficial information, which is primarily aimed at a broad, non-technical target group [7]. A detailed, systematic PoSA protocol description is not publicly available.

This thesis aims to analyze the PoSA consensus mechanism of the BNB Smart Chain. To this end, the system components are described, along with their roles within the consensus process. The protocol flow is traced with the help of a structured pseudo code representation and deepened by analyzing the publicly available source code. In addition, central procedures such as the random selection of block producers are reproduced using scripts to evaluate better and understand how they work. The analysis in this thesis is based on the BNB Smart Chain source code as of February 28, 2025, after the merged commit 0445f9b [8].

The structure of this thesis is as follows: Chapter 2 introduces some theoretical foundations of distributed systems, consensus mechanisms, cryptographic primitives, and abstraction models. Chapter 3 presents the architecture of the BNB Smart Chain, including its history, node roles, and tokenomics. Chapter 4 contains the core of the work: a detailed, source code-based analysis of the PoSA protocol, covering block production, consensus logic, validator rotation, and shuffling. Chapter 5 summarizes the key findings and reflects on their performance, fairness, and implications for decentralization.

Chapter 2

Background

2.1 Pseudo code notation

Pseudo code will be used to present the BSC Consensus distributed algorithm. It abstracts away implementation-specific details while maintaining the core logic of the actual code. We adopt the notation proposed by Cachin, Guerraoui, and Rodrigues (abbreviated as CGR11 in this work) [16].

2.1.1 Components

We describe interfaces and algorithms that use an asynchronous event-based composition model. Each process has a set of software components, also called modules, which are identified by a name and characterized by properties. In addition, components also contain an interface, consisting of requested or indicated events.

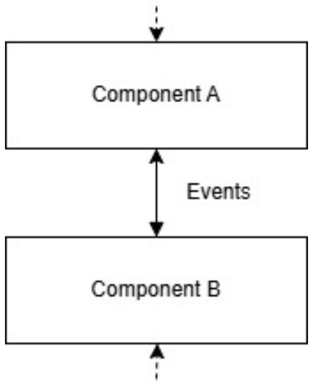

Figure 2.1. Composition Model of CGR11 [16]

2.1.2 Software stacks

Components can be combined to form software stacks. A component represents a specific layer in the stack in each process. Components in a stack communicate through events. Each component is structured like a state machine, which is triggered by receiving events. We represent these as

$$\langle\, co,\ \mathit{EventType} \mid \mathsf{Attribute}_1, \mathsf{Attribute}_2, \ldots \,\rangle$$

where *co* is the component and *EventType* is the event with its attributes. The attributes are information that are part of the event. A dedicated handler processes each event. Events from the same component are processed in the order in which they were triggered, according to the FIFO (first in, first out) principle. Events are processed sequentially, with each handler executing exclusively until completion. After execution, the process returns to a waiting state until the next event occurs. The corresponding behavior is illustrated in the pseudo code below:

Algorithm 1 Example of upon event and trigger

```
1: upon event ⟨ co1, Event | $att_1^1, att_1^2, \dots$ ⟩ do
2:     do something;
3:     trigger ⟨ co2, Event | $att_2^1, att_2^2, \dots$ ⟩ ;
```

2.1.3 Programming interface

The API of our components has two different event types:

- **Request events:** Are used to request a service from another component or to signal a condition. From the perspective of the component that handled the event, indication events are outputs.
- **Indication events:** Are used to transmit information or signal a condition. An indication event can also be a confirmation, for example, when the application layer component responsible for broadcasting indicates the message has been sent. From the component's perspective that triggered the event, indication events are outputs.

2.1.4 Modules

In our case, a module represents a cohesive unit of functionality describing the interaction between processes. Multiple such modules exist, each identified by a name and associated with events and properties that define its communication behavior. An example is shown below.

Module 1 Example interface and properties of example handler

```
4: Module:
5:     Name: ExampHandler, instance eh.

6: Events:
7:     Request: ⟨ eh, Examp1Event | $Attribute_1, \dots$ ⟩: $Description_1$
8:     Indication:⟨ eh, Examp2Event | $Attribute_2, \dots$ ⟩: $Descripton_2$

9: Properties:
10:    EH1: Property Name: Property Description.
```

2.2 System assumptions

2.2.1 Synchrony assumptions

An important part of distributed systems is dealing with the uncertainty of communication delays and processing speeds. A system's synchronization assumptions determine the time constraints on message transmission and process execution. There are three common models:

- **Asynchronous system:** No assumptions are made about the process running time and communication delays. There are no guaranteed upper limits for communication or processing latencies. Processes have no physical clocks, and time is defined exclusively by logical clocks, i.e., via message transmissions.

- **Synchronous system:** There is a known upper bound for process runtimes and communication delays. Each process has a local physical clock whose deviation from the global clock is limited by a constant δ. This makes implementing time-based coordination techniques, error detection with time limits, and worst-case performance analyses possible.
- **Partially synchronous system:** Such a system lies between the synchronous and the asynchronous model. Although there is an upper bound for process runtimes and communication delays, it is unknown when these bounds will be reached. It is assumed that there is an unknown point in time called Global Stabilization Time (*GST*) where the system becomes synchronous. Before that, the system is asynchronous.

2.2.2 Failure assumptions

Following the definitions presented in CGR11 [16], failure models specify how processes or communication links may behave incorrectly. We consider the following types:

Crash Failures A process halts and does not recover after it crashes.

Crash-Recovery Processes can crash but eventually recover and rejoin the system.

Omission Failures Messages are lost, but processes continue.

Byzantine Failures Processes may act arbitrarily or maliciously, deviating from the protocol in unpredictable ways.

2.2.3 Failure detection

Detecting process failures is important in distributed systems. However, it is not always necessary to recognize them immediately. In the asynchronous model, no assumptions are made regarding network or computational delays, making failure detection unreliable. Nevertheless, failures can be systematically addressed through abstractions known as *failure detectors*. In asynchronous systems, where neither processing nor communication delays are bounded, failure detectors cannot guarantee completeness or accuracy, and only provide partial information about faulty processes. In synchronous systems, where known bounds on processing and communication delays exist, failures can be detected completely and accurately. *Perfect failure* detectors reliably identify crashed processes using timeouts based on real-time clocks.

2.2.4 Classes of algorithms

Algorithms in distributed systems are classified according to the failure models in which they operate. This classification allows us to reason about the capabilities and limitations of different algorithms under various assumptions:

1. **fail-stop** algorithms: Processes can crash, but all others recognize this reliably (perfect detection).
2. **fail-silent** algorithms: Crashes are not detected
3. **fail-noisy** algorithms: Crashes can be detected, but not always accurately (eventually perfect detection).
4. **fail-recovery** algorithms: Processes can crash and recover later.
5. **fail-arbitrary** algorithms: Processes can behave arbitrarily, even maliciously.
6. **randomized** algorithms: Processes make probabilistic decisions.

These classes are not mutually exclusive, and many algorithms combine features from several of them.

2.3 Broadcast abstraction

In simple systems with only two processes, communication is typically based on point-to-point links using reliable protocols such as TCP. These protocols provide reliable message delivery. Distributed systems, however, usually have large numbers of processes and increased communication complexity. Although communication in distributed systems could be realized solely through point-to-point links, broadcast abstractions are introduced to simplify the design of algorithms by providing guarantees such as consistency and fault tolerance. Processes are allowed to send messages to a group of processes. However, achieving reliability in such group communication is difficult. Depending on the specific broadcast model, properties such as no message loss, no duplication, and consistent message ordering may be required. Different kinds of broadcast abstraction exist (see CGR11 [16]) e.g., Best-Effort, Regular Reliable Uniform Reliable, Stubborn, Logged Best-Effort, Logged Uniform Reliable, or Probabilistic Broadcast, which differ in their guarantees, underlying algorithms, and the complexity involved in ensuring reliability, ordering, and fault tolerance.
Probabilistic broadcast is a special broadcast abstraction. Compared to deterministic broadcasts, where it is guaranteed that every message will reach all correct processes with absolute certainty, a sent message is only guaranteed with a high probability of being received by all correct processes. This probability depends on parameters such as fanout (the number of recipients) and the number of forwarding rounds. A probabilistic broadcast is based on gossip protocols. In such protocols, a process sends its message to a small, randomly selected number of other processes. These, in turn, forward the message similarly, allowing it to propagate throughout the entire system gradually. Three properties apply here: Firstly, *probabilistic validity* means that a message sent by a correct process is received with a high probability by all correct processes. Secondly, there are *no duplicates*, which means that a message is delivered at most once by each process. Thirdly, there is *no creation*, according to which only messages sent by a process can be received by other processes. While the latter two properties are shared with other broadcast abstractions, probabilistic validity is unique to probabilistic broadcast models due to its reliance on probability rather than certainty.

Module 2 Interface and properties of probabilistic broadcast of CGR11

```
11: Module:
12:     Name: ProbabilisticBroadcast, instance $pb$.

13: Events:

14:     Request: $\langle pb, Broadcast \mid \mathsf{m} \rangle$: Broadcasts a message m to all processes.

15:     Indication: $\langle pb, Deliver \mid \mathsf{p}, \mathsf{m} \rangle$: Delivers a message m broadcast by process p.

16: Properties:

17:     PB1: Probabilistic validity: There is a positive value $\epsilon$ such that when a correct process broadcasts a
        message m, the probability that every correct process eventually delivers m is at least $1 - \epsilon$.

18:     PB2: No duplication: No message is delivere more than once.

19:     PB3: No creation: If a process delivers a message m with sender s, then m was previously broadcast
            by process $s$.
```

For example, we show an eager variant of the probabilistic broadcast of CGR11 in Algorithm 2. The algorithm uses fair-loss point-to-point links (abbreviated as *fll*), which model unreliable communication channels that may lose messages but do not duplicate or reorder them. Fair-loss links provide three key properties:

fair-loss If a correct process repeatedly sends a message to another correct process, the message is eventually delivered

finite duplication Messages are not delivered infinitely often unless they are sent infinitely often;

no creation Messages are only delivered if they were actually sent.

This abstraction is formally introduced in CGR11 and serves as the communication layer for gossip. A received message is forwarded immediately after receipt (*eager gossip*) to maximize the delivery probability. When the event $\langle pb, Broadcast \mid m\rangle$ is called, the process adds the message to its delivered set. After local delivery, the function *gossip* is called, which sends the message to a randomly selected subset of other processes. A handler receives the message with $\langle fll, Deliver \mid p, [Gossip, self, m, R]\rangle$ and checks whether it has already been received. If this is not the case, it is forwarded locally to other processes via gossip if there are still forwarding rounds ($R > 1$) left. *picktargets(k)* selects a random set of k processes. This choice is crucial for the probabilistic property of the algorithm.

Algorithm 2 Eager Probabilistic Broadcast by CGR11

```
20: Implements:
21:     ProbabilisticBroadcast, instance $pb$.

22: Uses:
23:     FairLossPointToPointLinks, instance $fll$.

24: upon event $\langle pb, Init \rangle$ do
        $delivered := \emptyset$

25: procedure gossip($msg$) do
26:     forall $t \in$ picktargets($k$) do trigger $\langle fll, Send \mid t, msg \rangle$

27: upon event $\langle pb, Broadcast \mid m \rangle$ do
28:     $delivered := delivered \cup \{m\}$
29:     trigger $\langle fll, Deliver \mid p, [Gossip, self, m, R] \rangle$
30:     gossip([Gossip, self,m,R])

31: upon event $\langle fll, Deliver \mid p, [Gossip, self, m, R] \rangle$ do
32:     if $m \notin delivered$ then
33:         $delivered := delivered \cup \{m\}$
34:         trigger $\langle pb, Deliver \mid s, m \rangle$
35:     if R $> 1$ then gossip([Gossip,s,n,R$-$1])
```

2.4 Consensus and atomic broadcast

A consensus abstraction describes an essential mechanism in distributed systems by which a set of processes reach a joint decision on a proposed value. All correct processes must initially propose a value, and all correct processes must decide on a value. The consensus protocol helps to ensure that all processes agree on the same value, even if some of them fail or misbehave, and typically fulfills the properties:

Termination Every correct process eventually decides some value.

Agreement No two correct processes decide differently.

Validity The decided value was suggested by at least one process.

An abstraction related to consensus is atomic broadcast (or total-order broadcast), where all correct processes must receive and process the same messages in order. This strong order and consistency property makes implementing consensus using atomic broadcast possible, as all processes decide on the same sequence of values. In contrast, atomic broadcast can also be implemented using consensus by deciding on the sequence of messages step by step. Therefore, atomic broadcast and consensus can be shown to be equivalent.

2.5 Blockchain

A *blockchain* is a decentralized ledger that stores transactions in ordered units called *blocks*. Each block contains a *header* with information such as timestamps, cryptographic hashes, references to previous blocks, and a *body* containing a list of validated transactions. Blocks are linked together chronologically and form a chain. Each block (in this relation called *child*) refers in the header to the previous block's hash (also called parent block). The *block height* of each block indicates its position in the chain. The very first block is known as the *genesis block* and has a height of 0. The block height helps to maintain the sequence and, in the case of forks, to determine the longest chain, also known as the *canonical blockchain*. Time is divided into repeating intervals in many blockchains, known as *epochs*. An epoch defines a fixed number of blocks or time units and coordinates activities such as validator changes, distribution of rewards, or protocol resets. Each blockchain also has a *consensus* protocol to agree on new blocks. Some are also EVM-compatible and support smart contracts, which are compiled for the *Ethereum Virtual Machine* (EVM). The EVM is a Turing-complete, stack-based virtual machine that executes smart contracts and transactions deterministically. Smart contracts written in EVM bytecode can be executed on such blockchains.

2.6 Cryptography

Cryptographic processes are a central component of secure blockchain protocols. They are essential in ensuring data integrity, authentication, and protection against manipulation or forgery. This section presents the most important procedures relevant to the PoSA consensus mechanism.

2.6.1 SHA-3

A cryptographic hash function $H : \mathcal{X} \rightarrow \mathcal{Y}$ is a function that maps input of arbitrary length to outputs of fixed length with the properties

- **Pre-image resistance:** It should be difficult to find any message $x \in \mathcal{X}$ from a hash value $y \in \mathcal{Y}$ such that $y = H(x)$.
- **Second pre-image resistance:** With a given value $x_1 \in \mathcal{X}$ it should be difficult to find a message $x_2 \in \mathcal{Y}$ such that $H(x_1) = H(x_2)$
- **Collision resistance:** It should be difficult to find any two different messages x_1 and x_2 such that $H(x_1) = H(x_2)$.

SHA-3 is a special variant of a cryptographic hash function that uses a sponge construction and the so-called Keccak permutation and provides additional security, such as resistance against length extension attacks. These properties ensure that reversing or extending hash outputs is computationally infeasible, making SHA-3 a robust alternative to its previous versions, like SHA-2.

2.6.2 BLS

The Bonneh-Lynn-Shacham (BLS) [14] digital signature scheme is a cryptographic signature scheme that uses a bilinear pairing $e : G_1 \times G_2 \rightarrow G_t$, where G_1, G_2 and G_T are elliptic curve groups of prime order q, with a hash function $H : \mathcal{X} \rightarrow G_1$. Based on the elliptic curves, it allows BLS to have shorter signatures than Full Domain Hash (RSA-based) [17] for a similar level of security. The BLS signature scheme consists of three functions:

1. Key generation: *KeyGen*$() \rightarrow (sk, pk)$, where sk as private key and $pk \in G_2$ as public key.
2. Signing: *Sign*$(m, sk) \rightarrow \sigma$, where m is the message and σ the signature
3. Verification: *Verify*$(\sigma, pk) \rightarrow \{$*true*,*false*$\}$

Properties

Authenticity: Ensuring that only the signer can produce valid signatures.

Unforgeability: Generating a valid signature without the secret key is computationally infeasible.

Unique and deterministic: Only one valid signature for a given key and message.

Signature Aggregation: Multiple signatures, public keys, and messages can be aggregated into a single signature.

Chapter 3

BNB Smart Chain

3.1 Historical background

The BNB Smart Chain (BSC) started as a hard fork of the *Go Ethereum* protocol and was created to increase the speed of block production with lower transaction fees and launched in 2020 [10]. This was achieved with a dual-chain construction. The BNB Beacon Chain (BC) provided high-performance trading and asset transfers with a Tendermint-based delegated Proof of Stake consensus and was primarily used for Binance trading and asset transfers. However, it had limited programmability, which meant no support for smart contract. On the other hand, the focus of the BSC is on programmability. It is a programmable blockchain with smart contracts and a newly developed consensus called Proof of Staked Authority. The BSC is EVM-compatible, and its primary focus is dApp, DeFi, and token ecosystems. The communication between these two blockchains initially happened in one direction, from BSC to BC, via an oracle relayer. This service forwarded cross-chain messages or smart contract events. In the reverse direction, communication from BC to BSC occurred over a BSC relayer that handled smart contract events. In November 2024, Binance finalized its most significant system change and shut down the BC to let the BSC run alone [4]. For this thesis, we focus exclusively on the new single-chain BSC.

3.2 Overview and architecture

3.2.1 BSC model

For our analysis, we assume a partially synchronous model for the BSC. This means that the system operates asynchronously for a period of time, but eventually enters a synchronous phase, for example, when a new block production starts. Production begins at a fixed time (UTC-based), meaning all participating nodes must work synchronously. In addition, the system allows processes to crash and rejoin the consensus and block-production processes and handles malicious process behavior. Thus, we have a byzantine failure model.

3.2.2 BSC network

The node discovery protocol provides a way to find nodes that can be connected. It uses a Kademlia protocol [18], a peer-to-peer distributed hash table (DHT), to maintain a distributed database of the IDs and endpoints of all listening nodes. In the BSC, a peer can connect to 200 other peers. A gossip protocol propagates new messages efficiently among peers to transfer information across the peer-to-peer network, which operates over TCP.

3.3 Node roles and sequencing

Fast node *Fast nodes* store the full blockchain history, verify the state of all accounts, and are optimized for fast data access and network queries. They can receive and validate new blocks and transactions and respond to data requests from the network.

Full node *Full nodes* can perform all functions of fast nodes, including storing the full world state. In contrast to fast nodes, they can achieve the status *validator node*, which is required to participate in the consensus process and thus produce and vote for new blocks. While both node types can synchronize from snapshots, only full nodes can become validators. Since the current block height exceeds 48 million, initializing a new full node from the latest snapshot significantly reduces the synchronization time to join the network.

Archive nodes *Archive nodes* are full nodes with additional settings. They store all historical data of the blockchain since the genesis block and, compared to a normal full node, store not only the status change data of recent blocks but also the complete state changes for every block in the chain [6].

Implementation structure To further understand the consensus, let us look at the architecture of a full node. It should be noted that Figure 3.1 is a simplified model that is limited to relevant parts. The origin of BSC as an Ethereum fork is also evident in the source code, where the main full node service is still implemented in a structure named Ethereum.

The architecture includes the following main components:

Consensus Engine Implements the PoSA protocol and is used by other components.

Vote Pool and Transaction Pool The vote pool stores local and external validator votes. The transaction pool stores all pending transactions.

Vote Manager Responsible for signing votes and storing vote-related data in a journal. It includes the *Signer* and *Journal* submodules.

Handler Manages communication between different full nodes. Notably, the handler includes a malicious vote monitor used to document the misbehavior of other nodes.

Miner Responsible for creating new blocks. This process orchestrates the entire block creation process, from initializing to finalization.

Worker Responsible for creating the block body, i.e., filling in the data.

Bid Simulator Simulates block variants that generate the highest possible rewards based on the fee. This is done according to the concept of Maximum Extractable Value (MEV), where the reward for the miner is maximized through the targeted arrangement of transactions and the addition and removal of transactions within a block. Particularly, fee-intensive transactions are prioritized.

Chain DB The Chain DB consists of the *BlockStore* for storing block data and the *StateDB* for maintaining the world state, including account balances and smart contract storage.

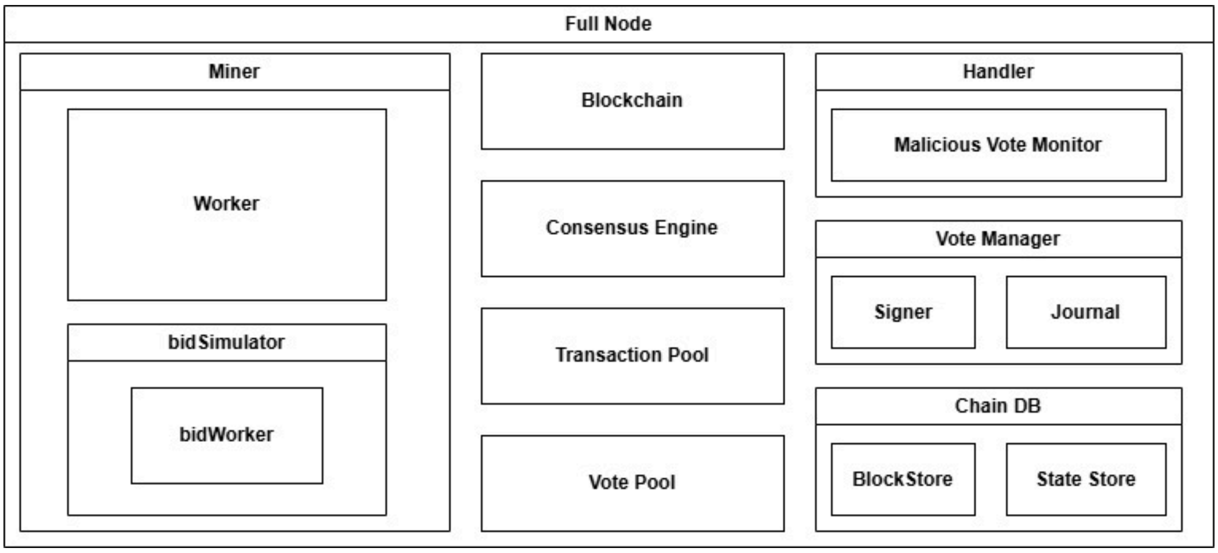

Figure 3.1. Logical architecture

3.4 Validator and delegators

Validators play a central role in the BSC. They are responsible for proposing and signing new blocks. Additionally, they participate in the consensus process to validate blocks proposed by others. In theory, every full node has a chance to become a validator if certain requirements are met:

- Fulfillment of the necessary hardware requirements
- Operation of a BSC full node
- Owning of at least 2,000 BNB (1,155,022 USD) as a stake

If these three conditions are met, the full node will be included in a list of validator candidates. However, active participation in protocols with block verification is restricted to the top 45 validator candidates based on the amount of BNB staked. From this set, 21 validators are selected for each epoch to produce and sign blocks. Further details are provided in Section 4.5.

Currently (as of 16.04.2025), the validator in 45th place has 75,375 BNB (43,529,901 USD), making it very difficult for individuals to participate in the process. To solve this problem, there is another role, namely delegators. These can assign their BNB to validators on the candidate list. This helps a validator achieve a higher ranking and increases their chances of being chosen to create and verify blocks. In return, the validator receives rewards and pays the delegators a certain percentage. Each validator can determine the amount of this percentage themselves. However, this delegation mechanism also has limitations. Although delegators can decide for themselves which validator they want to support, they have no control over the behavior of the validator. If a validator is penalized by having their voting power withdrawn, they can no longer generate rewards, and as a result, the delegators also no longer receive any rewards.

3.5 Tokenomics and staking

BNB is a token created by Binance in 2017. Initially, the cryptocurrency was launched as an ERC-20 token on Ethereum but was later transferred to the Binance blockchain. Initially, 200 million BNB coins were defined, and 50% were sold at an Initial Coin Offering crowdfunding to a price of 1 ETH for 2'700 BNB or 1 BTC for 20'000 BNB. Based on BNB's primary use for paying trading fees, the token is considered a utility coin with deflationary behavior [1]. Four times a year, there is a coin-burning event, where, based on produced blocks and the average price of BNB, Binance repurchases BNB coins and transfers them to a black hole wallet, making the tokens permanently inaccessible and thereby reducing the total supply. In addition, 10% of every transaction fee is also burned, reducing the total supply. Since the launch, 57.9 million BNB coins have been burned in that way, and the goal is to reach a remaining supply of 100 million coins [3].

3.6 System Contracts

The BSC works intensively with system contracts, integrating internal smart contracts into their protocols. On the one hand, system contracts carry out public votes on proposed changes to the BSC. On the other hand, they enable on-chain governance, including sorting and updating the list of validator candidates daily, selecting validators for the next block production epoch, and distributing rewards [9].

3.7 Slashing

Slashing is a punitive mechanism on the blockchain that penalizes validators who violate the protocol's rules, for example, by signing conflicting blocks (double signing), failing to produce blocks, or behaving maliciously or negligently. It ensures that validators have a financial incentive to behave honestly and remain online.
The enforcement of slashing is implemented partly by system contracts and partly through local validator logic [11]. While system contracts handle the distribution of penalties and rewards, local validator logic is responsible for collecting participation metrics, such as missed slots, and for detecting double signs or malicious votes.

A distinction is made between three slashing scenarios:

Double Sign A double sign occurs when a validator signs more than one block with the same block height and the same parent block. In this case, 200 BNB are withdrawn from the validator's self-delegated stake (self-delegated BNB). These are distributed among the remaining validators as a reward. In addition, the validator concerned loses their voting power for 30 days and is removed from the active validator set.

Malicious vote Here, a validator signs two fast-finality votes with the same target height or overlapping voting ranges. Since the protocol contains protection mechanisms against such behavior, such an incident strongly indicates manipulated or malicious code. If such a violation is detected, 200 BNB will be withdrawn from the self-delegated stake. The validator who reported the incident first receives 5 BNB as a reward; the rest is distributed among the other validators. In addition, the validator concerned is removed from the active set for 30 days and loses their voting power.

Unavailability If a validator misses at least 50 turns for block production within 24 hours, they will not receive any block rewards. These are distributed among the other active validators instead. If the number of missed blocks exceeds 150 within one day, an additional 10 BNB will be withdrawn from the validator's self-delegated stake and distributed among the other validators. In this case, the validator is removed from the active set for two days and loses voting power.

Chapter 4

Proof of Staked Authority

4.1 Overview

The Proof of Staked Authority (PoSA) is a hybrid consensus mechanism that integrates aspects of Delegated Proof of Stake (DPoS), where a small subset of network participants vote for validators, with aspects of Proof of Authority (PoA), where a fixed set of validators are responsible for producing and verifying blocks [2]. Specifically, a subset of full nodes, called validators, execute these tasks in a rotating schedule, similar to Ethereum's Clique consensus for private networks (e.g. in Geth) [19], in which a designated signer (or block producer) is selected per block in a round-robin manner among trusted nodes. The consensus process is divided into four sequential phases, as illustrated in Figure 4.1. Distinct events handled by dedicated components trigger each phase. Although only validator nodes actively produce blocks and vote on finality, other nodes, such as archives and fast nodes, still process and relay blocks and vote messages. This ensures that all nodes, regardless of their role, can recognize when a block has been finalized, a requirement for the BSC finality mechanism.

1. **Block Production (① – ⑤):** In the first phase, a currently selected validator according to the production schedule, which is set among the consensus validators, initiates the creation of a new block ①. This process includes gathering pending transactions from the transaction pool and generating the block header ② with relevant metadata (e.g., parent hash, timestamp, validator, and address). After producing and sealing the block header with relevant data (i.e., computing the block's cryptographic hash and signature so it cannot be changed anymore), the block is added to the local blockchain ③ and prepared for propagation ④. The sealed block is then announced by the gossip protocol ⑤.

2. **Block Reception and Voting (⟨1⟩ – ⟨6⟩):** In the second phase, validators who receive the new block ⟨1⟩ check its contents (e.g., signatures, header fields, transaction validity). Once the block is considered valid and successfully imported into the local blockchain ⟨2⟩, vote messages are generated for this block ⟨3⟩ ⟨4⟩. This vote confirms that the validator considers the block canonical. The vote is then further propagated through the network via a gossip protocol ⟨5⟩ ⟨6⟩. Meanwhile, the block is disseminated using complete block propagation to a subset of connected peers and hash announcements to all connected peers. Even if the full node is connected to a subset of peers in the network, the blocks are only sent to a subset of this subset

3. **Vote Reception and Propagation ($\boxed{1}$ – $\boxed{3}$):** The third phase is processing votes. When a validator gets a vote from a peer $\boxed{1}$, it first checks that the vote is valid and follows protocol rules (e.g., no voting twice, correct range, valid sign). Valid votes are stored in the local vote pool $\boxed{2}$ and sent out again to other peers $\boxed{3}$.

4. **Finality:** After the third phase, block production initializes again for the next block. Should the newly created block hold a valid attestation for its immediate predecessor, the parent block is

deemed justified. If a parent was already justified and now its child is also justified, then the parent block becomes finalized in the local blockchain.

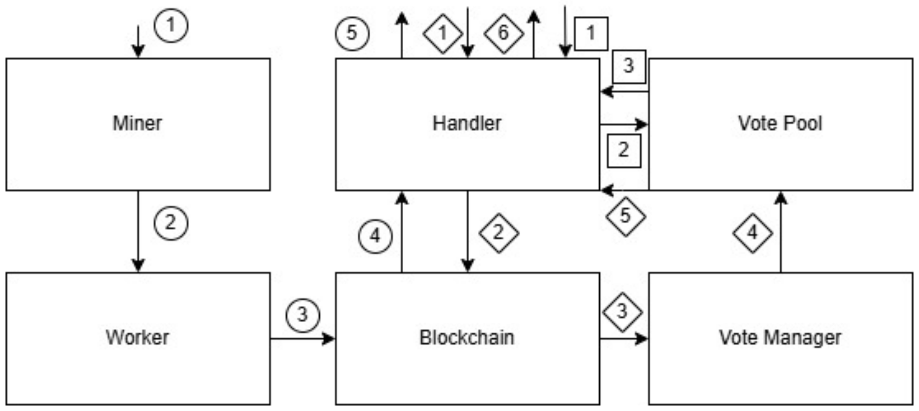

Figure 4.1. Componet process

4.2 Components for the consensus

To analyze the consensus, we should first know which components of the full nodes are relevant to the process.

Peer A peer represents a full node in the BSC network. Module 3 describes what happens when peers send and receive messages and then forward them to the internal routine. Only three events are relevant for us. With the *BlockAnnouncement* event, a peer receives a newly minted block and forwards it to internal processes. With the *BlockHashAnnouncement* event, the sending peer transmits the hash of a freshly mined block to the receiving peer. This signals that the sender already has this block in its blockchain. Both events are shown in Figure 4.1 at ⑤ and ⟨6⟩. The third and final event is *sendVote*, which receives a vote from the sender for a newly added block in its blockchain, as shown in Figure 4.1 at $\boxed{4}$.

Worker The worker is responsible for block production. The task is initialized with the *chainHead* event (Figure 4.1, ②). In addition, it can seal created blocks by triggering the *newTask* event, and when a block is finalized, it forwards the block to internal processes by triggering the *result* event.

Parlia The consensus engine of the BSC, which is called Parlia, is the central module responsible for consensus determination. It consists of several functions ensuring the block sequence matches all validating nodes. In the implementation, Parlia is structured as a collection of methods invoked during specific phases of block production. It verifies whether sufficient votes have been collected for a block and manages communication with the system contracts.

System contracts In the BSC, *system contracts* are essential to the consensus protocol. These special smart contracts are requested at specific points (called through Parlia methods). The system contracts are located at fixed addresses and are essential for governance, such as slashing, validator management, or selection of validators. Unlike regular smart contracts, they are not called by external transactions but specifically by the protocol. This work does not treat the system contracts as classical modules with an event interface called consensus. Instead, they will be modeled as functions where the result is returned immediately. This enables a more precise and deterministic model, which corresponds better to the actual process in the implementation. In practice, these calls behave as synchronous and blocking

Module 3 Interface and properties of a Peer

```
36: Module:
37:     Name: Peer, instance $peer$.

38: Events:
39:     Request: ⟨ peer, BlockAnnouncement | msg ⟩: Receives a message msg with a new block
        from another peer.

40:     Request: ⟨ peer, BlockHashAnnouncement | msg ⟩: Receives a announcement message msg
        with a new block hash from another peer.

41:     Request: ⟨ peer, SendVote | msg ⟩: Receives a vote message $msg$ from another peer.

42: Properties:
43:     P1: Probabilistic validity: There is a positive value $\varepsilon$ such that when a correct process broadcasts
        a message $m$, the probability that every correct process eventually delivers $m$ is at least $1 - \varepsilon$.

44:     P2: No duplication: No message is delivered more than once.

45:     P3: No creation: If a process delivers a message $m$ with sender $s$, then $m$ was previously broadcast
        by process $s$.
```

Module 4 Interface of Worker

```
46: Module:
47:     Name: Worker, instance $worker$.

48: Events:
49:     Request: ⟨ worker, chainHead ⟩: Receives an request to start with the production of a new block.
50:     Request: ⟨ worker, newTast | block ⟩: Receive a new block to seal.
51:     Request: ⟨ worker, result | block ⟩: Receives a new finalized block.
```

operations and are executed locally, so modeling them as immediate and deterministic function calls is a valid abstraction.

Handler The handler is responsible for communicating incoming and outgoing requests. The handler module is essential for the consensus protocol to process generated blocks and votes. It reacts to various events. If a new finalized block has been created locally, the handler receives the *newMinedBlock* event (Figure 4.1, ④), which is then communicated locally to initiate the next steps. In addition, if a block is to be announced to external peers (announceBlock), forwarded (propagateBlock), or a vote is to be disseminated (BroadcastVote) (Figure 4.1, ⑤ and ◇6◇). If a newly received block is to be imported, the *enqueue* event is processed. However, even if another peer has announced a new block, this request is intercepted after receipt with the *notify* event (Figure 4.1, ◇1◇).

Module 5 Interface of Handler

```
Module:
    Name: Handler, instance $handler$.

Events:
    Request: ⟨ handler, newMinedBlock | block ⟩: Receives a new mined block.
    Request: ⟨ handler, propagateBlock | peer, block ⟩: Receives a request to propagate
        a block to a certain peer.
    Request: ⟨ handler, announceBlock | peer, block ⟩: Receive sa request to announce a block to
        a certain peer.
    Request: ⟨ handler, enqueue | block ⟩: Receives a request to import a block to the blockchain.
    Request: ⟨ handler, notify | hash ⟩: Receives a notifyer for an announced block.
    Request: ⟨ handler, BroadcastVote | vote ⟩: Receives a request to broadcast a vote.
```

Vote Manager The Vote Manager is responsible for creating new votes. It starts when a *HighestVerifiedBlock* event (Figure 4.1, $\langle 3 \rangle$) occurs to generate a vote matching this block.

Module 6 Interface Vote Manager

```
Module:
    Name: VoteManager, instance $voteManager$

Events:
    Request: ⟨ voteManager, HighestVerifiedBlock | blockHeader ⟩: A new highest verified block event
    hvbe arrives.
```

Vote Pool The vote pool is a central part of the consensus. It stores votes cast by other validators for proposed blocks, indicating whether they consider a block valid. Both internal and external votes are recorded. When a new vote arrives from the vote manager or handler, a *putIntoVotePool* event (Figure 4.1, $\boxed{2}$ and $\langle 4 \rangle$) is triggered. This event first saves this vote but also forwards it to the handler so that it can be broadcast to peers who do not yet have it.

Module 7 Interface Vote Pool

```
Module:
    Name: VotePool, instance $vp$

Events:
    Request: ⟨ votePool, putIntoVotePool | vote ⟩: Request to put a vote into the vote pool.
```

Blockchain The blockchain structure is a central component of the BSC. It represents the canonical blockchain based on persistent data. The structure manages the chain's imports, processing, and reorganization and must maintain the current state of the blockchain. It also coordinates several subsystems, such as validation, transaction processing, and the state database. States are stored in a Merkle-Patricia Trie, a hybrid data structure combining a Merkle tree and a Patricia trie. This enables efficient and secure storage of key-value pairs such as account balances and contract storage. This mechanism is also used in Ethereum, and it allows state consistency to be verified via cryptographic proofs embedded in block headers.

4.3 Block production

4.3.1 Generate block

The entire routine is initiated by the worker (Module 4) receiving a *chainHead* event. As we shall see later, the event is triggered when a new block is inserted into the local blockchain. At the beginning of Algorithm 3, any ongoing block production tasks the worker might still execute are terminated. After the system has recorded the current time, many new blocks will be produced until either the time has expired (*prodTimeExpired*) or the gas in the pool is less than the transaction gas value. At the beginning of every block production, the system executes the function *prepareWork()*. As part of this process, the block header is initialized with information about the designated validators responsible for voting, and a delay is applied. If it is the peer's turn to produce, it starts with zero delay at a specific UTC time, and the vote result of the parent block will be checked. Then, the function *fillTransaction()* fills the block with transactions from the transaction pool (*txPool*). From all the blocks produced, *mostProfitableWork()* selects the block that gives the most reward based on the fee for the validator. If there is still time left until the block is validated, the selected block is compared with the best block based on the MEV of the bid simulator. Finally, the block is finalized with *FinalizeAndAssemble()*, filling the header with the validator information authorized to vote. The final block is passed to the sealer with the *newTask* event.

Algorithm 3 Generating Block Algorithm

```
69: Implements:
70:     Worker, instance $worker$

71: upon event ⟨ worker, chainHead ⟩ do                                      //worker.go:449
72:     worker.clearPendingTask()
73:     time ← currentTimestamp()
74:     blocks ← [ ]
75:     while gasPool.Gas() < TxGas ∨ prodTimeExpired(time) do               //worker.go:515
76:         block ← prepareWork()                                            //worker.go:994
77:         block ← fillTransaction()
78:         blocks.add(block)
79:     bestBlock ← blocks.mostProfitableWork()
80:     if sealTimeLeft() then
81:         bidBlock ← getBestBid(bestBlock.header.parentHash)
82:         if bidBlock.validatorReward > bestBlock.validatorReward then
83:             bestBlock ← bidBlock
84:     finalBlock ← bestBlock.FinalizeAndAssemble()                         //parlia.go:1398
85:     trigger ⟨ worker, newTask | finalBlock ⟩
```

4.3.2 Block sealing

After receiving the sealing task, Algorithm 4 starts with a check to determine whether the submitted block hash is already included in the blockchain. Then, all participants who are allowed to vote for this block are retrieved and saved in the seal header. Finally, a signature is generated from the header and the address of the block producer and appended to the header. Once the signature is complete, it triggers a *result* event.

Algorithm 4 Sealing Block Algorithm

```
86: Implements:
87:     Worker, instance $worker$

88: upon event ⟨ worker, newTask | finalBlock ⟩ do                 //worker.go:548
89:     sealHash ← SealHash(block.header)
90:     if sealHash ∈ blockchain then                             //check duplicate
91:         return
92:     rawSealHeader ← block.header                                //parlia.go:1602
93:     sealHeader ← assembleVoteAttestation(block, rawSealHeader)   //parlia.go:978
94:     sig ← sign(address, sealHeader)
95:     sealHeader ← sealHeader.append(sig)
96:     block.header ← sealHeader
97:     trigger ⟨ worker, result | block ⟩
```

4.3.3 Import block to blockchain

After generating the new block, it is integrated into the local blockchain with *writeBlockAndSetHead()* in Algorithm 5. The integration happens only for the local blockchain. Various system parameters, such as the highest verified block and header, are adjusted, which triggers a new *headChain* event that starts setting up the indication for a new block production. After successful integration, a new *minedBlock* event is triggered.

Algorithm 5 Import Block Algorithm

```
98: Implements:
99:     Worker, instance $worker$

100:Uses:
101:    Handler, instance $handler$

102:upon event ⟨ worker, result | block ⟩ do                    // worker.go:587
103:    blockchain.writeBlockAndSetHead(block)                  // peerset.go:457
104:    trigger ⟨ handler, newMinedBlock | block ⟩
```

4.4 Consensus

4.4.1 Propose block

After a new block has been created, the handler receives a message with the new block. Algorithm 6 starts a check if the block hash, if we have already received this block. This information is obtained locally from the cache and stored in a hash map. It is then checked whether the parameter for direct broadcast has been set. This is deactivated by default and can only be set manually. We assume that it has not been set. This means the block is not sent to all peers but only to the first $n = \sqrt{\text{amount of peers}}$. The set of peers is not shuffled, which would be unnecessary because we have a gossip network. Therefore, each peer is connected to a distinct set of peers, which may partially overlap with other peers, leading each peer to a different set. Before the block is distributed, it is noted locally who now knows the block, and then the block is transmitted. After transmitting to all desired peers, the remaining peers are informed that a new block exists. Before sending it to the individual peers, it is verified that they know the block and only the hash of the block is transmitted instead of the block.

Algorithm 6 Propose Block Algorithm

```
105: Implements:
106:     Handler, instance $handler$

107: Uses:
108:     Full Node, instance peer

109: upon event ⟨ handler, newMinedBlock | block ⟩ do                  // handler.go:918
110:     hash ← block.Hash()
111:     peers ← peersWithoutBlock(hash)
112:     if directBroadcast then
113:         transfer ← peers
114:     else
115:         transfer ← peers[:$\sqrt{\text{peers.length}}$]      // pick the first $\sqrt{|\text{peers}|}$ peers
116:     for each peer ∈ transfer do
117:         trigger ⟨ handler, propagateBlock | peer, block ⟩
118:     peer ← peersWithoutBlock(hash)
119:     for each peer ∈ peers do
120:         trigger ⟨ handler, announceBlock | peer, block ⟩

121: upon event ⟨ handler, propagateBlock | peer, block ⟩ do              // peer.go:297
122:     peer.knownBlocks.Add(block.hash)
123:     data ← rlpEncode(block)
124:     trigger ⟨ peer, BlockAnnouncement | data, data ⟩

125: upon event ⟨ handler, announceBlock | peer, block ⟩ do               // peer.go:271
126:     peer.knownBlocks.Add(block.hash)
127:     data ← rlpEncode(block.hash)
128:     trigger ⟨ peer, BlockHashAnnouncement | data.size, data ⟩
```

4.4.2 Receive new block

The transmission or the announcement is sent to the other peers in the network. With regard to the announcement, the message is then forwarded to the block fetcher. The header is checked to see whether the peer who made the announcement can be trusted and whether it is a useful block. It is then noted that a block announcement has been made and that the peer that sent the block owns it.
If a new block is transmitted, the body and hash are checked, and a sanity check is performed as Denial-of-Service protection. In addition, it is marked that the peer from which the block originates already owns this block. The block is then transferred to the block fetcher, which performs an additional validity and time difficulty check. After the header has been checked, the block is propagated to other peers. The insert function is executed to import the block locally in the blockchain, and if this call is successful, a block hash announcement is made to all peers so that they know that the block has been received and inserted.

Algorithm 7 Receive new block Algorithm

```
Implements:
    Peer, instance $peer$
    Handler, instance $handler$

upon event ⟨ peer, BlockAnnouncement | data, data ⟩ do                    // handler.go:179
    if data.sanityCheck() $\wedge$ data.validBodyAndHash() then  //protocols/eth/handlers.go:307
        return
    markBlock(data.peer, data.block.hash)
    trigger ⟨ handler, enqueue | block ⟩

upon event ⟨ peer, BlockHashAnnouncement | data.size, data ⟩ do //prot./bsc/handler.go:179
    blockHashPacket $\leftarrow$ data.hash                        //protocols/bsc/handlers.go:307
    hasBlock(data.peer, blockHashPacket)
    trigger ⟨ handler, notify | hash ⟩

upon event ⟨ handler, enqueue | block ⟩ do                          block_fetcher.go:760
    importBlock(block)

upon event ⟨ handler, notify | hash ⟩ do                            block_fetcher.go:255
    handler.announce.append(hash)

function importBlock(block)
    hash.check()
    block.verifyHeader()
    handler.propagateBroadcastBlock(block)
    blockchain.insertChain(block)
    handler.broadcastBlock(block)
```

4.4.3 Insert block

During block import, all block transactions are first checked with *checkTransactions()*. The consensus rules are then checked with *verifyHeader()*. All parameters and their conditions are listed in detail in Table 8. The *highest verified header* and *block* parameter are updated in the blockchain, and the import is started with *processBlock()*. With the *writeBlockAndSetHead()* function, the parameters highest verified block and header are updated, and after that, the *headChain* event will be triggered. Therefore, the worker starts a new task. The block is finally imported, triggering the *highestVerifiedBlock* event.

Header parameter	Condition
Extra size	≤ 32
Nonce	IS 8-Byte-0-Array
Uncle hash	IS EmptyUncleHash
Time	$>$ parent.time
Gas limit	$\leq 2^{63} - 1$
Gas used	$<$ Gas limit
Block Nr.	= Parent Nr. + 1

Table 4.1. Header conditions

Algorithm 8 Insert block Algorithm

```
151:function insertChain(block)                                   //blockchain.go:2036
152:    block.checkTransactions()
153:    block.verifyHeader()
154:    blockchain.updateHighestVerifiedHeader(block.Header())
155:    blockchain.processBlock(block)                              //start import

156:function processBlock(block)                                  //blockchain.go:2368
157:        writeBlockAndSetHead(block)                           // blockchain.go:2466
158:        trigger ⟨ voteManager, HighestVerifiedBlock | block.header ⟩
```

4.4.4 Generating vote

The vote manager now processes the *highestVerifiedBlock* event. It first checks whether the peer is allowed to send a vote at all. If it is not an active validator, the event is ignored. Then three consensus rules are checked, which are as follows

1. A validator must not publish two distinct votes for the same height.
2. A validator must not vote within the span of its other votes.
3. Validators always vote for their canonical chain's latest block.

If all rules are met, a vote is created and passed to the votepool with an event.

Algorithm 9 Generating Vote Algorithm

```
159:Implements:
160:    Vote Manager, instance $voteManager$

161:Uses:
162:    Vote Pool, instance $votePool$

163:upon event ⟨ voteManager, HighestVerifiedBlock | block.header ⟩ do        vote_manager.go:128
164:    nextBlockMinedTime ← block.header.Time
165:    if !voteManager.IsActiveValidator(voteManager.chain, block.header) then
166:        return
167:    if !voteManager.UnderRules(block.header) then
168:        voteMessage ← voteManager.SignVote(voteMessage)
169:        voteManager.WriteVote(voteMessage)                                  //into journal
170:        trigger ⟨ votePool, putIntoVotePool | voteMessage ⟩
```

4.4.5 Send vote

When the *votePool* receives a vote event, the header will be checked for a valid size. Then extracts the block from the chain and makes a basic verification. A basic verification checks whether this vote exists in the pool. Since votes can also be saved for future blocks, it is checked whether it is a future vote. The system limits the number of votes, so it can store a maximum of 21 per block hash and 50 future votes per block hash to prevent DOS attacks (Vote-Spam) and process them later. After a final verification, which checks if the vote comes from a valid validator, the vote is passed to the handler for transmission. The handler then checks who does not yet have this vote. Then, the vote is sent to each peer with a BSC connection, which is required as a full node. The time difficulty is the same as that of the sender. Therefore, it has the same current block. The vote is then saved in the vote pool and added to the node's local vote database.

Algorithm 10 Send Vote Algorithm

```
171:Implements:
172:    Vote Pool, instance $votePool$
173:    Handler, instance $handler$

174:Implements:
175:    Peer, instance $peer$

176:upon event ⟨ votePool, putIntoVotePool | vote ⟩ do                     //vote_pool.go:117
177:    checkVote(vote)
178:    voteBlock ← pool.GetVerifiedBlockByHash(vote.Data.Hash)
179:    if !pool.basicVerify(vote, vote.hash, blockchain.currentBlock()) then
180:        return
181:    if !pool.isFutureVote(vote) then
182:        if !pool.VerifyVote(vote) then
183:            return
184:        trigger ⟨ handler, broadcastVote | vote ⟩
185:    pool.putVote(vote)

186:upon event ⟨ handler, broadcastVote | vote ⟩ do                         // handler.go:883
187:    peers ← peersWithoutVote(vote.Hash())
188:    for each peer ∈ peers do
189:        ⟨ peer, sendVote | vote ⟩
```

4.4.6 Receive vote

If a peer receives a vote, it checks whether any content was sent. The vote is extracted from the message and transferred to the vote pool with a *putIntoVotePool* event. It is then processed there using the same logic as in Algorithm 10 and stored in its local vote pool at the end.

Algorithm 11 Receive Vote Algorithm

```
190:Implements:
191:    Peer, instance $peer$
192:    Vote Pool, instance $votePool$

193:upon event ⟨ peer, sendVote | vote ⟩ do                handler_bsc.go:49
194:    if votes.length > 0 then
195:        vote ← votes[0]
196:        trigger ⟨ votePool, putIntoVotePool | vote ⟩
```

4.4.7 Malicious behavior detection

Incorrect behavior by peers is tolerated to a certain degree in the BSC. However, based on snapshots of the system, malicious behavior can be tracked and checked how often this happens and reported to the slashing contract, which takes action depending on the frequency. The validator's BNB can be slashed, up to and including the loss of voting power, which means that the peer can no longer be selected as an active validator. This mechanism effectively acts as a failure detector, identifying faulty or malicious peers and ensuring they are removed from the consensus set if they miss their slots. In such cases, other validators take their place to maintain the integrity of the block production process.

System reaction on delayed validators As we see in Algorithm 3, when a block is created, it is first prepared with the *prepareWork()* function. The header time is set based on blockTime (:= *blockTime* + *p.backOffTime(snap, header, p.val)*). This delay is calculated with the function *backOffTime()* (Algorithm 12). The first step is to check whether it is the validator *val* turn to produce. If this is the case, the delay is 0. The system then checks whether a block was created a short time ago. If this is the case and it does not exceed a specific number, it can start again without delay. Then, it is checked whether the validator currently designated to produce the next block has already signed a block. All other validators that have recently signed a block are sorted out. A pseudo-random value *rand* is defined and generated deterministically based on the highest block number. While this value appears random, it is reproducible, as the same block height always yields the same result. This value shuffles the selected set of validators with a deterministic Fischer-Yates shuffle. The defined random functions accept a value whereby the generated pseudo-random number with the same input always has the same output.

Algorithm 12 Set random Production Queue Algorithm

```
function backOffTime(snap, header, val)                         //parlia.go:2124
    delay ← 1
    counts ← snap.countRecents()
    allValidators ← snap.validators()
    if snap.inTurn(val) then
        return 0
    if signRecentlyByCounts(val, counts) then
        return 0
    if signRecentlyByCounts(snap.inturnValidator(), counts) then
        delay ← 0
    validators ← [ ]
    for each validator ∈ allValidators do
        if !snap.signRecentlyByCounts(validator, counts) then
            validators.add(validator)
    randNumber ← rand.New(snap.highestBlock.Nr)
    r ← rand.New(snap.highestBlock.Nr)
    shuffleValidators ← shuffle(validators, rand)  // variant of the Fisher-Yates shuffle
    delay ← delay + shuffleValidators.index(val)
    return delay
```

4.5 Validator selection

4.5.1 Updating validators

Every day at 00:00 UCT the set of active validators is updated. This is initialized by the first newly produced block when it is finalized in the *FinalizeAndAssemble()* function from Algorithm 3. This checks whether the time of the parent block is the same day in UCT time as the block is currently being finalized (parlia.go:1469). If this is not the case, Algorithm 13 is executed. With *getValidatorElectionInfo()*, which is the parameter of the header of the parent block, all validators with the current voting power status are now retrieved. Firstly, the last updates are made using the parent block number, such as transferring rewards to the previous set of active validators and withdrawing the voting power from Byzantine validators. *getMaxElectedValidators()* specify the number of active validators selected for the next day. This value is currently set to 45 and is managed by a system contract. The list of validators is then sorted according to their voting power, which is the number of staked BNB, and the 45 validators with the most staked BNB are returned. This list is passed to a system contract, which saves the new active validator set.

4.5.2 Choosing mining validators

The *prepareWork()* function, which is executed in Algorithm 3, initializes the consensus field in the header for the vote signatures. Various parameters are set in the header, such as nonce and a timestamp. But the extra header is also set. Part of this is when a new set of mining validators is selected. This is done with prepareValidators. Algorithm 14 shows that the first check checks whether it is the first block of an epoch. Every 200th block marks a new epoch, and only then are newly selected mining validators written in the Extra header. The selection occurs in the *getCurrentValidators()* function, which calls a system contract.
The system contract first defines the number of cabinets and candidates to be selected. In the PoSA context, a cabinet always consists of the top 21 validators with the highest voting power. Candidates are defined as all active validators who are eligible to be selected as consensus validators but are not selected after the first shuffle in the selection process. Specifically, the 45 active validators are first retrieved and sorted by their voting power, and the epoch is initialized. The first *shuffle()* call then selects 18 validators from the top 21 (cabinets). The second shuffle call selects 3 validators from the remaining 27

Algorithm 13 Update Validator Algorithm

```
function FinalizeAndAssemble()                                    //parlia.go:1398
    systemcontracts.TryUpdateBuildInSystemContract()
    distributeIncomingRewards(header.coinbase)
    if isBreathBlock(parent,header)
        updateValidatorSetV2()
    buildBlock()
    block.setRoot(stateRoot)
    return block

function updateValidatorSetV2(chain, Header, parents)           // feynmanfork.go:99
    blockNr ← rpc.BlockNumberOrHashWithHash(header.ParentHash)
    validatorItems ← getValidatorElectionInfo(blockNr)
    maxElectedValidators ← p.getMaxElectedValidators(blockNr)
    topValidators ← getTopValidatorsByVotingPower(validatorItems, maxElectedValidators)
    method ← "updateValidatorSetV2"
    data ← p.validatorSetABI.Pack(method, topValidators)
    return p.applyTransaction(data)
```

candidates. These 21 chosen validators together form the consensus validator set for the next epoch and are responsible for producing and signing blocks. All values used in this procedure, such as the number of active validators, cabinets, and candidates, are defined and stored in system contracts.

4.5.3 Shuffle algorithm

In Algorithm 15 is the shuffle function implemented in the system contract. Three validators are always selected per shuffle call and swapped with three randomly chosen validators. The parameter *startIdx* determines which validator is first selected for the swap. The following two validators are always in descending order by ranking (or ascending order in the array). The pseudo-random variable used for the random selection of validators creates its value based on the epoch and the ranking of the validator to be swapped. The hash value is then calculated from these values and converted into an integer.
How it is implemented in the BSC, during the first execution of the shuffling algorithm, 18 out of the top 21 validators are "shuffled away," which means they get selected as consensus validators. The remaining three validators are carried over and shuffled again with the rest of the 45 active validators during the second term, in which three more validators are chosen.
Since the same parameters and ordered set of validators always return the same shuffled result, the output of the shuffle algorithm is predictable.

Algorithm 14 Get Mining Validators Algorithm

```
Implements:
    BSCValidatorSet, instance $bvs$

function prepareValidators(header)                                    // parlia.go:1807
    if header.number % Epoch $\neq$ 0 then
        return
    parentBlockNumber $\leftarrow$ header.number - 1
    Validators $\leftarrow$ getCurrentValidators(parentBlockNumber)
    newValidators $\leftarrow$ sortByAddress(Validators)
    for each validator $\in$ newValidators do
        header.appendToExtra(header, validator)
    return header

function getCurrentValidators(blockNr)                         // BSCValidatorSet.sol:385
    CandNum $\leftarrow$ 3                                      // number of candidates to select
    CabSize $\leftarrow$ 21                                 // number of validators to be selected
    validators $\leftarrow$ getValidators()
    epochNumber = blockNr / Epoch                                        // EPOCH := 200
    validators $\leftarrow$ shuffle (validators, epochNumber, CabSize - CandNum, 0, CandNum, CabSize)
    validators $\leftarrow$ shuffle (validators, epochNumber, CabSize - CandNum, CandNum,
                        CabSize - CandNum, validators.length - CabSize + CandNum)
    miningValidators $\leftarrow$ validators[:20]
    return miningValidators
```

Algorithm 15 Shuffle Algorithm

```

function shuffle(validators, epochNum, startIdx, offset, limit, modNum) //BSCValidatorSet.sol:757
    for i=0 to limit-1 do
        random $\leftarrow$ int(hash(epochNumber, startIdx + i)) mod modNum
        n $\leftarrow$ startIdx + i
        m $\leftarrow$ offset + random
        swap( validators[n], validators[m])
    return validators
```

BSC shuffle algorithm

Using an example, we show how the 21 consensus validators are selected. The first time the algorithm is run, 18 cabinets are selected. At the beginning, you have a list of all active validators sorted by the number of their voting power in descending order. The for loop is executed 3 times, as *limit* was defined with the number of candidates to be elected (3). modNum would be defined with the number of cabinets (21), which means that our random number is between 0 and 20. *startIdx* is chosen from the difference between the number of cabinets and the candidates to be selected, which is 18. Therefore, we will be $n \in [18, 20]$. *offset* was initialized with 0 and therefore $m \in [0, 20]$.

Example cabinet selection from Figure 4.2

1. Iteration: For $n = 18$ and for m a pseudo-random number 2 was generated. This means that the validators are swapped at $n + 1$ and $m + 1$.

2. iteration: For $n = 19$ and for m a pseudo-random number 17 was generated. This means that the validators are swapped with each other at $n + 1$ and $m + 1$.

3. iteration: For $n = 20$ and for m a pseudo-random number 1 was generated. This means that the validators are swapped with each other at $n + 1$ and $m + 1$.

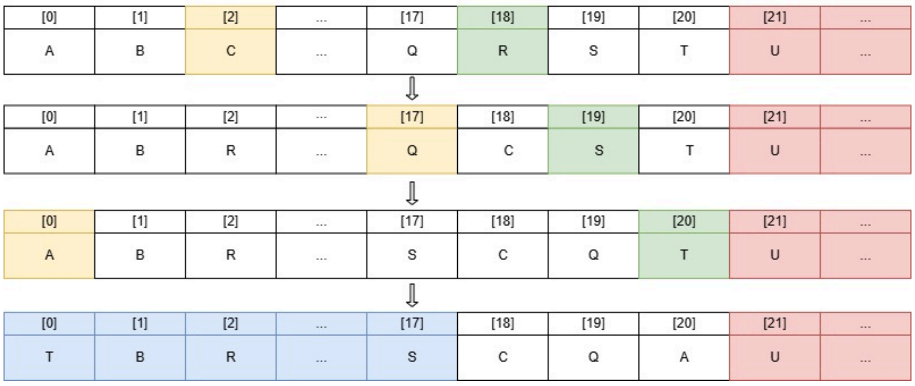

Figure 4.2. Example cabinet selection

Now the top 21 validators of the list have been shuffled with 3 iterations. Now the shuffled list is shuffled again, but with slightly different parameters. Newly defined is $offset$ with the same value as $startIdx$, namely 18. This means that m can no longer be less than 18 and therefore all elements set at positions 0 to 17 can no longer be selected. In addition, $modNum$ was set by the number of validators not chosen yet (27).

Example candidate selection from Figure 4.3

1. Iteration: For $n = 18$ and a pseudo-randomly generated $m = 42$. This means that the validators are swapped at $n + 1$ and $m + 1$.

2. iteration: For $n = 19$ and a pseudo-randomly generatedd $m = 22$. This means that the validators are swapped with each other at $n + 1$ and $m + 1$.

3. iteration: For $n = 20$ and a pseudo-randomly generated $m = 20$. This means that the validators are swapped with each other at $n + 1$ and $m + 1$. Since $m = n$, nothing is done.

The list of active validators has been shuffled so that the new top 21 are defined as consensus validators.

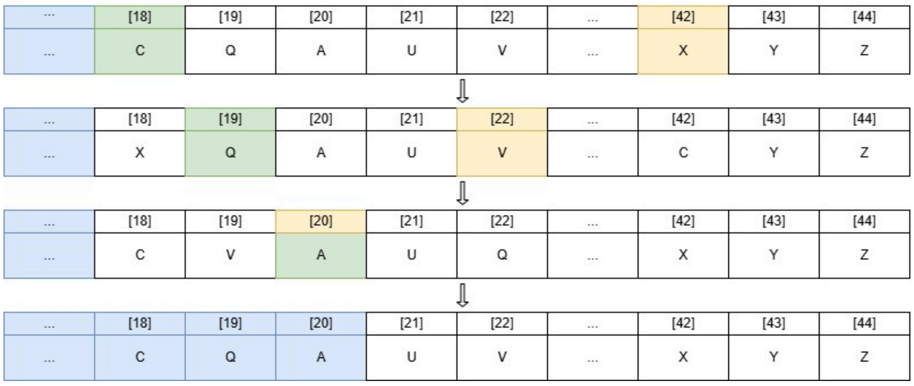

Figure 4.3. Example candidate selection

Analysis

Due to the deterministic behavior of the algorithm, it can be determined how and at which epoch the shuffling takes place. A simulation of the shuffling results has shown that the probability of being selected

as a validator at positions 1 to 19 in the next epoch is evenly distributed. The probability decreases significantly from position 20 and again significantly at position 21. However, this is to be expected due to the given shuffle algorithm. The simulation confirms that behavior and also, in theory, the probability that validators 20 and 21 are in positions 19-21 at the end of the first shuffle is significantly higher, as when it is their turn to swap, they have more opportunities to be swapped with a validator that was previously placed in position 18 and/or 19. As a result, they are less likely to make the top 18 on the first shuffle. As can be seen in the evaluation of `bscscan.com`, which is summarized in Figure 4.4, this also influences the behavior of the delegators. Validators ranked 19th or higher are significantly more attractive and potentially more profitable than those ranked lower.
Binance's official documentation states that the top 21 validators have a higher chance of being selected for an epoch than the remaining 24 active validators [13]. However, it does not clarify that validators ranked 20th and 21st have a lower probability of selection than those ranked in the top 19.

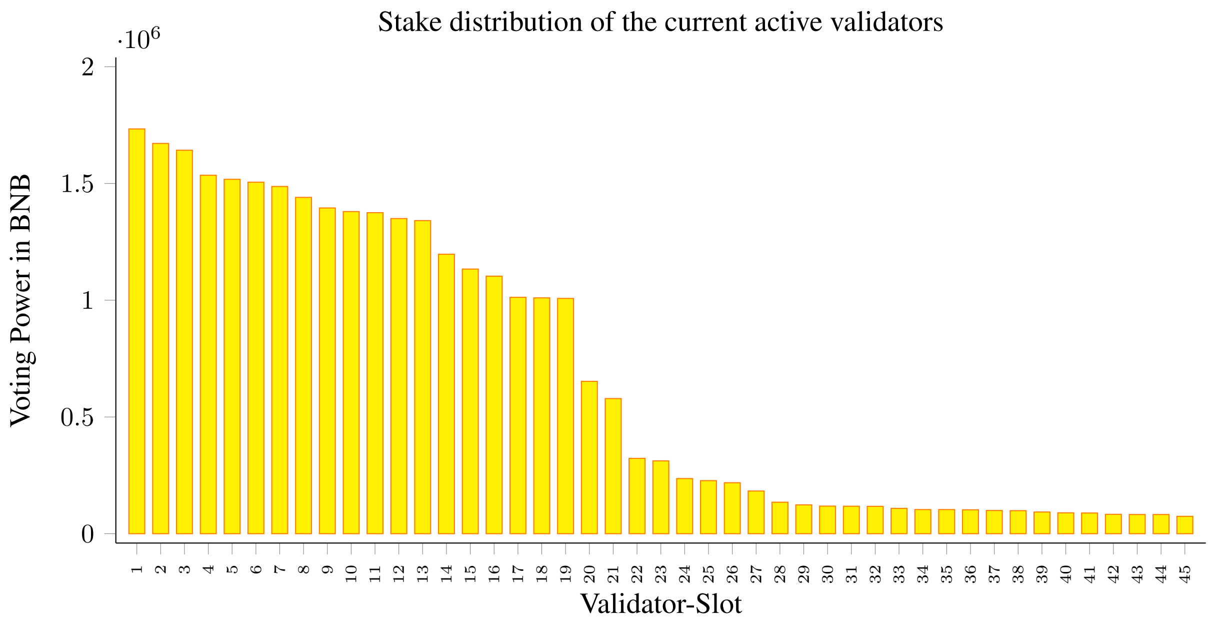

Figure 4.4. Current stake distribution across the 45 active validator slots, including both self-delegated stake and delegated tokens from delegators [15].

Since each shuffling permutation can be calculated for each epoch, it is possible to predict precisely when which validator can produce blocks and thus generate rewards. A simulation can be used to determine the probability of a validator being assigned to a particular rank. As shown in Figure 4.5, the higher probabilities correlate with the validators' accumulated stakes. In addition, the ranking of the validator sets is very static, and there are hardly any significant changes. Mechanisms have been introduced to prevent this predictability from exploitation, making so-called redelegation hopping more difficult. Among other things, a redelegation fee of 0.002% of the delegated amount was introduced.

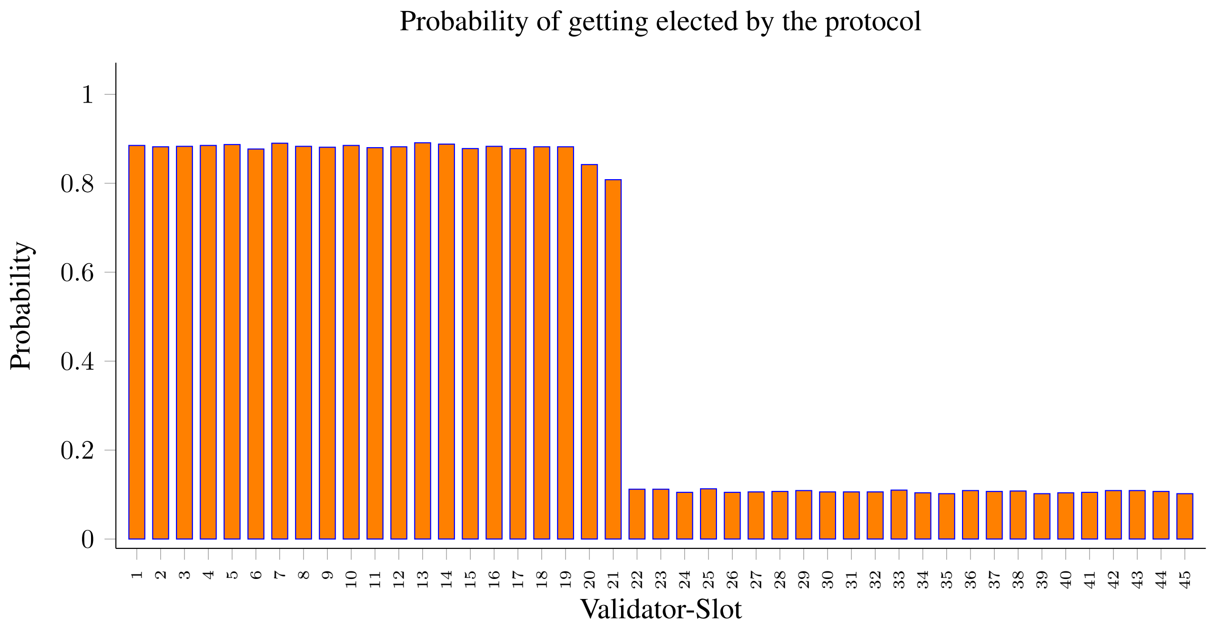

Figure 4.5. Data based on a simulation of over 45 million epochs using the BSC validator selection logic. Each bar shows the empirical probability that a validator is selected in a given epoch, depending on their rank (slot) in the ordered validator list.

4.5.4 Fast finality

Once a block has been successfully imported into the blockchain, it is not initially finalized. The BSC employs a *fast finality* mechanism introduced in BEP-126 [5], which deterministically finalizes blocks based on aggregated validator votes. To reach *fast finality*, the mechanism includes voting rules:

Rule 1: A validator must not publish two distinct votes for the same block height.

Rule 2: A validator must not issue a vote that falls within the span of its previously issued votes.

Rule 3: Validators should always vote for the head of the canonical chain known to them.

Violations of these rules will lead to slashing, as they threaten the accountable safety of the protocol.

Finalization The finalization begins when the next block (i.e., the direct child block) is created. During block production, a new block is prepared using the *prepareWork()* function (see Algorithm 3 in Section 4.3.1), where the header of the new block is generated in this step. Provided there are all valid votes for the parent block. When this new block is imported, the header is checked by the *verifyHeader()* function (see Algorithm 8 in Section 4.4.3). Part of this check is validating the contained vote result for the parent block, which is carried out by the *verifyVoteAttestation()* function (see Algorithm 16). At the beginning of this function, *getVoteAttestationFromHeader()* is used to extract all votes for the parent

block. Following this, the parent block is referenced from the current blockchain and an associated snapshot is loaded, from which the validators entitled to vote are determined. Afterward, check whether the parent block has received a majority of at least two-thirds of the votes of all validators eligible to vote. This two-thirds threshold is a known bound for Byzantine Fault Tolerance. It ensures consensus as long as the number of faulty nodes f satisfies the condition $3f + 1 \leq N$, where N is the total number of validators.
If this condition is met, an attestation is generated for the parent block, based on which the block is classified as *justified*. This justification is the first step for the *fast finality* mechanism. In the second step, a block is considered *finalized* if it and its direct child block are classified as justified. Finalized blocks are permanently part of the canonical chain. They are protected by the protocol, which explicitly prevents any chain reorganization that would create an alternative fork, excluding these blocks.

Algorithm 16 Check Vote Attestation Algorithm

```
function verifyVoteAttestation(chain, Header, parents)                    //parila.go:459
    attestation ← getVoteAttestationFromHeader(header)
    parent ← getParent(chain,header, parents)
    snap ← snapshot(chain, parent.Number-1, parent.Hash, parents)
    validators ← snap.validators()
    validatorsBitSet ← bitset.From(attestation.VoteAddressSet)
    votedAddrs[]
    for i = 0 to validators.length do
        if !validatorsBitSet.Test(i) then
            continue
        voteAddr ← bls.PublicKeyFromBytes(snap.validators[i].voteAddress)
        votedAddrs ← append(voteAddr)
    if votedAddrs.length < snap.validators.length · $\frac{2}{3}$ then
        return error("Invalid attestation, not enough validators voted")
    aggSig ← bls.SignatureFromeBytes(attestation.AggSignature)
    aggSig.FastAggregateVerify(votedAddrs, attestation.Data.hash)
```

Chapter 5

Conclusion

This thesis is dedicated to the structured and source code-based analysis of the Proof of Staked Authority (PoSA) consensus protocol of the BNB Smart Chain (BSC). By analyzing the components, process flows, and methods, a contribution was made to the formal description of a protocol that has hardly been documented to date. The implemented pseudo code shows independent, comprehensible and detailed documentation for the first time to understand the complex interactions within the BSC consensus.
A central insight of this work follows from the analysis of the validator shuffling mechanism, which determines at the beginning of each epoch the validators that should create the next blocks. This implementation is based on deterministic pseudo-random functions, which is efficient, but the result of the shuffling can be predicted. Our simulation has shown that the probability for validators in certain positions is significantly lower than those in the top 19. As a result, this strengthens a centralized structure, as certain validators almost all receive delegated stakes, which can also be observed on BSCScan.
In addition to shuffling, the entire PoSA consensus flow, which goes from block production to finalizing the block, was also presented and traced. This shows that high transaction speed can be achieved in the BSC through the targeted use of the Gossip protocol, system contracts and deterministic algorithms.
Although BSC is very efficient in block production and finalization, there are still concerns about the protocol's fairness and decentralization. The strong dependency on centrally managed parameters (number of active validators or voting power) creates an environment prone to concentration of power and potentially limited diversity in the validator set.
The analysis of the validator shuffling mechanism has shown a distinct imbalance in the selection of validators in the long run. Based on these results, additional research could examine how the selection process can be adjusted or complemented to ensure a more balanced distribution of the validation task and thus improve fairness and decentralization in the protocol. Future work could investigate whether the deterministic nature of validator selection introduces new attack vectors, such as targeted denial-of-service attacks against predictable block producers. Furthermore, the concentration of delegated stakes raises concerns about censorship, central control, and the network's resilience against collusion or malicious behavior among dominant validators.

Bibliography

[1] Binance Academy, "What Is BNB?." https://academy.binance.com/en/articles/what-is-bnb, 2018. Accessed: 2025-03-04, Updated: 2024-02-16.

[2] Binance Academy, "Proof of staked authority (posa)." https://academy.binance.com/en/glossary/proof-of-staked-authority-posa/, 2024. Accessed: 2025-06-09.

[3] BNB Chain, "30th BNB Burn." https://www.bnbchain.org/en/blog/30th-bnb-burn-2, 2024. Accessed: 2025-05-05.

[4] BNB Chain, "Final Sunset Plan of BNB Beacon Chain." https://www.bnbchain.org/en/blog/final-sunset-plan-of-bnb-beacon-chain, 2024. Accessed: 2025-05-05, Published: 2024-10-16.

[5] BNB Chain Community, "BEP-126: Fast Finality Mechanism." https://github.com/bnb-chain/BEPs/blob/master/BEPs/BEP126.md, 2021. Accessed: 2025-05-06.

[6] BNB Chain Developers, "Full node | bnb smart chain documentation." https://docs.bnbchain.org/bnb-smart-chain/developers/node_operators/full_node/, 2024. Accessed: 30 April 2025.

[7] BNB Chain Developers, "BNB Chain Documentation." https://docs.bnbchain.org/, 2025. Accessed: 2025-06-05.

[8] BNB Chain Developers, "BNB Smart Chain Source Code Repository." https://github.com/bnb-chain/bsc, 2025. Accessed: 2025-05-15.

[9] BNB Chain Developers, "System Contract Repository." https://github.com/bnb-chain/bsc-genesis-contract, 2025. Accessed: 2025-05-15.

[10] BNB Chain Documentation, "BNB Smart Chain - High Performance DeFi Hub." https://docs.bnbchain.org/bnb-smart-chain/overview/, 2020. Accessed: 2025-05-05, Launch Date: 2020-09-01.

[11] BNB Chain Documentation, "BSC Slashing Overview." https://docs.bnbchain.org/bnb-smart-chain/slashing/overview/, 2024. Accessed: 2025-04-30.

[12] BNB Chain Documentation, "BSC Staking Overview." https://docs.bnbchain.org/bnb-smart-chain/staking/overview/, 2024. Accessed: 2025-05-05.

[13] BNB Chain Documentation, "BSC Validator Overview." https://docs.bnbchain.org/bnb-smart-chain/validator/overview/#the-network-topology, 2024. Accessed: 2025-05-05.

[14] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," *Journal of Cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[15] BscScan, "Validator Set Info – BNB Smart Chain Explorer," 2025. Accessed: 2025-05-06.

[16] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[17] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. CRC Press, 2nd ed., 2014.

[18] P. Maymounkov and D. Mazieres, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” in *Peer-to-Peer Systems. IPTPS 2002* (P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, eds.), vol. 2429 of *Lecture Notes in Computer Science*, pp. 53–65, Springer, 2002.

[19] P. Szilágyi, “EIP-225: Clique proof-of-authority consensus protocol.” `https://eips.ethereum.org/EIPS/eip-225`, 2017. Ethereum Improvement Proposal.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Thun, 18.06.2025	[illegible]
Ort/Datum	Unterschrift