Mathematical Modelling of Bitcoin and Ethereum

by

Masuda Rahman Fatima
20101015

A thesis submitted to the Department of Mathematics and Natural Science
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering
with a Second Major in Mathematics

Department of Mathematics and Natural Science
BRAC University
April 2025

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at BRAC University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

_____

Masuda Rahman Fatima
20101015

Approval

The thesis/project titled "Mathematical Modelling of Bitcoin and Ethereum" submitted by

1. Masuda Rahman Fatima (20101015)

Of Spring, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science and Engineering with a Second Major in Mathematics on April 20, 2025.

Examining Committee:

Supervisor:
(Member)

Dr. Md. Sadek Ferdous
Professor
Department of Computer Science and Engineering
BRAC University

Co-supervisor:
(Member)

Dr. Syed Hasibul Hassan Chowdhury
Professor
Department of Mathematics and Natural Sciences
BRAC University

Program Coordinator:
(Member)

Dr. Syed Hasibul Hassan Chowdhury
Professor
Department of Mathematics and Natural Sciences
BRAC University

Head of Department:
(Chair)

Dr. Md. Firoze H. Haque
Associate Professor and Chairperson
Department of Mathematics and Natural Sciences
BRAC University

Abstract

Blockchain technology provides a decentralized ledger that enables secure, transparent, and immutable information sharing. It serves as a platform for exchanging cryptographic coins with economic value or data over the internet without requiring a central authority, achieving consensus through a peer-to-peer network. This thesis presents mathematical models for Bitcoin and Ethereum (version 1.0 and 2.0), two of the most prominent blockchain platforms, to analyze and represent their core mechanisms.

The models define transactions, blocks storing these transactions, and the consensus algorithms that synchronize blocks across the blockchain network, through the use of sets, tables, diagrams and algorithms. Additionally, a Finite State Machine (FSM) representation is provided to illustrate Ethereum's transaction based state machine behavior. Use of cryptography in both Bitcoin and Ethereum, specifically Elliptic Curve Cryptography (ECC) for transaction validation has also been explored.

By providing a mathematically structured representation of these systems, this thesis could help researchers and engineers gain a deeper understanding of blockchain mechanisms, analyze their security and efficiency, and explore potential optimizations. The simplified approach also makes the topic more accessible to those without a computer science background, encouraging interdisciplinary research and collaboration in blockchain development.

Keywords: Bitcoin, Ethereum, Ethereum 2.0, Proof of Work, Proof of Stake, Elliptic Curve Cryptography, Mathematical model, Finite State Machine

Dedication

To Momo and Rusty, my two cat companions, who sat beside my laptop throughout the process. This thesis is written in the first-person plural because, by "we", I mean the three of us—me, Momo, and Rusty.

Acknowledgement

I would like to give my acknowledgment and the longest thanks to my supervisor Dr. Md Sadek Ferdous, for his patience and unwavering support throughout this long writing process, even during months of radio silence on my end. His guidance and tolerance have been truly invaluable.

Shoutout to my closest friends—Rica, Paim, Saif, Asif, and Umama—for enduring my endless frustrations and rants about almost giving up multiple times, and patiently listening as I read my drafts over and over again. Your support made this journey so much more bearable.

Finally, a big thanks to the countless informative Youtube videos and blog posts that helped me understand some of the difficult concepts, and to the Ethereum stack exchange community for sharing the same confusions I had, except with answers.

Table of Contents

List of Figures

List of Tables

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

key Public key

key^{-1} Private key

RR Transaction receipts trie root

sig Signature

SR State trie root

TR Transactions trie root

BLS Boneh-Lyn-Shacham signature algorithm

DLT Distributed Ledger Technology

ECC Elliptic Curve Cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

EOA Externally Owned Account

EVM Ethereum Virtual Machine

FSM Finite State Machine

MPT Merkle Patricia Trie

P2P Peer to Peer

PoS Proof of Stake

PoW Proof of Work

UTXO Unspent Transaction Output

Chapter 1

Introduction

A blockchain is a public ledger that records digital transactions in interconnected blocks maintained across computers in a peer-to-peer[1] network. It is a Distributed Ledger Technology (DLT) [16] where the transactions are distributed across the network without any central controlling body, ensuring transparency, security, and immutability. The transactions could be used to either transfer digital currency called cryptocurrency or data without any authoritarian restrictions, geographic constraints, or interference from legal systems. The synchronization of the blocks carrying these transactions are done through a set of algorithms called the consensus algorithms, making it impossible to alter or forge coins without the alteration of all subsequent blocks. Each coin in the system needs to be validated and consistent with everyone's record of the ledger.

Blockchain technology was first introduced with Bitcoin as a solution to constraints in traditional financial systems, such as trust reliance, high transaction fees, and lack of transparency, through the use of cryptocurrencies [29]. Through fluctuations, its popularity and demand began to rise, increasing bitcoin's economic value, making it an acceptable exchange currency in many businesses. Soon the developers realized this technology can have other applications, opening the door to decentralized apps using pieces of code called smart contracts introduced in Ethereum. Transfer of data allowed the exchange of digital assets, or NFTs (Non Fungible Tokens) which bought blockchain systems into pop culture. Even though Bitcoin was launched in 2009 [11], it still remains a widely used and accepted digital currency, while Ethereum has become one of the most popular blockchain platforms due to its complexity, varity of applications, versatility and programmability. That is why this thesis focuses on these two leading platforms and present their mathematical modeling for an abstract understanging of their core functions.

1.1 Motivation

Being the two most influential blockchain systems, understanding Bitcoin and Ethereum is important for grasping DLT concepts. While comprehensive resources such as Mastering Bitcoin book [6], Mastering Ethereum book [7], and the official

[1] A Peer-to-Peer network is a decentralized network where the participants have the same privileges and responsibilities

Ethereum Developer Documentation [33] provide in-depth technical details, they often lack abstract mathematical models to represent key blockchain concepts and focus on implementation and applications. This creates a barrier for engineers and researchers who wish to analyze or optimize blockchain systems at a structural level without diving into code or intricate implementation details. The Ethereum yellow paper [3] on the other hand provides deep mathematical formulations, but its definitions are often complex and difficult to interpret.

Our approach aims to bridge this gap by offering simplified, structured, and mathematically grounded models of Bitcoin and Ethereum. These models would help grasp a general understanding of how blockchain systems work, helping to identify areas for optimization in scalability, efficiency, and security. This could also help mathematicians understand this technology and help in areas like, improving the cryptographic algorithms used to secure the blockchain network, etc. Students studying Number theory could read this and see how the theory they learned is being applied in interesting current technology.

Ethereum is often described as a transaction-based state machine, yet this representation is rarely visualized or explained clearly. We will address this by representing Ethereum as a finite state machine (FSM) where the state transitions due to different conditions of the executing transactions. This will help better analyze Ethereum.

The abstraction that this thesis will provide could help beginner researchers, mathematicians and engineers to work together in analyzing, improving, and innovating on the DLT without being overwhelmed by the complicated definitions, intricate architecture, and extensive code repositories.

1.2 Research Objectives

The main goal of this research is to simplify and abstractify blockchain concepts by mathematically modeling the two most popular blockchain systems. To achieve this, following are some research objectives that the thesis will try to cover:

1. Establish the cryptographic foundation of Bitcoin and Ethereum using mathematical functions and proofs where necessary. Explain how it is used for security, to later reference this in our model.
2. Construct a mathematical model for Bitcoin, including its fundamental components, transaction processing, block creation, verification and synchronization in the network. Provide simplistic algorithms to illustrate these processes.
3. Construct a mathematical model of Ethereum with Proof-of-Work (PoW)[2] based consensus, highlighting its structural difference from Bitcoin. Simplify the additional components of Ethereum and explain how they work together to enable the extra functionalities.
4. Represent Ethereum's state transitions using a Finite State Machine (FSM), showing clear transitions between different states triggered by transaction execution.

[2]PoW is a consensus mechanism based on the use of computation power to mine a block.

5. Extend the PoW based Ethereum model to the mathematical modelling of Ethereum 2.0, incorporating the Proof-of-Stake (PoS)[3] consensus mechanism and how they operate in contrast to its previous version.

1.3 Outline

The thesis is structured in the following way to cover the objectives mentioned in the above section:

Chapter 2 goes over the background starting with a general concept of blockchain systems, followed by an overview of Bitcoin and Ethereum. It highlights their main use cases in the current world and main functionalities, including transaction handling (UTXO vs account model), block mining, and how consensus is reached in the network. Three different consensus mechanisms are discussed briefly. The chapter also talks about the use of cryptography, especially elliptic curve cryptography with mathematical definitions, to give unique, verifiable identities for users joining a blockchain network.

Chapter 3 presents the Literature review on the thesis topic, showcasing existing researches done on the mathematical modeling of Bitcoin and/or Ethereum. Key findings will be highlighted and how this thesis can contribute to the field. Due to the niche nature of the topic, only a limited number of related papers were found for this section.

Chapter 4 forms the first part of the thesis's main content: the mathematical modeling of the Bitcoin system. It starts by defining the fundamental components of the blockchain such as entities, transactions and blocks. Functions are then provided in algorithmic form to illustrate the step-by-step processes of block creation, distribution, verification, and consensus. Key terms used in defining the structure of Bitcoin blockchain components are summarized in tables.

Chapter 5 follows a similar approach to mathematically model Ethereum, starting with its initial version and its addition components, including transaction receipts, the Ethereum Virtual Machine (EVM), and its ability to execute Turing-complete programs called smart contracts. This version uses the Proof-of-Work (PoW) consensus mechanism. The thesis then models Ethereum's transition to its second major version, which introduces a completely different consensus mechanism, Proof-of-Stake (PoS), alongside new structures and functionalities like attestations, slashings, etc. Between these two version models, we present the life cycle of a smart contract and a finite state machine (FSM) representation to demonstrate Ethereum's transaction-based state transitions.

Chapter 6 evaluates the extent to which the thesis fulfills its research objectives from Section 1.2, provides a comparative analysis with related studies, discusses limitations and explores future research directions. These include developing a generalized model applicable to multiple blockchain platforms and finding better cryptographic algorithms.

[3]PoS is a consensus mechanism where a block miner is selected based on a stake and probability.

Chapter 7 concludes the thesis by summarizing its findings and highlighting its potential contributions.

A key convention in this thesis is the use of angle brackets $\langle ... \rangle$ to represent ordered sets and tuples (immutable ordered sets). These and other visual representations will be used to simplify the complex data structures used in the actual Bitcoin and Ethereum systems.

Chapter 2

Background

A blockchain is a decentralized ledger consisting of a sequence of blocks, each containing a list of verified transactions. A transaction is the atomic unit in a blockchain system by which an entity (a user or a smart contract) transfers a certain amount of the underlying cryptocurrency and/or data to another entity. This ledger is maintained by a network of nodes, ensuring transparency where everyone has all the necessary data and verifiability to maintain synchronization between them without requiring a central authority. Each block is cryptographically linked to the previous one through a hash, for data integrity and immutability, except for the very first block in the chain, known as the *genesis block*. That is, once the transactions included in block are linked to the blockchain, they cannot be altered without dropping that block and all the blocks after it. Figure 2.1 shows the chained structure of a blockchain. Consensus mechanisms synchronize the addition of new blocks across the network, ensuring agreement among the nodes.

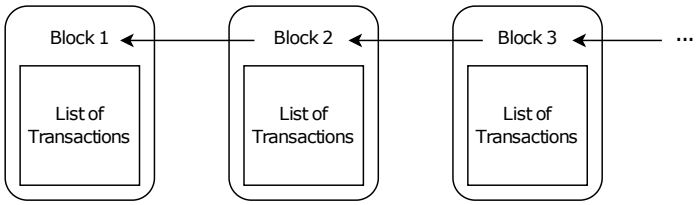

Figure 2.1: Blocks in a Blockchain system

This section provides a brief overview of the Bitcoin and Ethereum networks, the two blockchain platforms modeled in this thesis. It covers their fundamental principles, cryptographic foundations and consensus mechanisms to help establish a basis for the mathematical models.

2.1 Bitcoin

The concept of Bitcoin was introduced in 2008 by a pseudonymous person named Satoshi Nakamoto in a white paper [1][60]. The aim was to have a digital currency without the need of any intermediaries. Instead the cryptocurrency[1] is maintained

[1]Bitcoin's underlying currency is abbreviated as BTC

by every person who joins the bitcoin network and all the peer to peer transactions within the network are recorded and synced across their devices. No one node or company is in control of it and anyone can join by downloading the Bitcoin software, making it decentralized unlike traditional banking. This allows users to send money anywhere across the world through the internet with maximum control, speed and absence of geographical restrictions. Many companies accept bitcoin as a medium of exchange with its value depending on its supply and demand. When there is more demand for bitcoin, the price goes up and vice versa [59].

The Bitcoin P2P network consists of nodes that join it and depending on their responsibilities in the network, there are two types. General nodes handle transactions and miner nodes are responsible for creating new blocks through a process called mining. After joining the network, a node can send or receive Bitcoin (native cryptocurrency) through transactions that utilizes the UTXO (Unspent Transaction Output) model, where the sender uses unspent outputs from previous transactions addressed to them and creates new outputs for the recipient. These transactions are propagated across the network, where miner nodes collects and combines them into a block. Miners compete to solve a cryptographic puzzle using their computation power as "work" in the Proof of Work (PoW) consensus mechanism. The first miner to solve the puzzle, propagates their block across the network, earning themselves newly generated bitcoins as a reward for their effort. Other nodes validate the block and add it to their blockchain.

Due to the decentralized nature of this mining process, occasionally multiple valid blocks are mined simultaneously, causing branches in the blockchain called *forks*. This is resolved by letting the branches grow and then keeping the longest chain or the branch with the highest "work" put in it. The orphaned branches are discarded, resulting in only the miners in the main branch receiving their Bitcoin rewards. A distributed consensus emerges in the network when majority of the nodes agrees on the same blocks on their main branch.

2.2 Ethereum

Evolving from the Bitcoin ledger, Ethereum introduced a more advanced blockchain that facilitates the deployment and execution of immutable and irreversible computer programs, known as smart contracts written in a Turing complete language. Ethereum features a virtual machine, called the Ethereum Virtual Machine (EVM) which executes these smart contracts and records changes to its state in the ledger alongside additional data and coin transactions. The Ethereum platform also offer decentralized applications (dApps) which are digital applications or programs that run on the decentralized EVM rather than a single computer or server. One of the most popular dApps is OpenSea where users can trade or collect digital assets or Non fungible tokens (NFTs). These NFTs are unique assets that cannot be replaced or exchanged and are stored in the blockchain through transactions. With these facilities on top of transferring of coins, Ethereum soon became popular and still remains one of the most popular cryptocurrencies on the market.

Unlike Bitcoin, which uses the UTXO model where balance is calculated from the outputs in the blockchain, Ethereum uses the account model where each account

directly tracks its balance stored in the blockchain. Smart contracts in Ethereum are deployed or executed via transactions and each transaction in Ethereum involves a computational fee called gas, paid in Ether[2] (Ethereum's native cryptocurrency). This gas fee ensures efficient resource utilization and prevent misuse of computational resources. Executing transactions in the EVM produces outputs that changes its state and thus, Ethereum can be defined as a transaction based state machine. We will later represent this feature of the EVM using a finite state machine in our proposed Ethereum model.

Initially, Ethereum used PoW (Proof of Work) like Bitcoin to mine blocks and ensure agreement among all the nodes, but later transitioned to PoS (Proof of Stake) along with several updates for better scalability and energy efficiency. PoS is another consensus algorithm to mitigate some of the limitations of PoW, where instead rewarding the miner with the most work done, only certain nodes are allowed to add blocks to the chain.

2.3 Cryptography

When a sender sends a transaction to a receiver, it has to be made safe so that no third person can alter the amount or sender or receiver address. To maintain this data integrity and user privacy, blockchain systems uses cryptographic encryption to keep the transactions secure and temper proof while maintaining trust and transparency for verification [17]. This is part of the 3 triads of security (CIA): Confidentiality, Integrity, and Availability. The transaction data is encrypted by the sender using a key and decrypted by the receiver using another key. Encryption is the conversion of data into a random sequence of bits and decryption is its inverse process. A third person would not be able to interfere with this without the set keys. There are two types of encryptions used in blockchain systems. **Symmetric key cryptography** makes use of a secret key known only by the sender and receiver, whereas **Asymmetric key cryptography** makes use of two keys.

Both Bitcoin and Ethereum uses Asymmetric Key Cryptography to encrypt and decrypt data using a pair of keys called private key and public key. Specifically, **Elliptic Curve Cryptography** or ECC is used, which has a set of algorithms that make use of elliptic curves to secure data and ensure transactions are verifiable and fraud-proof.

2.3.1 Private key, Public key generation

Entities participating in blockchain networks need a **private key** that only they will know for accessing and managing their accounts or initiating transactions. This key is randomly generated for the entity when they first enter the network. A corresponding public key, derived from the private key, is known and used by others to verify operations initiated by the entity. This public-private key generation has to be secure so that, one cannot get to the private key with the public key.

Bitcoin and Ethereum uses the same elliptic curve called *secp256k1*, which has the

[2] 1 Ether or 1 ETH = 10^{18} Wei

following equation over a finite field $\mathbb{F}_p$ where p is a very large prime number ($\approx 2^{256}$).

$$y^2 = x^3 + 7 \pmod{p} \tag{2.1}$$

Point addition on the elliptic curve: Given two points P and Q on the curve and a third intersection point R between the curve and the line going through P and Q. The point addition of P and Q is the reflection of R on the x-axis denoted as R' and can be written as $P + Q = R'$.

If $P = Q$, the line is tangent to the curve at P, giving us $2P = R'$ with point doubling. This can be represented graphically in following Figure 2.2.

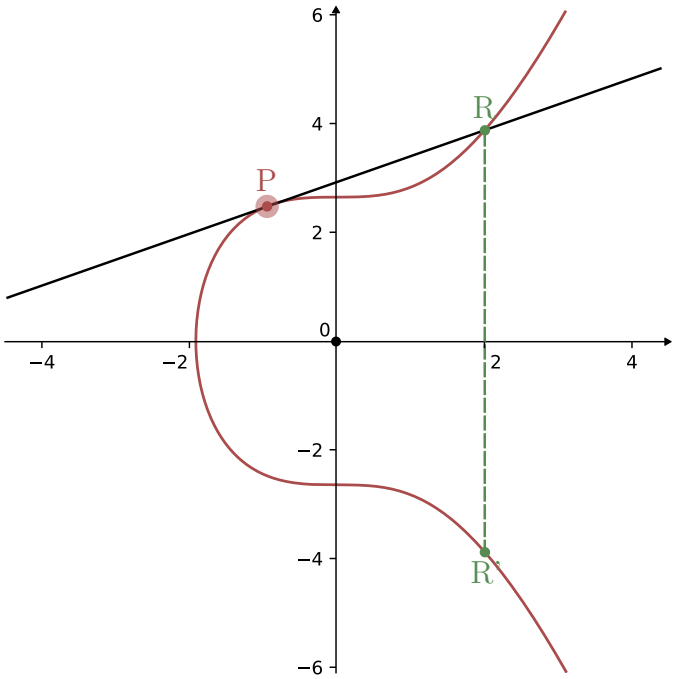

Figure 2.2: Elliptic curve point doubling on $y^2 = x^3 + 7$

Thus we can perform point multiplication, $n \cdot P$ by successively adding P to itself along the curve.

$$n \cdot P = P + P + ... + P$$

After randomly generating the 256-bit private key, d, the public key Q is calculated by point multiplying a predefined generator point G on the elliptic curve by d. [6]

$$Q = d \cdot G \tag{2.2}$$

Even though d is very large, this point multiplication will take very few steps (maximum 510 steps[3]) using the **Double and Add Algorithm**, compared to naively point adding G (maximum 2^{255} steps) [54]. We can summarize the steps of this algorithms in the following way:

1. First find the points $2G$, $4G$, ..., 2^nG successively using point doubling, where $n = \lfloor \log_2 d \rfloor$. Since Bitcoin and Ethereum uses 256-bit private key, $n \leq 255$.

[3] 255 point doubling steps, plus 255 point addition steps

2. Use d's binary to decimal expression, $d = \sum_{i=0}^{n} a_i 2^i$ where $a_i \in \{0, 1\}$.

3. Combine step 1 and 2 to get Q by point adding in,

$$Q = d \cdot G = a_0 G + a_1(2G) + ... + a_n(2^n G)$$

The graph represented in Figure 2.2 is not defined over a finite field. In reality, the *secp256k1* curve is on a large prime-order finite field $\mathbb{F}_p$, where all operations are performed with modulo p. This modular structure ensures that, even if someone were to try to derive the private key d from the known generator point G and public key Q, it would be computationally infeasible. This is due to the hardness of the elliptic curve discrete logarithm problem (ECDLP) over the finite field $\mathbb{F}_p$.

For key generation, compared to other algorithms, ECC gives high security with smaller key sizes, making it efficient and preferable for blockchain applications. A 256-bit ECC key gives the same amount of security as a 3072-bit RSA key [40].

2.3.2 Hashing

Hashing is a mathematical process that transforms input data of arbitrary length into a fixed length fingerprint of that data. It is a one-way function: easy to compute, but computationally infeasible to reverse (find the original input) or to find two distinct inputs that produce the same hash. This way of scrambling data does not require any key and is what contributes to the immutability of the Blockchain by linking blocks through hash pointers and enabling efficient data verification [17].

In blockchain systems the data in a block is hashed into a unique set of bits, which will be used in the next block to point to this block. Any change in the block would alter the hash and disrupt the chain, thus maintaining data integrity. A good hash function should produce significantly different outputs even for the smallest changes in the input and should be computationally infeasible to reverse. Bitcoin uses the *SHA-256* cryptographic hash function and Ethereum uses *Keccak-256*, both of which share a similar design and provides high security to prevent various cryptographic attacks [44]. It is used for generating addresses, hashing transactions, linking blocks, summarizing transactions in merkle tree structures, etc. Throughout this thesis, we use hash functions, written as hash(msg). For example, the public key generated earlier is hashed and the first few bits of it will be considered as the **address** to the account.

2.3.3 Signature generation

In blockchain systems, a digital signature is used to claim that a message carrying some data, belongs to the entity signing it with their private key. Other entities can then verify using the sender's public key that the message and signature indeed came from that sender. This signature generation and verification in Bitcoin and Ethereum is done using **ECDSA (Elliptic Curve Digital Signature Algorithm)** with the same elliptic curve *secp256k1* [6][7]. ECDSA has two parts: signature generation by the sender and verification by other nodes.

1. Generation: As input, this requires the sender's private key, denoted as key_s^{-1} and the message being signed.

$$(key_s^{-1},\ \text{message}) \rightarrow sig \tag{2.3}$$

A signature is composed of two components: $sig \triangleq \langle r, s \rangle$

First, the message is hashed: $h = \text{hash(message)}$. Then a random number q is generated, paired with its temporary public key $Q = (x, y)$ following previously defined steps of $Q = q \cdot G$ (point multiplication). Using these values, the signature components are thus calculated in the following way:

$$r = x \pmod{p}$$

$$s = q^{-1} \times (h + key_s^{-1} \times r) \pmod{p} \tag{2.4}$$

Here p is from the finite field elliptic curve with mod p in *secp256k1* and q^{-1} is the mod p inverse of the temporary private key q [42]. Hence, we have:

$$sig = \langle r, s \rangle$$

2. Verification: This takes the sender's public key, denoted as key_s, the signature (r, s), and the message as input and outputs either true or false.

$$(sig,\ key_s,\ \text{message}) \rightarrow True/False \tag{2.5}$$

First, hash the message to h, compute s^{-1} of mod p from s in sig assuming it is invertible and then compute these two scalar values:

$$u_1 = h \times s^{-1} \pmod{p} \quad \text{and} \quad u_2 = r \times s^{-1} \pmod{p} \tag{2.6}$$

Use these to compute the point Q on the curve:

$$(x, y) = u_1 \times G + u_2 \times key_s \tag{2.7}$$

Verify if $r = x \pmod{p}$. If true, the signature is valid, otherwise it is not [42].

Proof of how Equation 2.7 is equivalent to Q for the verification of r:

$$\begin{aligned}
(x, y) &= u_1 \times G + u_2 \times key_s \\
&= \{h \times s^{-1} \pmod{p}\} \times G + \{r \times s^{-1} \pmod{p}\} \times key_s \\
&= h \times s^{-1} \times G \ + \ r \times s^{-1} \times (key_s^{-1} \times G) \pmod{p} \\
&= (h \ + \ r \times key_s^{-1}) \times s^{-1} \times G \pmod{p} \\
&= (h \ + \ r \times key_s^{-1}) \times \{q \times (h + key_s^{-1} \times r)^{-1}\} \times G \pmod{p} \\
&= q \times G \pmod{p} \\
&= Q \pmod{p}
\end{aligned} \tag{2.8}$$

This process ensures that only the entity with the correct private key can generate the signature, and anyone with the corresponding public key can verify it.

2.4 Consensus algorithms

A consensus algorithm is a fundamental component to synchronize a distributed ledger across multiple nodes.

1. **Proof-of-Work (PoW):** This algorithm, introduced in Bitcoin [5], requires miners to solve a cryptographic puzzle by producing a hash that satisfies certain properties to mine blocks. Hence each block added to the main chain is a proof of the computational work done by the miner to mine that block. While secure, solving such a cryptographic puzzle demands a lot of computational power and thus consumes a lot of electricity.
2. **Proof-of-Stake (PoS):** Here, nodes willing to participate in block creation must prove that they own a certain amount of cryptocurrency, called a stake, locked into an escrow account. The stake acts as a guarantee that they will behave as per the protocol rules. Only stakeholder nodes or validators are allowed to propose blocks in the network. Unlike PoW, miners can lose their stake for misbehaving. Furthermore, PoS is more energy-efficient than PoW and can improve decentralization [9].
3. **Delegated PoS (DPoS):** A variant of PoS in which stakeholders votes for a small number of delegates who can propose blocks, reducing the number of active validators but an increased centralization [2].

These algorithms form the backbone of blockchain platforms, governing block validation and network synchronization.

Chapter 3

Literature Review

Research on Blockchain systems is a growing area focused on defining, analyzing and refining their mechanisms, consensus protocols and cryptographic security. However, few of these studies attempts to provide mathematically rigorous models of these systems, but still leaving gaps in clarity and formal understanding. Since only 6 relevant papers were found, each of their contribution to the topic will be talked about in separate sections.

3.1 The Mathematics Behind Cryptocurrencies and Blockchain - Jiménez-Ayala and Medina [12]

The paper by Jiménez-Ayala and Medina [12] starts off by pointing out the importance of formally presenting the mathematical foundation of cryptocurrencies and its security, particularly for students interested in real-world applications of number theory and algebraic structures. It then describes blockchain as a system of "linked unalterable databases" [12], followed by a historical overview of bitcoin and briefly discusses block mining without delving into the specific consensus mechanisms.

The main focus of this paper are the cryptographic operations but before getting into that, for a refresher, concepts of modular arithmetic like finite fields, equivalent relations, quotient sets and modular operations were explained with examples. Cryptographic fundamentals, including symmetric and asymmetric encryption, the RSA algorithm, the Diffie-Hellman algorithm, and the discrete logarithm problem, which forms the basis of ECC used in Bitcoin and Ethereum, were discussed. The study graphically illustrates point addition on the *secp256k1* curve and the mathematical formulations using the double and add algorithm to compute $Q = nP$.

Despite its mathematical depth, the paper does not connect these cryptographic algorithms to blockchain operations. It doesn't discuss hashing, keys and digital signatures generation, and other important aspects of blockchains that can be a modeled as well.

3.2 Mathematics and Data Structures in Blockchain and Ethereum - Hoseini [10]

Hoseini [10] presents a master's thesis that provides a detailed mathematical and algorithmic study of Ethereum, making it one of the most relevant works for our thesis. It explores Ethereum's cryptographic mechanisms, data structures, and attempts to formalize it as a transaction-based state machine, which is similar to our approach, but lacks depth in several areas.

The paper starts with a historical overview of blockchain as a fusion of multiple innovations like PoW, time stamping, mining, digital signatures, and privacy in a P2P network. It covers hash functions, hashcash PoW, and ECDSA for determining the authenticity and authorship of digital assets, along with a brief mention of Lamport Signatures for quantum resistance. The work also touches on Zero-Knowledge Proofs (ZKPs) as an interactive proof system, and algorithms for all of them.

The discussion on Ethereum's data structures includes linked lists for blockchain storage, Merkle trees and tries for efficient verification, their distinction, and Bloom filters for large-scaling data handling. The paper describes Ethereum as a multilayered, open source, programmable and cryptographically secure system, including its P2P network and the Recursive Length Prefix (RLPx) protocol for secure communication through encryption. Ethereum's block structure and the PoW mechanism are presented but it remain focused on of Ethereum 1.0 with only a brief comparison of PoS to PoW. While it introduces FSM for a mathematical formalization of Ethereum's behavior, it only presents a general five-tuple definition and a basic state transition diagram, lacking details on individual transaction execution across different transaction types and failure states.

Ethereum's economic model is covered, explaining Ether, gas prices, and tokens or tradable digital assets. The Ethereum Virtual Machine (EVM) is briefly introduced, followed by an overview of identity generation using ECC over finite fields, represented with general elliptic curves notions, graphs and equations. The paper then details Ethereum accounts and transaction structures but skips other components like receipts and logs. Smart contracts which is an important part of the EVM, are also introduced briefly without delving into details like, how they are created and executed. The thesis concludes with a discussion for potential improvements in Trie structures, smart contracts, and ECC algorithms.

Overall, this master's thesis provides a strong mathematical foundation for Ethereum, covering cryptographic algorithms, consensus mechanisms, and data structures, while comparing them with the flaws of Bitcoin. However, it lacks a structured, chronological description of Ethereum's operations, omits Ethereum 2.0 which has is one of the most significant developments in Ethereum history, and does not fully formalize its FSM model. These gaps leave room for further refinement and will be some of the key contributions of our research.

3.3 Mathematical Modeling Approaches for Blockchain Technology - Suresh Kumar et al. [15]

Suresh Kumar et al. [15] introduces blockchain concepts by going over its functionalities and definitions in their paper. Despite the attempt, it lacks clear explanations of fundamental mechanisms such as block mining, digital currency, and consensus algorithms. The paper presents blockchain as a layered structure — "Accord layer", "Mining layer", "Spread layer", "Semantic layer" and "Application layer" - but fails to provide a meaningful breakdown of these layers and how they interact.

The study briefly mentions smart contracts, referring to them as "PC programs" "containing a lot of rules" [15] and compares dApps with centralized apps. On the cryptographic side, the paper touches on hashing, Merkle trees, and elliptic curve cryptography (ECC) with a few mathematical formulations but does not link its role in transaction validation and security. Ethereum and Bitcoin are lightly compared in terms of functionalities and punishment systems, yet key distinctions in their consensus models are left unexplored. Their excessive use of synonyms for even important terms like "squares" instead of "Blocks" creates further confusion.

Overall, this study lightly introduces the tech, but lacks structured definitions, clear consensus modeling, and precise cryptographic formulations. This highlights the need for a well-defined mathematical approach to understanding blockchain mechanisms.

3.4 The Mathematics of Bitcoin - Grunspan and Pérez-Marco [13]

This paper by Grunspan and Pérez-Marco [13] presents Bitcoin as a "mathematical algorithm on a network which manages transaction data and builds majority consensus among the participants" [13]. Instead of focusing on the mathematics behind Bitcoin's cryptographic tools, it analyses research on Bitcoin from a statistical perspective.

The paper introduces nodes, miners, their roles in the Bitcoin network and explains key operations, including PoW, hashing, and difficulty adjustment, before applying probability functions to analyze them. First, they mathematically models the time taken to mine a block and deduces that the probability of mining blocks using PoW follows a Poisson distribution. Then the paper models the probability of a successful double-spending attack where a dishonest node tries to alter past transactions by outpacing honest miners, showing that the probability is very small. Similarly a few more probability models for mining strategies and their profitability are analyzed, concluding that such attacks are realistically unlikely.

Overall, the paper demonstrates how, given a clear model, a blockchain architecture's security risks can be assessed using probability functions. Our thesis could similarly help formalize Bitcoin and Ethereum 1.0/2.0 to analyze potential risks and mitigation strategies.

3.5 The Mathematics Behind Blockchain - Sutra Dhor [57]

This short paper by Sutra Dhor [57] starts off by comparing blockchain with centralized systems and then introducing some bitcoin features like PoW, hashing (without explaining its role in security and integrity), and ECC with a graphical representation of point doubling. It briefly mentions ECC's vulnerability to quantum attacks but does not explain why it is quantum-unsafe. The paper presents ECDSA equations but without the proof, outlines Bitcoin's block structure without the header, and mentions nonce usage in PoW while skipping hashing and difficulty adjustment. It also lacks transaction structure details and algorithms, focusing primarily on Bitcoin despite claiming to discuss blockchain. Overall, the paper demonstrates several gaps that need to be addressed.

3.6 Ethereum: A Secure Decentralised Generalised Transaction Ledger - Wood et al. [4]

This paper by Wood et al. [4] is the official Ethereum Yellow Paper, introducing and formally defining its protocol. It presents Ethereum as a transaction based state machine, where the state of the accounts (world state) σ_t changes based on transactions T through a state transition function Υ (transaction execution):

$$\sigma_{t+1} = \Upsilon(\sigma_t, T)$$

The paper defines accounts, blocks and transactions using sets and tuples, with brief descriptions of their fields. Trie structures, gas costs and other mappings are expressed with well-defined functions, providing a set-theoretic approach that we can use as a reference. However, as the paper progresses, it becomes more complex and harder to follow without prior knowledge of blockchain concepts. For example, the Ethereum Virtual Machine (EVM) is defined as a 6-tuple, assuming familiarity with operating systems and low-level computation terminologies.
The block structure mixes elements from both Ethereum 1.0 and 2.0, likely because the paper focuses mostly on the *execution layer* while providing little discussion on consensus mechanisms, which differ between the 2 versions and are crucial to Ethereum's security and decentralization.

For Ethereum's Merkle trie, the paper introduces: $\text{TRIE}(L_1^*(\sigma[a]_s)) \equiv \sigma[a]_s$
where L_1^* is referred to as the collapse function but is not well explained. Understanding this requires prior knowledge of Ethereum's trie structures. As a result, mathematicians may struggle with the missing context, while engineers and developers might prefer reading the code rather than engaging with the formal mathematical details. This suggests that the Yellow Paper is mainly written for computer scientists.

In conclusion, the paper presents Ethereum as a purely functional state-transition system aimed for higher-level readers. We will use it as a guide to model Ethereum using sets, algorithms and diagrams while ensuring clarity and simplicity.

3.7 Research Gap

From the reviewed papers, we have identified several research gaps that this thesis will try to address. They are as follows:

1. Lack of a formal mathematical model covering the entire Bitcoin and Ethereum process, from joining the network to block synchronization, incorporating both the execution and consensus layers.
2. Lack of clarity and structured chronology in existing blockchain models, making them difficult to follow.
3. Lack of studies linking cryptographic techniques to corresponding blockchain operations, despite their fundamental role in security and transaction validation.
4. Lack of a step by step, simplified explanation of smart contract deployment and execution in Ethereum's mathematical models, for better understanding of it's functionalities.
5. Absence of a detailed functional state-transition model for Ethereum with a Finite State Machine (FSM) that accounts for different transaction types.
6. Lack of studies on the Ethereum 2.0's structure and mathematical modeling.

By addressing these gaps, this thesis will contribute a well-defined, structured, and precise mathematical model for Bitcoin and Ethereum.

Chapter 4

Proposed Bitcoin Blockchain Model

In our model we consider that there are many available blockchain systems. We use the notation BC to denote the set of blockchains where $bc \in BC$ denotes a single particular blockchain. We will model a specific blockchain $bc \in BC$ starting from the fundamental components used in the network.

4.1 Bitcoin Network Components

The fundamental components to create the bitcoin network are the entities, transactions and blocks. We assume that each entity utilizes a computing device called *node* to interact with the blockchain system, however, we do not differentiate between an entity and their corresponding node. Instead they are considered as a single entity for simplicity. It is to be noted that we assume a node joins and maintains its connectivity with the P2P network of a blockchain system in a standardized approach as required by the corresponding P2P network. This is why we do not include such functionalities in our model.

4.1.1 Entity

An entity in our model represents a person which interacts with the blockchain system using a node. Each user or entity in a blockchain system is identified using an identifier called **address**. To get an address in a network, a user utilizes the respective wallet software to generate a key pair, consisting of a private key and the public key as mentioned in Section 2.3. Then, the public key is used to generate the address of the user. These three values, private key, public key and the respective address forms the identity of the user/node in the bitcoin network.

4.1.2 Transaction

In the Bitcoin network, a transaction is the atomic unit by which an entity transfers certain amount of bitcoins to another entity. To develop this model, we assume s as the sender and r as the receiver. In case of multiple receivers in a single transaction, we denote them as $r_0, r_1, ..., r_n$.

We use the notation T_{bc} to denote the ordered set of all transactions.

$$T_{bc} \triangleq \langle t_1, t_2, ..., t_n \rangle \tag{4.1}$$

Each single transaction ($t_i \in T_{bc}$) has two arrays of inputs and outputs, where the inputs point to unspent outputs (UTXOs) that were addressed to the sender before and the outputs are for the receivers [61]. We can define them in the following way:

$$t_i \triangleq \langle \mathcal{T}_i^{in}, \mathcal{T}_i^{out} \rangle$$
$$\mathcal{T}_i^{out} \triangleq \langle \mathbf{t}_i^{out_0}, ..., \mathbf{t}_i^{out_j} \rangle$$
$$\mathcal{T}_i^{in} \triangleq \langle \mathbf{t}_i^{in_0}, ..., \mathbf{t}_i^{in_l} \rangle$$

Here, $j, l \in \mathbb{N}$ are the number of outputs and inputs that t_i has, respectively.
For $0 \leq k \leq j$, we define each output in the following way:

$$\mathbf{t}_i^{out_k} \triangleq \langle key_{r_k}, val_k \rangle \tag{4.2}$$

Here key_{r_k} is receiver r_k's public key and val_k is the bitcoin amount being sent. For $0 \leq m \leq l$, we define each input in the following way:

$$\mathbf{t}_i^{in_m} \triangleq \langle outHash,\ o,\ sig_s \rangle \tag{4.3}$$

Here $outHash$ is the hash of a previous transaction that has the output being used in this input and o is the index of that output in that transaction. sig_s is the signature of the sender using ECDSA as discussed in section 2.3 that will be used to verify sender's permission to use that referenced output.

We have simplified the structure of the transaction input and output by directly placing key_r and sig_s inside them. In reality, they are placed inside two scripts called the *scriptPubKey* and *scriptSig*, using a scripting pattern [38]. In P2PK (Pay to Public key) scripting pattern, which is the simpliest one, the structure of the scripts can be defined in the following way:

$$scriptPubKey_k \triangleq \langle key_{r_k}, \text{OP\_CHECKSIG} \rangle$$

$$scriptSig_m \triangleq \langle sig_s \rangle$$

$scriptPubKey_k$ is the locking script that ensures the receiver can only spend this output addressed to them when they can produce a signature belonging to this public key, key_{r_k}. $scriptSig_m$ is the unlocking script, signed by the sender s to prove ownership of the output they are referencing in the input and thus unlock it's locking script. This is done by executing the two scripts together. Bitcoin uses a stack based scripting language (Turing incomplete) and "OP_CHECKSIG" is one of its opcode with the steps to verify this signature [43]. First the signature in pushed into the first-in-last-out stack data structure, followed by the unlocking script's public key and opcode [37]. It is then executed top to bottom from the stack and based on the verification output, a true or false value is pushed back into the stack. We will represent these verifying steps in the form of an algorithm later.

4.1.3 Blocks

Let B_{bc} be the ordered set of all the verified blocks in a blockchain system bc after reaching consensus:

$$B_{bc} \triangleq \langle b_1, b_2, \ldots, b_n \rangle \tag{4.4}$$

Each block b_p has a header and an ordered set of validated transactions, and can be defined as follows:

$$b_p \triangleq \langle H_p, T_p \rangle \tag{4.5}$$

$$T_p = \{t_{p_i} | i \in \{1, 2, \ldots, n\}\} \tag{4.6}$$

$$H_p \triangleq \langle ts_p, h_p, MR_p, \delta_p, \eta_p \rangle \tag{4.7}$$

$$h_p \triangleq \text{hash}(H_{p-1})$$

Table 4.1 explains the expressions used in the above transaction and block structure.

Expression	**Terminology**
ts_p	Timestamp of when block b_p was created
h_p	Hash of the previous block's header
MR_p	Hash of Merkle root summarizing all t_{p_i}
δ_p	Difficulty target of mining this block b_p
η_p	Nonce which satisfies $\text{hash}(\text{hash}(H_p)) < \delta_p$
key_s^{-1}	Private key of the sender s
key_s	Public key generated from key_s^{-1} using ECC
ad_s	$ad_s = \text{hash}(key_s)$
sig_s	Signature generated from key_s^{-1} and transaction data using ECDSA

Table 4.1: Bitcoin transaction and block component terminologies

To generate the sig_s for a transaction input, the transaction data is serialized with all the sig_s's set to null. This is then hashed and passed into the **ECDSA** using key_s^{-1}, returning the signature sig_s as discussed in Section 2.3.3. The verification of this signature later involves re-hashing and then using key_s [6].

T_{bc} is generated from the union of all the T_p in all the blocks b_p in B_{bc}:

$$T_{bc} \triangleq \bigcup \langle T_p \mid b_p \in B_{bc} \rangle \tag{4.8}$$

For ease of searching through transactions and blocks, we will define 2-way **hashmap functions** that map a hash to their corresponding transaction or block:

$$TxnHash = \{(\text{hash}(t), t) \mid t \in T_{bc}\} \tag{4.9}$$

$$BlockHash = \{(\text{hash}(b.H), b) \mid b \in B_{bc}\} \tag{4.10}$$

4.1.4 UTXO Set

UTXO's are outputs from all the transactions in B_{bc} that have not yet been referenced in any of the inputs. This prevents entities from double spending and maintains the program's integrity [61]. We will define the UTXO set in the following way:

$$UTXOset = allOuts - getUTXO(allIns) = \{\mathbf{t}_q^{out}, \dots\} \tag{4.11}$$

where,

$$allOuts = \bigcup\{\mathcal{T}_i^{out} \mid t_i \in T_{bc}, \text{ where } t_i = \langle \mathcal{T}_i^{in}, \mathcal{T}_i^{out} \rangle\}$$
$$allIns = \bigcup\{\mathcal{T}_i^{in} \mid t_i \in T_{bc}, \text{ where } t_i = \langle \mathcal{T}_i^{in}, \mathcal{T}_i^{out} \rangle\}$$

We can write the element-wise function of *getUTXO(allIns)* as $getUTXO^*(\mathbf{t}^{in})$ where $\mathbf{t}^{in}$ is a single input. First the transaction that has the referenced output is found using the transaction hashmap function and then the output at index $\mathbf{t}^{in}.o$ is returned. This is shown in Algorithm 1 and visually in Figure 4.1.

Algorithm 1 Retrieving UTXO from a transaction input

1: **function** GETUTXO*($\mathbf{t}^{in}$)
2: $txn \leftarrow TxnHash(\mathbf{t}^{in}.outHash)$
3: $\mathbf{t}^{out} \leftarrow txn.\mathcal{T}^{out}.atIdx(\mathbf{t}^{in}.o)$
4: **return** $\mathbf{t}^{out}$
5: **end function**

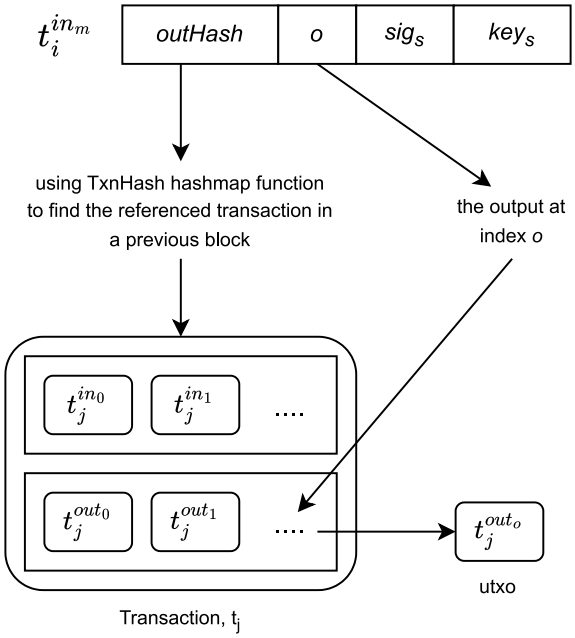

Figure 4.1: Retrieving the UTXO from the transaction input

This UTXO set is updated every time a new block is added to B_{bc} using the above steps to add the new outputs and remove the UTXOs referenced in the new inputs.

4.1.5 Node

As mentioned before, for the sake of simplicity, we use the term nodes to refer to any entity. Nodes are the technical components that participate in a blockchain network. Each node stores and maintains a *blockchainState*, which is basically the ordered set B_{bc} that we defined earlier. It comprises of all the blocks mined and validated in the blockchain. All nodes also keep a *UTXOset* and update it accordingly.

Some nodes in a blockchain system may also perform mining activities, and we will call them miner nodes.

$$M = \{m_0, m_1, \ldots, m_n\}$$

where $M \subseteq N_{bc}$ and N_{bc} is the set of all nodes participating in bc.

4.2 Block Generation

When a transaction is created by a node, it is broadcasted to rest of the nodes in N_{bc}. Each node n collects these unconfirmed transactions in a set called the **mempool** (not global) [28].

$$mempool_n = \{t_1, t_2, \ldots, t_i\}$$

Let $m \in M$ be a miner in the blockchain bc. m takes a subset of transactions from their *mempool*,

$$txnSet \subseteq mempool_m$$

The miner then verifies each transaction in this set. Each input in a transaction are individually parsed to retrieve their corresponding output using $getUTXO(input)$ and then, ECDSA is used to verify sender's signature, ensuring permission to spend the mentioned UTXO. This helps avoid double spending. For the message part of ECDSA, a copy of the transaction is made with all the signature fields set to null and then hashed. Total value in all the outputs in the transaction are ensured to be equal or less than total value in all the inputs. If all checks pass, the transaction has been verified or declined otherwise. These steps are put in the function defined below and shown in Algorithm 2 :

$$\begin{gathered} checkTxn : txnSet \times B_{bc} \rightarrow \{True/False\} \\ verifiedTxnSet = \{t \in txnSet \mid checkTxn(t) = True\} \\ verifiedTxnSet \subseteq txnSet \end{gathered} \tag{4.12}$$

Algorithm 2 Bitcoin checkTxn

```
1: function CHECKTXN(t)
2:     inputs, outputs ← t.items()
3:     val_in, val_out ← 0, 0
4:     t* ← t
5:     t* = makeSignNull(t*)
6:     for all input ∈ inputs do
7:         utxo ← getUTXO(input)
8:         if utxo ∉ UTXOset then return False
9:         end if
10:        if ECDSA(input.sig_s, utxo.key_r, hash(t*)) is False then
11:            return False
12:        end if
13:        val_in = val_in + utxo.val
14:    end for
15:    for all output ∈ outputs do
16:        val_out = val_out + output.val
17:    end for
18:    if val_out ≤ val_in then
19:        txnFee = txnFee + (val_in − val_out)        ▷ outside variable for reward
20:        return True
21:    else
22:        return False
23:    end if
24: end function
```

The verified transactions are then hashed repetitively, creating a binary tree data structure, called the **Merkle Tree** [26], which will be later used to verify the list of transactions. Figure 4.2 shows a Merkle tree produced from an example transaction set, $verifiedTxnSet = \{t_1, t_2, t_3, t_4\}$. The following function returns the root hash of this Merkle tree:

$$merkleRoot : verifiedTxnSet \rightarrow MR \tag{4.13}$$

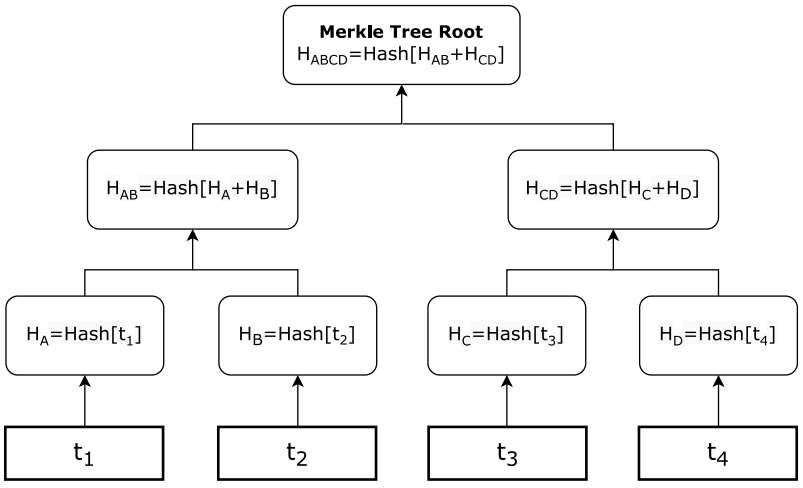

Figure 4.2: Merkle Tree of a verifiedTxnSet = {t1, t2, t3, t4}

Hash of previous block h_p is found by hashing the head of the last block in the chain in B_{bc}. **Difficulty** δ is computed from how many blocks were mined in a time period, to make sure that one block is mined every 10 minutes on average. It is adjusted every 2016 blocks in B_{bc} using the following equation:

$$\delta_i = \delta_{i-1} \times \frac{t}{2016 \times 10 \text{ mins}}$$

where δ_{i-1} is the last difficulty and t is the time taken in minutes to mine the last 2016 blocks [45]. We can put this calculation behind a function,

$$\delta = \textit{calcDifficulty}(\textit{timeStamp})$$

Nonce η is initially set to 0 and will be altered later during mining [14].

Hence, the miner will have a block with header information as defined above and *verifiedTxnSet* as the array of transactions inside the block. To mine this block, the miner sets nonce to 0 and goes through a series of incrementing nonce and hashing the header until $\textit{hash(hash}(H)) \leq \delta$. Then, the block is mined. This process of creating a block b can be explained in the pseudo-code in Algorithm 3.

Algorithm 3 createBlock (PoW)

```
1:  function createBlock(txnSet, prevHash)
2:      T ← {t ∈ txnSet | checkTxn(t) = True}
3:      t_0 ← createRewardTxn(ad_m, T)          ▷ ad_m is the miner's address
4:      T = {t_0} ∪ T
5:      MR ← merkleRoot(T)
6:      h ← prevHash
7:      ts ← currentTime()
8:      δ ← calcDifficulty(ts)
9:      η ← 0
10:     H ← tuple(ts, h, MR, δ, η)
11:     H ← solveBlock(H)
12:     return b ← (H, T)                       ▷ Hence, a new block b has been mined
13: end function
14:
15: function solveBlock(H)
16:     H.η ← 0
17:     while hash(hash(H)) > δ do
18:         H.η += 1
19:     end while
20:     return H
21: end function
```

The function $\textit{createRewardTxn}(ad_m, T)$ used in Algorithm 3, outputs the reward transaction, also known as the coinbase transaction, addressed to the miner m. This is Proof of Work's incentive for mining a block where the value of the reward is equal to the block reward plus the total transaction fee from all the transactions included in the block. The block reward is halved after every 210000 blocks [41]. After successfully creating and mining the block, the miner will append this b to their B_{bc} set and broadcast it to other nodes in N_{bc}.

4.3 PoW-based Consensus Algorithm

When a node receives a mined block $b = \langle H, T \rangle$, it verifies it with the steps shown in Algorithm 4. It takes in an argument *prevHashes*, which is the hash of the header of the node's last block in their blockchain. Nodes could have more than one last block due to *forks*, as we will see later.

In *verifyBlock(b, prevHashes)*, first the difficulty value is checked by double hashing the block header and then, all the transactions are verified with $checkTxn(t)$. A Merkle tree is constructed from these transactions as discussed earlier and it's root matched with MR in the block. Finally it checks if h in block's header points to any of the node's last blocks. If any of the checks fail, the block is discarded.

Algorithm 4 verifyBlock

```
1:  function VERIFYBLOCK(b, prevHashes)
2:      H, T ← b.items()
3:      if hash(hash(H)) > δ then                          ▷ Verifying nonce
4:          return False
5:      end if
6:      for all t ∈ T do                                   ▷ Verifying transactions
7:          if checkTxn(t) is False then
8:              return False
9:          end if
10:     end for
11:     if merkleRoot(T) ≠ H.MR then                       ▷ Verifying Merkle root
12:         return False
13:     end if
14:     if H.h ∉ prevHashes then                           ▷ Verifying h of b
15:         return False
16:     end if
17:     return True
18: end function
```

After verifying it, the node adds b to their B_{bc} and also propagates b to neighboring nodes until consensus is reached in the Bitcoin network. That is, everyone in the network will have the same ordered set of blocks in their B_{bc}.

Let's say a node has a blockchain state, $B_{bc} = \{b_1, \ldots, b_{n-1}\}$, and it receives two valid blocks at the same time, both of whose h values match the hash of b_{n-1}'s head. After verifying, both will be appended to B_{bc}, creating a *fork* [20], which needs to be resolved for consensus.

To resolve this, the node will wait for the next block, which will point to either b_n or b'_n, and thus extending one of the two chains. The node then calculates the sum difficulty of all the blocks in each chain and discards the chain with the lower sum, keeping the longest chain. This could be illustrated in Figure 4.3.

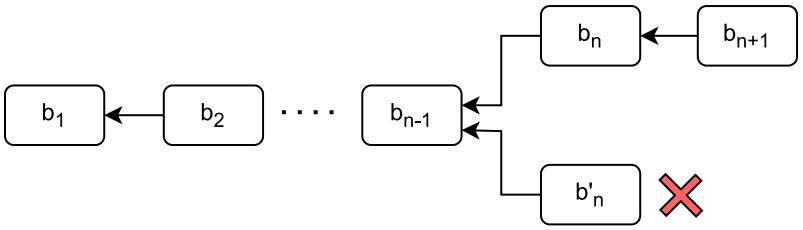

Figure 4.3: Fork resolution in Bitcoin

The above steps of resolving a fork by picking the longest chain is done using the function *resolveFork(prevHashes)*. The transactions that got omitted in the fork resolve and are not in the main chain, are put back in the mempool. Algorithm 5 shows the pseudocode of this function.

Algorithm 5 resolveFork

```
1:  function RESOLVEFORK(prevHashes)
2:      tails ← BlockHash(prevHashes)
3:      maxDiff ← 0
4:      longestChain ← {}
5:      for all tail ∈ tails do
6:          b ← tail
7:          chain ← {b}
8:          diffSum ← b.H.δ
9:          while b.H.h ≠ 0 do                ▷ Looping until genesis block
10:             b ← BlockHash(b.H.h)
11:             diffSum = diffSum + b.H.δ
12:             chain = {b} ∪ chain
13:         end while
14:         if diffSum > maxDiff then
15:             longestChain = chain
16:             maxDiff = diffSum
17:         end if
18:     end for
19:     orphanBlocks ← B_bc − longestChain
20:     orphanTxns ← ⋃{T_p | b_p ∈ orphanBlocks}
21:     T ← ⋃{T_p | b_p ∈ longestChain}
22:     mempool = orphanTxns − T          ▷ Orphan transactions goes to mempool
23:     B_bc = longestChain
24:     return B_bc
25: end function
```

A miner node does the same fork resolving steps but before creating a new block. It executes *resolveFork(prevHashes)* before *createBlock* to ensure the new block always extends the chain with the most accumulated Proof of Work [20]. When every node agrees on the same set B_{bc} and there are no forks, **distributed consensus** is achieved.

Chapter 5

Proposed Ethereum Blockchain Model

First, we will outline the model of the older version of Ethereum that used PoW for the consensus mechanism. We start with a collection of Blockchain systems and let $bc \in BC$ denote a single particular Ethereum blockchain system. Ethereum works in a similar way to Bitcoin with some differences and then extends it further to smart contracts and decentralized applications (dApps) on the EVM (Ethereum Virtual Machine) [18]. Smart contracts are self-executing programs or pieces of code, written in high level language like Solidity and compiled in EVM bytecode, stored on the network, that can get executed by transactions.

5.1 Ethereum 1.0 with PoW

A key difference between Ethereum and Bitcoin is that Ethereum is account-based, meaning entities have accounts with stated balances, whereas Bitcoin calculates balances from all the UTXOs associated with the entity.

5.1.1 Entity Accounts

Similar to nodes/entities in the Bitcoin blockchain model, Ethereum too assigns addresses to all accounts in the network, but not all accounts have a private key. There are two types of accounts in Ethereum:

- **Externally Owned Account (EOA)**: These accounts can initiate transactions at any time and are controlled by the entity's private key [48]. Similar to Bitcoin, Ethereum uses ECDSA to generate the public key from the private key, and the account's address is derived from the last 20 bytes after hashing the public key.
- **Contract Accounts**: These accounts are created when EOAs create smart contracts deployed through transactions [48]. Hence, they do not involve a private-public key pair. The address of the account is generated by hashing the sender's address and nonce during contract creation.

Changes in these accounts are stored in the network as account states, defined as follows:

$$\sigma_k = \langle n_k,\ balance_k,\ codeHash_k,\ storageRoot_k \rangle \tag{5.1}$$

where n is ***nonce*** which is the number of transactions or contracts created by the entity k, and ***balance*** is the amount of Ether (Ethereum coin) owned by the account. For EOAs, ***codeHash*** and ***storageRoot*** are left as empty strings since they are not needed.

For contract accounts, the account contents, including input and output data of the smart contract, are kept in the contract storage trie (discussed in section 5.1.6), and the root of this trie is stored in *storageRoot*. The *codeHash* field in contract accounts has the hash of the smart contract bytecode, used to identify and verify the smart contract on the blockchain [48].

All the account states in the network at a given time, combined, is referred to as the **current state** of the network. We can represent this state as Σ in the following way:

$$\Sigma = \{(ad_1 : \sigma_1), (ad_2 : \sigma_2), \dots\}$$

where ad_i is the address of the account. Later we will see this represented in a Trie structure.

5.1.2 Transactions

Ethereum can be viewed as a transaction-based state machine, where each transaction executed causes state $\sum$ transitions. These transactions are instruction carriers initiated and signed by EOA accounts, either to transfer Ethereum coins or create/execute a smart contract. To execute them on the EVM, computation power referred to as *gas* will be used, which needs to be paid by the transaction initiator/sender [34].

Unlike Bitcoin, it doesn't have the inputs or outputs array in a single transaction. Instead, we can define an Ethereum transaction in the following way and the terms [36] used explained in Table 5.1:

$$t_i = \langle n_i,\ gasLim_i,\ gasPrice_i,\ to_i,\ val_i,\ sig_i,\ data_i \rangle \tag{5.2}$$

There are three types of transactions based on its task:

- **Regular transactions**: Sending coins from one EOA account to another.
- **Contract deployment transactions**: Sent by the contract initiator with the code to deploy it as a smart contract. The ***data*** field contains the contract code and ***to*** address is kept empty. If successful, it creates a contract account.
- **Contract executing transactions**: Sent addressing a contract account to execute/call function(s) in that smart contract. The ***to*** field contains the address of the contract account.

Expression	Terminology
n	Nonce of the sender's account
$gasLim$	Maximum gas the sender is willing to spend
$gasPrice$	Amount in wei (10^{-18} ETH) per unit of gas
to	Address of recipient
val	Value in wei being transferred
sig	Has sender's signature components
$data$	Optional field that could have code to create a contract or call a function

Table 5.1: Ethereum transaction component terminologies

5.1.3 Signing a Transaction

To initiate a transaction, first the sender entity, s, using their EOA, creates the transaction tuple t^* with all the fields as mentioned in equation 5.2 except sig. The fields are then hashed in order and passed through the ECDSA (Elliptic Curve Digital Signature Algorithm) with the sender's private key, to generate sig.

$$ECDSA(key_s^{-1}, \text{hash}(t^*)) \rightarrow (r, s) \tag{5.3}$$

A extra component v, called the recovery identifier is included in the Ethereum signature to help recover the sender's public key, key_s from sig later during verification. [25]

$$sig = \langle v, r, s \rangle$$

The completed transaction is now broadcast and propagated to other nodes in the network.

Verification steps of the signature by other nodes:

$$key_s \leftarrow getPubKeyFromSig(sig)$$

$$ECDSA(sig, key_s, \text{hash}(t^*)) \rightarrow \text{True/False} \tag{5.4}$$

5.1.4 Transaction receipts and logs

Transactions added to a block are executed on the current world state and the outcome about what happened during execution, how much gas was spent, etc, are recorded in transaction receipts [52]. We can define a receipt in the following way:

$$\begin{aligned} tr_i \triangleq \langle & blockHash_i, blockIdx_i, txnHash_i, txnIdx_i, from_i, to_i, \\ & cmlGasUsed_i, gasUsed_i, contrAdr_i, status_i, Logs_i \rangle \end{aligned} \tag{5.5}$$

Here,

- $blockHash_i, blockIdx_i$: hash and index of the block containing the transaction.

- $txnHash_i, txnIdx_i$: hash and index (in the block) of the transaction.
- $from_i, to_i$: address of the sender and recipient of the transaction respectively. to_i is *null* if it was from a contract deployment transaction.
- $gasUsed_i$: amount of gas used by the transaction during execution.
- $cmlGasUsed_i$: cumulative gas used until this transaction in the block, to help nodes keep track of gas usage across multiple transactions for consistency checks.
- $contrAdr_i$: address of the contract account if the transaction created a new contract.
- $status_i$: 1 if the execution was a success, 0 otherwise.
- $Logs_i$: a list of event log tuples emitted from the smart contract if it was a contract executing transaction, or null otherwise.

$$Logs_i = \{log_{i_j} | j = 1, 2, ..., n\}$$

We can define a single log to have the following structure:

$$\begin{aligned} log_{i_j} \triangleq \langle & blockHash_i, blockIdx_i, txnHash_i, txnIdx_i, \\ & contrAdr_i, logIdx_{i_j}, topics_{i_j}, data_{i_j} \rangle \end{aligned} \tag{5.6}$$

Here,

- $logIdx_{i_j}$: index of the log within the block
- $topics_{i_j}$: a list of indexed event parameters in the log designed for efficient filtering and searching of specific events [35]. The values are hashed.
- $data_{i_j}$: log data containing unindexed event parameters, typically larger values or those that don't need to be searchable [35]. The values are stored without hashing.

An example of indexed and unindexed event parameters is discussed later in section 5.2, under the life-cycle of smart contracts.

5.1.5 Blocks

Ethereum 1.0 uses PoW to verify their blocks in a similar way to Bitcoin with some variation. Structure of a single Ethereum block:

$$b_i \triangleq \langle H_i, T_i, O_i \rangle$$

$$\begin{aligned} H_i \triangleq \langle & h_i, oh_i, benef_i, SR_i, RR_i, TR_i, logsBloom_i, \delta_i, n_i, \\ & gasLim_i, gasUsed_i, ts_i, mh_i, \eta_i \rangle \end{aligned} \tag{5.7}$$

$$T_i \triangleq \{t_{i_j} | j \in \{1, 2, ..., n\}\}$$

$$O_i \triangleq \{o_{i_j} | j \in \{1, 2, ..., m\}\}$$

Expression	Terminology
h_i	hash of previous block b_{i-1}
oh_i	hash(O_i)
$benef_i$	Address of this block's miner to send the rewards to
SR_i	Root hash of World state trie
RR_i	Root hash of Receipts trie
TR_i	Root hash of Transaction trie
$logsBloom_i$	Bloom filter to quickly verify event logs of the transactions in T_i
δ_i	Difficulty of mining this block b_i
n_i	Current block index, i
$gasLim$	Total gas limit of all the $t \in T_i$ in the block
$gasUsed_i$	Total gas used by the transactions in T_i
ts_i	Timestamp of when this block was created
mh_i, η_i	mixHash and nonce used for PoW
O_i	List of Ommer blocks (valid blocks that were not included in the main chain)

Table 5.2: Ethereum 1.0 block component terminologies

5.1.6 Trie Data Structures and Bloom filter

In the block structure, we have seen Ethereum using root hash of trie data structures for easier referencing/verification and retrieval of data stored in the network. In Bitcoin we have seen Merkle tree being used to verify transactions stored in a block but that tree cannot be used to retrieve data. For this, Ethereum uses Modified Merkle Patricia Trie (MPT) for all its tree data structures, that uses common prefix in keys to locate the data [58]. We can show a simplified version of the MPT in an example diagram in Figure 5.1 using 3 key-value pairs:

$$\Sigma = \{(\text{b732e5} : \sigma_1), (\text{b79f24} : \sigma_2), (\text{b73281} : \sigma_3)\}$$

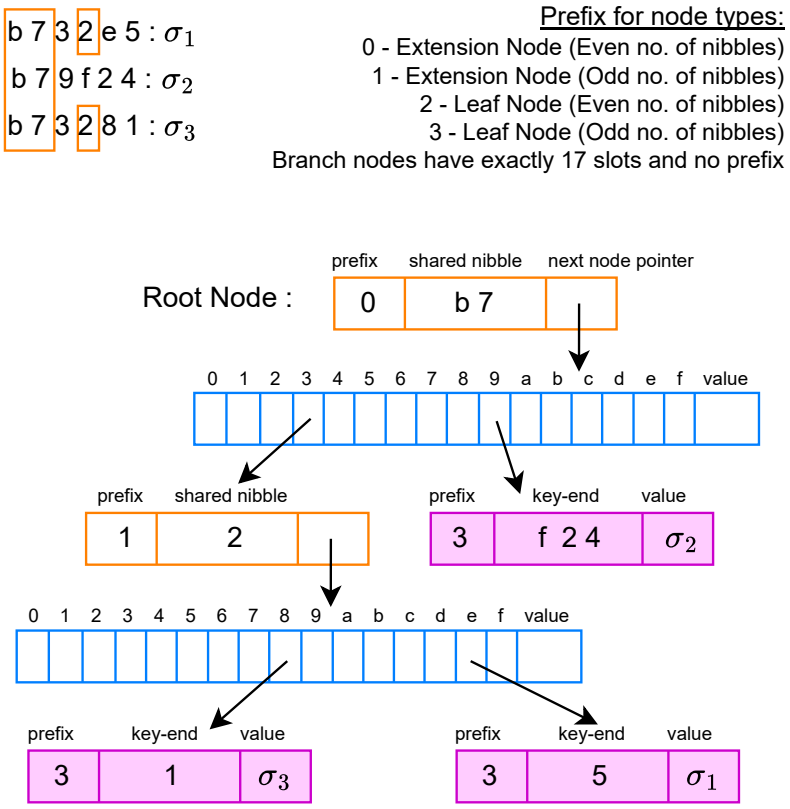

Figure 5.1: Simplified structure of a MPT used in Ethereum

The 4 tries [27] used in Ethereum are:

- **World State trie:** This trie holds the state of all accounts in the network and is mapped by their addresses. To retrieve data of an account state, we use their address and go down from the State Root until we reach a leaf node containing the account state data.

$$getAccState : ad \times SR \rightarrow \sigma \tag{5.8}$$

- **Transaction trie:** The list of transactions in a block are kept in this trie for easier verification. Even though a trie structure for block transactions might seem unnecessary, it's for uniformity without having to introduce another tree data structure. For simplicity, our model's transaction trie uses the transaction index as key and transaction data as the value.
- **Receipt trie:** Similar to transaction trie, this trie has transaction receipts created by the EVM after executing the transactions in the block. Like transaction trie, this trie is also used to validate the block. Receipt index is the key and its data is the value.
- **Contract storage trie:** This trie has all the internal data of the Contracts accounts and its root node is included in *storageRoot* in the account state as mentioned before. The data is stored in the EVM's database (locally on the

user's machine) with their own memory addresses. This address is the key and the data itself is the value, for the trie.

Another data structure used for verification other than Tries, is the **Bloom filter**. It is a probabilistic data structure that Ethereum uses to efficiently store, search and verify the existence of event logs emitted by smart contracts in a block during execution [7]. Using logs in this way instead of storing in contract storage variables, significantly reduces gas costs. The log data are processed using multiple fast hash functions to map the data to specific indexes with allowed overlaps in the Bloom filter, setting those positions to 1. Figure 5.2 is a simple example of a bloom filter with 2 strings.

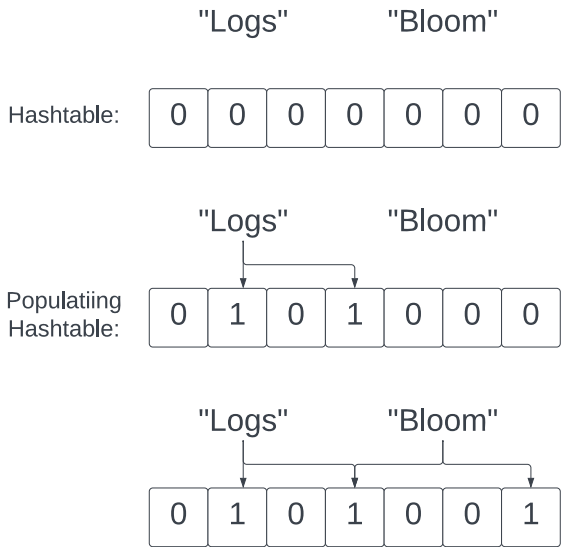

Figure 5.2: Bloom Filter data structure

To verify existence of a specific log, its hash is recalculated: hash($log_{i,j}$)
and the relevant indexes are checked in the Bloom filter. If all bits at these indexes are 1, the log likely exists (with a small chance of a false positive). If any bit is 0, the log definitely does not exist, ensuring no false negatives. This method reduces storage costs and improves verification efficiency.

5.1.7 Validating a Transaction

Similar to the Bitcoin model, Ethereum miner nodes collect transactions and verifies them before execution. First, a temporary copy of the transaction tuple is created with its signature component (*sig*) set to *null*, and the tuple is hashed. According to section 5.1.3, the sender's public key is reconstructed via the function *getPub-KeyFromSig(t.sig)* to be used to validate the signature using ECDSA. From the public key, the corresponding address is derived using *getAddress(*key_s*)*, which is then used to retrieve the sender's account state. The nonce in the transaction and account state is compared to ensure transaction order. If any of these checks fail, the transaction is deemed invalid.

Additionally, a temporary copy of the world state is created, and the transaction is executed on it using the function *executeTxn(t, tempState)*, which will output the gas used and a status indicator. If status is 0 (and not 1), the transaction

was unsuccessful, but will still get added to the block to discourage nodes from sending faulty transactions. Finally, the sender's balance is checked to ensure it can cover the total transaction fee, which includes the gas cost and transaction value. If the balance is insufficient or the gas limit (*gasLim*) is exceeded, the transaction is rejected. We can summarize these steps in Algorithm 6.

Algorithm 6 Ethereum checkTxn

```
1:  function CHECKTXN(t)
2:      t* ← t
3:      t*.sig = null
4:      key_s ← getPubKeyFromSig(t.sig)
5:      σ_s ← getAccState(getAddress(key_s))
6:      if t.sig is null then
7:          return False
8:      end if
9:      if ECDSA(t.sig, key_s, hash(t*)) is False then
10:         return False
11:     end if
12:     if t.n ≠ σ_s.n then
13:         return False
14:     end if
15:     tempState ← copyState(Σ)
16:     gasUsed, status ← executeTxn(t, tempState)        ▷ Simulate t execution
17:     if gasUsed > t.gasLim then
18:         return False
19:     end if
20:     if (gasUsed × t.gasPrice) + t.val > σ_s.balance then
21:         return False
22:     end if
23:     return True
24: end function
```

5.1.8 Block Generation

A miner entity picks verified transactions from their mempool, prioritizing those with higher *gasPrice* for higher reward, to create the transaction list, T_i for the new block. These transactions are executed on the miner's current world state, which transitions to the new world state. We can summarize how the Ethereum state transitions in the state machine diagram in Figure 5.3.

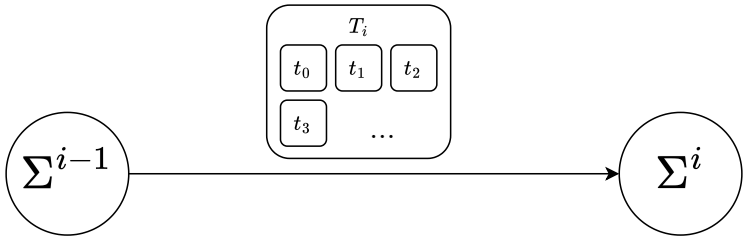

Figure 5.3: State transition summary due to block transactions

During transaction execution, receipts and logs are generated by the EVM. Using these outputs, the transaction trie, receipts trie, and logs bloom filter are constructed as described in section 5.1.6 and their root node hash stored in the block. The world state trie is also updated. The remaining block elements are calculated or generated following the definitions in Table 5.2.

In Ethereum, a block is mined approximately every 12 seconds, resulting in frequent forks during mining. These rejected but valid blocks from the fork resolution that were created at almost the same time are referred to as **Ommer blocks** (uncle blocks). The miner includes these ommer blocks in O_i for the new block, rewarding the miners of these blocks for their valid work which they were not able to propagate in time, thus incentivizating honest mining.

The candidate block constructed so far has ***mixHash*** (mh_i) and ***nonce*** (η_i) initially set to 0 to begin the PoW process using the ***Ethash*** mining algorithm [47]. The block's header is hashed along with a dataset generated from the current blockchain state (referred to as DAG or Directed Acyclic Graph). The calculated hash is placed in mh_i and compared with the difficulty value δ_i. This hashing process is repeated in a loop while incrementing nonce η_i until $mh_i \leq \delta_i$. Once this condition is met, the block is mined.

5.1.9 PoW-based Consensus Algorithm

The miner then broadcasts the mined block and propagates it to other nodes. These nodes then perform the following checks:

- Verify if previous hash h_i in the block matches the hash of their last block.
- Verify each transaction in T_i using *checkTxn(t)*.
- Execute the transactions on a copy of their current world state and reconstruct the three tries. Compare their roots with the corresponding fields (ST, RT, TT) in the block header.
- Verify logs emitted during execution against *logsBloom*.
- Ensure the total gas used is below the *gasLim* and equal to the *gasUsed*.
- Lastly, verify the Proof of Work by ensuring the δ_i, mh_i and η_i values satisfy the condition $mh_i \leq \delta_i$.

If all checks pass, the nodes transition their world state to the new state and add the block to their blockchain. Fork resolving is done similar to in Bitcoin, building on the longest chain with the most work. When the majority of the network includes the block in their longest chain, consensus is achieved.

5.2 Smart Contracts and the EVM

Smart contracts are pieces of code with functions and data, written in a Turing-complete programming language suited for Ethereum, for example, Solidity. These are deployed by EOAs and executed in Ethereum's Virtual Machine (EVM).

5.2.1 EVM and Gas

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts. Unlike a real machine, it is not connected to the internet or any file system and is restricted only to the Ethereum network it is part of. It runs on every node's machine ensuring synchronization of the *blockchain state* and hence is decentralized. As discussed before, when a transaction is initiated by an EOA, they are charged with a certain amount of gas for the computation.

$$\text{Initial gas price paid for } t_i = gasLim_i \times gasPrice_i \tag{5.9}$$

When the transaction is being executed in the EVM, the gas is gradually depleted with each computation. If it exceeds *gasLim*, all the changes are reverted and the transaction's receipt status is set to unsuccessful. This prevents infinite loops and resource depletion. If it does not exceed *gasLim*, the unused gas is refunded to the sender account through the receipt [46].

5.2.2 Lifecycle of a Smart Contract

We can model the lifecycle of a smart contract with the example code [46] below:

```
pragma solidity ^0.8.26;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Code explanation: The first line specifies the compiler version to be used to convert this high-level programming language to machine language for the EVM to execute. `uint storedData` declares a 256-bit unsigned integer. The function `set(uint x)` puts a value x in that variable to be stored in the EVM database and `get()` retrives that value.

1. **Initiation:** An entity (EOA) writes the above program in Solidity, compiles it into EVM bytecode (machine instructions), and puts it in the ***data*** field of a contract deployment transaction, which is then propagated to other nodes.
2. **Storing in the Blockchain:** If the transaction gets included in a block and executed, a contract account is created for the smart contract, modifying the world state trie and storing its address in the receipt. This address is generated from the sender's address and nonce in the following way:

$$contrAdr_i = \text{hash}(ad_s \oplus \sigma(n)_s) \tag{5.10}$$

The account will have its own persistent storage, that can be accessed via key-value pairs in the storage trie as discussed in section 5.1.6.

3. **Calling a Function:** When an account (EOA or contract) wants to execute the `set()` function to store a value `42`, it initiates a contract-executing transaction, placing the contract address in the ***to*** field and encoding `set(42)` in the ***data*** field. A Contract Application Binary Interface (ABI) [22] may be used for encoding the function selector and parameters.

4. **Execution:** If this transaction is added to a block and executed, the EVM will map `storedData = 42` to a storage slot in the contract's storage trie [46]. The receipt records the computation cost as ***gasUsed***, deducted from the sender's balance. Writing to a new storage slot consumes much more gas than modifying the value in an existing slot.
   If the contract emits events during execution, the EVM records them as ***logs*** whose structure is described in section 5.1.4. Each log contains indexed and unindexed parameters. For instance, in an event `Stored(uint256 oldValue, uint256 newValue)`, `oldValue` is indexed for efficient querying, while `newValue` is stored in the unindexed data field. This helps in tracking specific state changes easily without scanning the entire dataset.

A smart contract can be instructed to self-destruct, freeing up storage, but execution and interactions with it is irreversible, as transactions are permanently recorded on the blockchain.

5.2.3 Decentralized Applications (dApps)

Decentralized applications (dApps) provide an interface between traditional web applications and smart contracts within a blockchain. These software applications operate on the EVM using multiple smart contracts as core logic. To simplify our model, we will not include dApps in our Ethereum network.

5.3 Ethereum as a Finite State Machine (FSM)

We have shown how executing transactions changes the world state in Figure 5.3, but state transitions happen for each transaction execution. Below, we model the Ethereum network as a transaction-influenced Finite State Machine (FSM).

A **Finite State Machine** is a mathematical model of computation that shows how the states of a system change from one state to another (transitions) in response to some inputs. It can be defined by the following five-component tuple:

$$\langle Q, \Sigma, \delta, q_0, F \rangle \tag{5.11}$$

- Q : Finite set of states
- Σ : Finite set of input symbols
- δ : Transition function that maps one state to another (potentially influenced by internal parameters or conditions as we will use later)
- q_0 : Initial state
- F : Set of final (or accepting) states

For our model, we define how the world state (the account states) changes as verified transactions in a block are executed, until reaching the new world state.

$$\begin{aligned} Q &= \{S_0, S_n, S_{P_R}, S_{P_C}, S_{P_E}, S_{S_R}, S_{S_C}, S_{S_E}, S_R, S_F\} \\ \Sigma &= \{t_1, t_2, ..., t_n\} \\ q_0 &= S_0 \\ F &= \{S_F\} \end{aligned} \tag{5.12}$$

States:

- S_0 : Initial state representing the world state before processing any transaction.
- S_n : Transaction selection state where an internal variable i (initiated at 0 at S_0) is incremented by 1 to select the next transaction t_i.
- S_{P_R} : (Processing state for regular transactions, RT). Verify the transaction signature, deduct transaction fee from the sender's balance, and update the receiver's balance.
- S_{P_C} : (Processing state for contract deployment transactions, CDT). Verify the transaction signature, calculate the new contract address, store the bytecode in the newly created contract account, and deduct the gas cost from the sender's balance.
- S_{P_E} : (Processing state for contract execution transactions, CET). Verify the transaction signature, execute the specified smart contract function in the EVM, update the contract's storage, and deduct the gas cost from the sender's balance.
- S_{S_R} : Success state for regular transactions, with updated sender and receiver account balances.
- S_{S_C} : Success state for contract creation, with a new contract account added to the world state and the sender's balance updated.
- S_{S_E} : Success state for contract execution, with contract storage updated and sender/contract balances modified.
- S_R : Failure or revert state when the transaction processing fails (due to out-of-gas or other errors).
- S_F : Final state when all transactions in the block have been processed.

Transitions:

- $\delta(S_0, \epsilon) = S_n$, (Start processing transactions)
- $\delta(S_n, t_i) = S_{P_R}$, if t_i is a RT
  $\delta(S_n, t_i) = S_{P_C}$, if t_i is a CDT
  $\delta(S_n, t_i) = S_{P_E}$, if t_i is a CET
- $\delta(S_{P_R}, t_i) = S_{S_R}$, if transaction t_i execution succeeds
  $\delta(S_{P_R}, t_i) = S_R$, if transaction t_i execution fails
- $\delta(S_{P_C}, t_i) = S_{S_C}$, if transaction t_i execution succeeds
  $\delta(S_{P_C}, t_i) = S_R$, if transaction t_i execution fails

- $\delta(S_{P_E}, t_i) = S_{S_E}$, if transaction t_i execution succeeds
  $\delta(S_{P_E}, t_i) = S_R$, if transaction t_i execution fails
- From S_{S_R}, S_{S_C}, S_{S_E} or S_R with ϵ (null):
  - If $i < n$, i.e., more transactions are left, return to S_n.
  - If $i = n$, i.e., no more transactions are left, go to S_F.

We can represent this in the following FSM diagram, starting from S_0 in Figure 5.4.

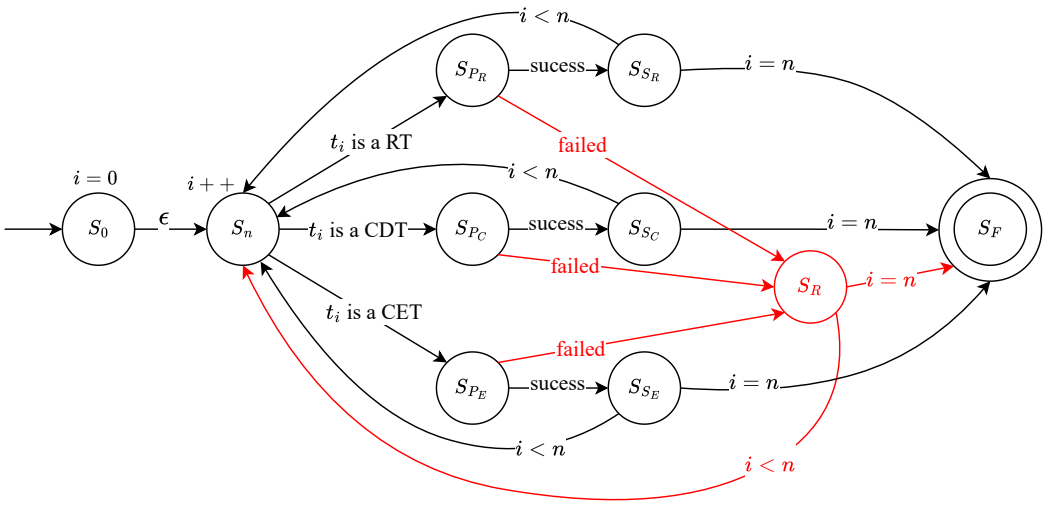

Figure 5.4: FSM of state transitions due to block transactions

5.4 Transition to Ethereum 2.0

Ethereum 1.0, while revolutionary, had several limitations that needed improvement. The Proof of Work (PoW) consensus mechanism, adopted from bitcoin, allowed any nodes to be able to mine and add blocks to the blockchain, which made the network vulnerable to attacks and very power costly. Additionally, it favored centralized mining pools where bigger cooperations with more powerful machines had more advantages of getting reward compared to smaller miners, and could potentially take over the network to approve fraudulent transactions. It also had no penalties for misbehaving miners.

To address these issues, Ethereum fully transitioned to the **Proof of Stake (PoS)** consensus mechanism along with other updates in 2022 in an event called the *Merge*, marking the shift to **Ethereum 2.0**. This upgrade introduced better functionality, decentralization, improved security, and reduced the environmental impact of blockchain operations with a huge decrease in power consumption.

5.4.1 Proof of Stake (PoS)

In this mechanism, instead of every node competing to mine the next block, only selected nodes are allowed to mine and add blocks to the network, reducing the energy consumption by a lot. Instead of calling these nodes *miners*, they are now called **validators**, and instead of *mining*, validators *mint*, *propose* or *forge* blocks.

To become a validator in Ethereum, a node must lock at least 32 ETH from their account balance, as a **stake**, in an *escrow account* through a special smart contract [55]. At every slot, the network choses the next proposer from the list of validators. This selection process is done pseudo-randomly through the RANDAO (Random Number DAO) algorithm. *'Randomly'* because a proposer can't be given any chance to manipulate who gets to be the next proposer, and *'pseudo'* because all nodes should be able to decentrally verify next proposer's ID. This algorithm makes use of a global value called the *randao* which is an accumulation of each proposer adding a bit of randomness in the form of a value, *randaoReveal* that's included in the block they propose [24].

We can model the RANDAO algorithm to find next proposer index from the list of validators, $V = \{v_i \mid i = 1, 2, ..., n\}$ and update *randao* value with the new *randaoReveal* in the steps shown in Algorithm 7.

Algorithm 7 RANDAO Algorithm

function GENERATERANDAOREVEAL($slot$, key_p^{-1}) ▷ proposer, p
  return $BLS\_sign(slot, key_p^{-1})$
end function

function GETRANDAO($slot$)
  $randao \leftarrow 0$
  for each b_i **in** B_{bc} **where** $1 \leq i < slot$ **do**
    $randao = randao \oplus b_i.randaoReveal$
  end for
  return $randao$
end function

function NEXTPROPOSERIDX($V, slot$)
  $randao \leftarrow getRandao(slot)$
  $proposerIdx \leftarrow randao$ **mod** $length(V)$
  return $proposerIdx$
end function

function VERIFYPROPOSER($proposerIdx, slot, V$)
  $randao \leftarrow getRandao(slot)$
  if $(randao$ **mod** $length(V)) \neq proposerIdx$ **then** ▷ Verify proposer idx
    return $False$
  end if
  $pubKey \leftarrow getPubKey(V, proposerIdx)$ ▷ Verify randaoReveal
  $randaoReveal \leftarrow b_{slot}.randaoReveal$
  if $BLS\_verify(pubKey, slot, randaoReveal)$ **is** *False* **then**
    return *False*
  end if
  return *True*
end function

BLS_sign(slot, key_s^{-1}) is the **BLS (Boneh–Lynn–Shacham)** signature generation

algorithm, which is similar in purpose to ECDSA discussed in Section 2.3. But unlike ECDSA, it uses a different elliptic curve and its *pairing property*[1] to aggregate multiple signatures [23]. This makes BLS particularly suitable to generate the *randaoReveal* value by signing the slot number with the proposer's private key.

The Randao value is calculated as the cumulative XOR ($\oplus$) of all the *randaoReveal* values from blocks in the canonical chain, as shown in the *getRandao(slot)* function in Algorithm 7. The next proposer's index in the validator set V is determined by taking the modulus of the Randao value with the size of the validator set.

In the *verifyProposer(proposerIdx, slot, V)* function, the proposer's index is recomputed to ensure it matches the block's proposer, and then the verification function of BLS is used to verify the *randaoReveal* with the proposer's public key.

The definition at Ethereum.org [55] makes use of a weighted probability where validators with higher staked amounts, capped at 32ETH, have more chance of getting picked as the next proposer. We will skip this in our model, but here is how the probability would have looked like:

$$w_i = \min(s_i, 32), \quad \text{where } s_i \text{ is } v_i\text{'s stake in ETH}$$

$$P(v_i) = \frac{w_i}{\sum_{k=1}^{n} w_k}$$

A node can lose their stake through Ethereum 2.0's slashing mechanism if they misbehave (unlike in PoW), which makes a node less likely to commit fraud. The staked money is returned to the owner after a certain amount of time.

5.4.2 Entity Accounts and Transactions

Along with this change from PoW to PoS, some other structural changes were implemented in Ethereum 2.0. The structure of the accounts and transactions stayed the same as described in Section 5.1, except an EOA can now initiate a *deposit transaction* to deploy a special smart contract called a ***Deposit Contract*** to lock the value in the transaction as stake. This transaction called a *deposit*, is a contract executing transaction initiated by the EOA to the fixed address of the deposit contract. The ***val*** field contains the stake of minimum 32 ETH, and the ***data*** field contains the public key, signature of the EOA and deposit amount again, to be passed as parameters to the deposit contract [8]. When the transaction is executed, the deposit contract first verifies the signature with the public key and if successful, the contract emits an event that appends the sender to the list of Validators.

5.4.3 Attestations

In Ethereum 2.0, blocks are indexed by *slot* numbers instead of traditional block indices n_i, based on the sequence. A *slot* represents a single time interval of 12 seconds where a block can be minted, and 32 consecutive slots make up one *epoch*.

In every epoch, validators propose **attestations**, which represent their view of the blockchain. This includes their start block of the current epoch (***source***) and the

[1] Elliptic curve pairing is where 2 points from 2 elliptic curves are mapped to a single number.

current block that's the head of their chain (***target***) i.e. the last justified block. These attestations act as votes to support their perspective, which further strengthens the security and integrity of the Ethereum 2.0 blockchain by solidifying the correct blocks in the main chain. Similar to block proposers, attesters are also rewarded.

To ensure efficiency and scalability, not all validators attest to every block. Instead, the network divides validators into *committees* for each slot and attesters are pseudo-randomly picked from these committees using *randao* [31]. An attestation has the following structure:

$$attestation_j = \langle aggrBits_j, slot_j, attesterIdx_j, source_j, target_j, sig_j \rangle \tag{5.13}$$

- *aggrBits*: A bitmap of validators' indices and whether they agreed with the view (1) or not (0).
- *slot*: Slot number in which this attestation was generated.
- *sig*: Aggregated signature (using BLS) of all participating attesters.

At the start of every epoch, a subset of validators is selected from each committee to become attesters. These attesters generate attestations representing their blockchain view and *aggrBits* containing only their vote. They then propagates it to the other nodes within their committee.

Along with picking attesters, each committee also picks a list of validators to serve as aggregators, who are responsible for collecting the attestations from their committee members that align with their view of the blockchain. The aggregators combine these attestations by updating the *aggrBits* with their own votes and aggregate their signatures using BLS in *sig* field [49]. The aggregated attestations are broadcast to the rest of the network, including other committees, to inform other validators of the consensus state. These steps of generating aggregated attestations is represented in the diagram in Figure 5.5.

Aggregated attestations are included in the next block by the block proposer as a list, as it plays an important role in reaching consensus:

$$Attestations_i = \{attestation_{i_j} \mid j \in \{1, 2, ..., p\}\}$$

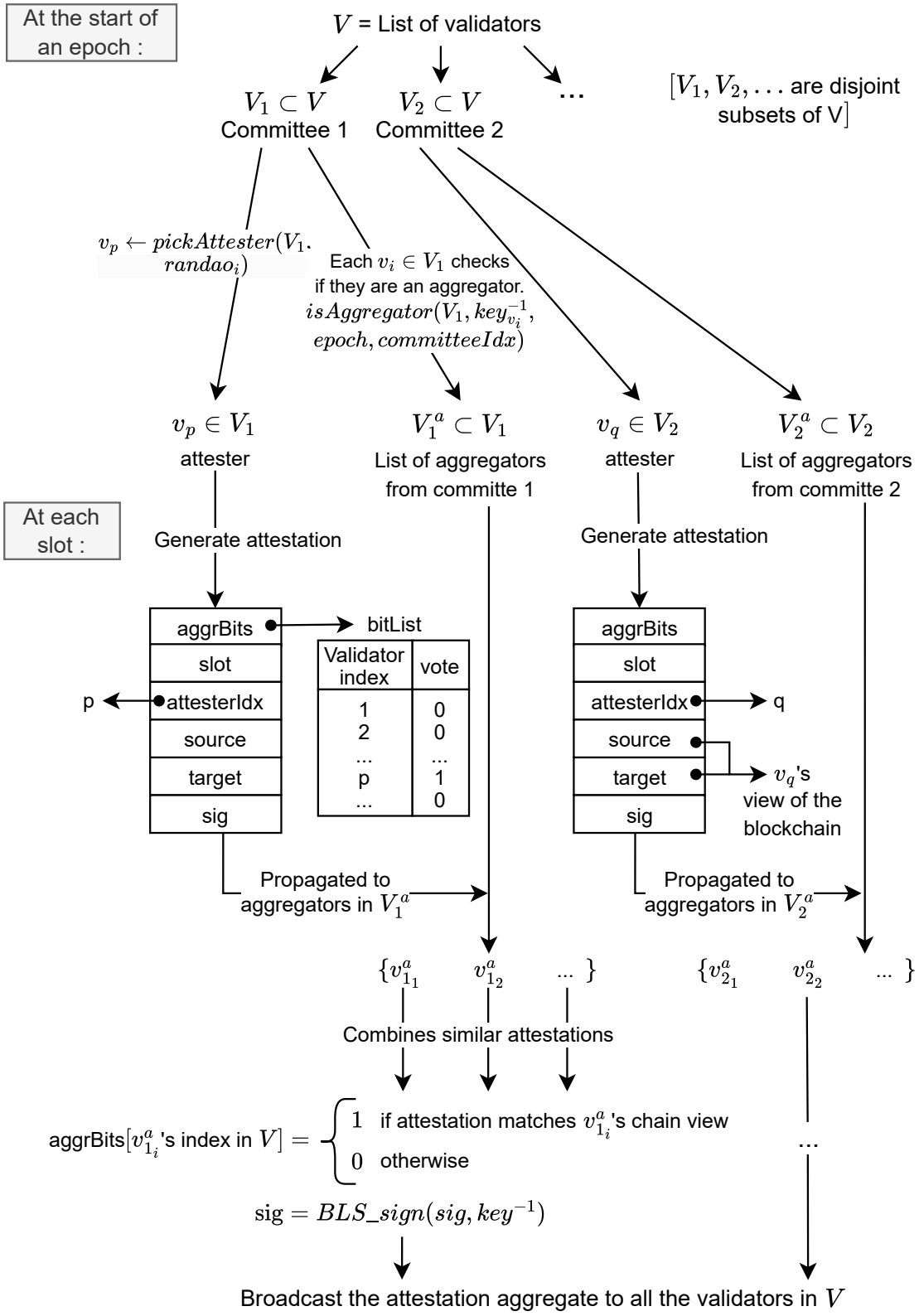

Figure 5.5: Generating aggregated attestation

5.4.4 Withdrawals

Withdrawals in Ethereum 2.0 enable validators to retrieve their earned rewards or staked ETH coins [19]. These withdrawals are either:

- **Partial withdrawals**: Automatically processed when the validator's balance exceeds 32 ETH, to transfer rewards earned to their designated address.
- **Full withdrawals**: Processed when a validator voluntarily exits or is slashed due to misbehaving, releasing their entire balance and thus removing them from the validator list.

Withdrawals have the address of the validator or coin recipient. They are not initiated by transactions but are automatically queued by the block proposer by linearly scanning through the validators list and processed. This also ensures validator activity is finalized and no new misconduct has been detected, before proposing the new block. Once ready, the withdrawals are included as records in the block's execution payload. We can represent a withdrawal in the following way:

$$w_{i_j} = \langle ad_{i_j}, val_{i_j}, j, validatorIdx_{i_j} \rangle \tag{5.14}$$

where, ad_{i_j} is the withdrawal address, j is the index of the withdrawal in this slot, val_{i_j} is the withdrawal amount and $validatorIdx_{i_j}$ is the validator index.

The list of all withdrawals in a block b_i is:

$$Withdrawals_i = \{w_{i_j} \mid j \in \{1, 2, ..., m\}\}$$

5.4.5 Blocks

In Ethereum 2.0, the consensus and execution layers are separated within the block structure for modularity and scalability. The execution layer maintains data fields similar to Ethereum 1.0, including components like SR (state trie root) and TR (transaction trie root), while the consensus layer introduces new fields related to validators, attestations, etc, replacing fields like mh and η used in the PoW model. Ommer blocks are no longer included, as only one validator is responsible for proposing a block at a time, significantly reducing chances of forks. This adds to the efficiency improvement by reducing energy consumption and storage requirements. We can remodel the structure of a Ethereum block after the Merge, in the following way (first letter capitalised for list items) and the teminologies used in Table 5.3:

$$b_i \triangleq \langle slot_i, proposerIdx_i, parentHash_i, SR'_i, body_i \rangle \tag{5.15}$$

$$\begin{aligned} body_i \triangleq \langle & randaoReveal_i, eth1Data_i, ProposerSlashs_i, \\ & AttesterSlashs_i, Attestations_i, deposits_i, \\ & VolnExits_i, syncAggr_i, exePayload_i \rangle \end{aligned} \tag{5.16}$$

Expression	Terminology
$slot_i$	slot number of this block
$proposerIdx_i$	Index of this block's minter in the list of all validators
$parentHash_i$	Hash of previous block
SR'_i	Root hash of the State trie before execution of this block's transactions
$randaoReveal_i$	Randomness value that will be added to randao value that is used to select next block's proposer.
$ProposerSlashs_i$	List of validators who proposed conflicting blocks and will get slashed
$AttesterSlashs_i$	list of attesters with conflicting history
$Attestations_i$	List of attestations in favor of the current block, b_{i-1}
$Deposits_i$	List of new deposit transactions
$VolnExits_i$	List of validators who want to voluntarily exit the network
$syncAggr_i$	summary of blockchain activity
$exePayload_i$	Execution layer fields

Table 5.3: Ethereum 2.0 block component terminologies

Sync aggregate (*syncAggr*) aggregates and compresses data from validators in the *sync committee* to provide summarized view of blockchain activity for nodes with minimal storage and computation capacity called *light node*s [30]. In every 256 epochs, the Ethereum network chooses 512 validators to form the sync committee, who are responsible for sending committee signatures for each block they attest to [51]. A *syncAggr* contains the following fields:

- a bitList of validators in the sync committee and their votes for a block.
- an aggregated signature combining all their signatures for later verification.

The current Ethereum block structure has a field named *eth1Data* which contains the merkle root of the deposit contract's logs, the total number of deposits processed, and the block hash of the Ethereum 1.0 block that has these deposit transactions. This was to ensure synchronization between Ethereum 1.0 and 2.0 chain, by validating deposit events before the Merge. We are excluding this from our simplified model of Ethereum 2.0 since our model has fully transitioned from PoW to PoS. Another field that was simplified, is the *Deposits* set which in the Ethereum specification contains *eth1Data* logs.

The State Root, SR'_i represents the World State trie before execution of the transactions in this block b_i for later verification. The new SR_i will be included in the execution payload or *exePayload*. The header of this *exePayload* contains other fields that changed after executing the block transactions [50] and are similar to what we saw in Ethereum 1.0's block header. This can be modeled in the following way:

$$exePayload_i \triangleq \langle exePayloadHeader_i,\ Withdrawals_i, T_i \rangle \tag{5.17}$$

$$\begin{aligned} exePayloadHeader_i \triangleq \langle & h_i, proposerAdr_i, SR_i, RR_i,\ TR_i,\ WR_i, \\ & logsBloom_i, prevRandao_i, n_i, gasLim_i, \\ & gasUsed_i, ts_i, minGasFee_i \rangle \end{aligned} \tag{5.18}$$

$$T_i = \{t_{i_j} | j \in \{1, 2, ..., n\}\}$$
$$h_i = \text{hash}(exePayloadHeader_{i-1})$$

WR_i is the root hash of the trie that has all the withdrawals in this payload. $prevRandao_i$ is the *randao* value before this block's *randaoReveal*. n_i is this block's index. Rest of the fields are similar to Ethereum 1.0 block structure or previously discussed components.

5.4.6 Block Generation

At the beginning of each slot, validators check if they are selected as the proposer for that slot by computing *nextProposerIdx(V , slot)* from the RANDAO algorithm discussed earlier. If the computed index matches their validator index, they become the proposer and proceed to create the next block in the following steps.

Consensus Layer:

1. The proposer selects transactions from their mempool and verifies them to form the ordered set of verified transactions.

   $$T = \{t \in txnSet \mid checkTxn(t) = True\}$$

   The deposit transactions from this set T are included in the *Deposits* set:

   $$Deposits = \{t_i \in T \mid t_i.to \text{ is the deposit contract address}\}$$

2. The proposer aggregates valid attestations, sync committee contributions, and any queued withdrawals.

3. Validator index of validators who were identified to be misbehaving, are put into sets for slashing:

   - *ProposerSlashs*: for proposing conflicting blocks
   - *AttesterSlashs*: for attesting contradictory chains

   Validators who requested for full withdrawals and were not part of *ProposerSlashs*, are added to *VolnExits*.

4. The proposer generates *randaoReveal* for this block using their private key:

   $$randaoReveal \leftarrow generateRandaoReveal(slot, key_p^{-1})$$

Execution Layer:

4. The proposer executes all transactions $t_i \in T$ on their current world state, updating the state trie, while tracking gas usage to make sure it doesn't exceed the *gasLim*.
5. From the receipts and event logs emitted during execution, the proposer constructs the receipts trie, transaction trie, and logs bloom filter.

Finally, the proposer assembles the block with all this information according to the block structure, and broadcasts it to the network. As a reward, $\frac{1}{7}$th of the total duties the proposer included in the block is added to their account balance by generating new ethers [32].

5.4.7 PoS-based Consensus Algorithm

When a node in the network receives the proposed block, it performs the following checks:

- Verify proposer index and *randaoReveal* of the block b using $verifyProposer(b.proposerIdx, b.slot, V)$.
- Verify the block transactions with $checkTxn(t)$ and then execute them on a copy of their world state to reconstruct all the tries, logs bloom, gas used, etc., comparing them with the block header values.

If all checks pass, the node updates its world state and memory with the new information and adds the block to its chain. Otherwise, if the block fails verification, it is rejected, and the misbehaving proposer is reported for slashing by aggregating the evidence to a slashing contract. The reporter also gets a small reward to their account for valid slashing reports. Consensus is reached when a majority of the validators attests to this chain view with the new block.

Chapter 6

Discussion

Thus the thesis has presented mathematical models for Bitcoin and Ethereum, covering both Proof of Work (PoW) and Proof of Stake (PoS) versions, while focusing on their fundamental structures, consensus mechanisms, and cryptographic foundations. We have made use of graphs, table, proofs, algorithms and diagrams to deconstruct the architectures of both systems and simplify their complexities.
For example, a real bitcoin block is a sequence of bits, where each field occupies a specific number of bytes (except the transaction list). The first 4 bytes is the version number of the Bitcoin software being used, the next 32 bytes represent the previous block's header hash, and so on. Figure 6.1 shows an example of a raw Bitcoin block in hexadecimal format.

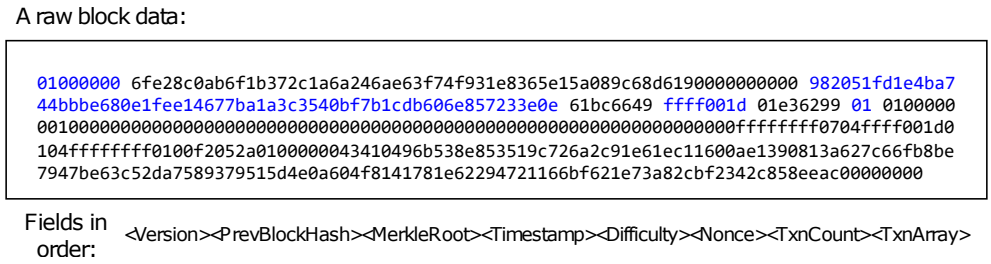

Figure 6.1: Example of a raw Bitcoin block

Comparing this to our bitcoin block structure modeled in Equation 4.5, we can see certain fields missing. This was intentional to simplify the model and exclude fields that are not relevant for the algorithms developed later in this work.

Similarly, structures in Ethereum were covered including the new ones introduced in version 2.0 such as attestations and withdrawals. The Ethereum model chapter extensively compared Ethereum with Bitcoin, highlighting key improvements. The formal representation of Ethereum's state machine behavior using FSM was able to show the state transitions resulting from different types of transactions, including failure states that revert changes. Overall this study was able to cover the research objectives mentioned in Section 1.2 to a significant extent.

6.1 Comparative Analysis

From the literature review in Chapter 3, we observed a lack of comprehensive research papers written on this topic. Even those that appeared to be relevant had several shortcomings.

Many papers interpreted "mathematical modeling" as focusing solely on the cryptographic aspects of blockchain systems, mostly demonstrating concepts like point doubling and ECDSA with graphical representations and formulas. This thesis extends beyond that by incorporating other important mechanisms essential to a functioning blockchain system. It details the steps involved in transaction creation, verification, inclusion in a block through consensus mechanisms, and network-wide synchronization, often with algorithms. This work therefore addresses the major gaps left by previous studies. Additionally, a more detailed FSM representation of Ethereum's state machine was developed.
Ethereum 2.0, one of the most significant innovations in blockchain technology [39], was launched near the end 2020. This explains why prior research papers did not include it. Our paper successfully captures Ethereum 2.0's main concepts, its newly introduced structures, and how they operate.

Despite our attempt to cover the gaps identified in the literature review, some aspects remain unexplored in the main body of the thesis. One such aspect is the use of mathematical models of the blockchains to analyze security of the systems through probability functions, which could be explored in future research.

6.2 Limitations

To keep the model simple, prioritizing conceptual clarity over exhaustive technical detail, several functionalities were skipped. Features such as light nodes, sync committees, and decentralized applications (dApps) were not explicitly covered, as they add complexities beyond the core blockchain mechanisms. Since these elements are mostly associated with Ethereum, its model might not feel as rigorous as Bitcoin's with the missing contexts.
Additionally, while our FSM representation effectively captures Ethereum's state transitions, the internal conditions and parameters used may not fully align with the generalized FSM definition. FSM with internal variables is called the Extended Finite State Machine (EFSM)[53] defined using a 7-tuple, which could make the model difficult to grasp. Some multi-step operations were also simplified using XOR operations for ease of understanding.
These simplifications helped in the modeling process but might have omitted certain technical details needed for a complete understanding of the blockchain representation.

6.3 Future Research Direction

1. **Extending the model to other Blockchain systems:** Following our methodology we can extend our thesis to model other Blockchain platforms like IBM Blockchain and Hyperledger Fabric, and study how different con-

sensus mechanisms and architectures influence blockchain performance. Thus we can combine the models to form a generalized blockchain model that can integrate multiple consensus mechanisms.

2. **Extend to dApps and real-world blockchain use cases:** This thesis does not include the mathematical representation of decentralized applications (dApps) and their interactions with smart contracts to avoid delving deep into their complexities. We could do that in future research and explore popular dApps like OpenSea [21], which manages Non-Fungible Tokens (NFTs), analyzing how they are bought, sold, and stored while ensuring their non-fungibility.

3. **Investigate use of Tree and Tries for verification:** In Ethereum, transactions and receipts in a block are verified using a Merkle Trie which may be an overkill given that no data retrieval is required, as shown in our model. A simple Merkle tree, as used in Bitcoin, could be investigated as a potential alternative. This change or change with some other data structures could be explored whether they improves efficiency without compromising Ethereum's flexibility.

4. **Investigate use of other cryptographic algorithms:** Although ECC provides great security for private-public key pairs, it is vulnerable to attacks from a hypothetical quantum computer [40]. Shor's algorithm for example, can factorize a large integer in polynomial time, making ECC easy to break [56]. Hence, we could explore post-quantum safe asymmetric cryptographic schemes or alternative cryptographic algorithms that could offer better security for Bitcoin and Ethereum.

5. **Analyze probabilistic functions in blockchain security:** As seen in the paper Grunspan and Pérez-Marco [13] in Chapter 3, a statistical analysis of blockchain mechanisms using mathematical models of the systems can provide valuable insights into security concerns. Since that paper covered only Bitcoin, we could similarly extend our study to using the Ethereum model to assess security of its consensus mechanism and explore possible solutions.

The thesis with its two models serves as a foundation for further blockchain research, with a structured mathematical framework. Future researches could build upon these models to optimize blockchain systems, and explore new use cases within decentralized technologies.

Chapter 7

Conclusion

To summarize, the thesis began by introducing **Bitcoin** and **Ethereum**, followed by the mathematical foundations of their cryptographic functions, like hashing, key generation, and and signatures, which were later applied in the models. We then modeled both systems by defining the structures of transactions, receipts (for Ethereum) and blocks using sets and tuples, with structured tables outlining their functions. Algorithms were provided to show some of the field's derivations and key processes. To ensure accuracy, we studied books discussing the respective blockchain systems, their online documentations and relevant articles, to combine and simplify the architectures into coherent models. This approach allowed us to abstract away the low-level byte representations and other complex implementation details.

The Bitcoin model covered its UTXO-based transaction system and PoW consensus, while the Ethereum model covered the account-based system, the EVM and smart contracts with its life-cycle. To enhance clarity, a **Finite State Machine (FSM)** representation of Ethereum's transaction execution was introduced, providing a structured visualization of state transitions. The **Ethereum 2.0** model showed the changes that were made to shift from PoW to PoS, incorporating new structures and processes like attestations and slashing mechanisms to penalize misbehavior. How these updates improved the Ethereum 1.0 model were also analyzed. Additionally, apart from the ECDSA signature scheme, we explored **BLS signatures** used in Ethereum 2.0 for validator signature aggregation, though the underlying mathematics was not covered in depth like we did for ECDSA.

For simplicity and clarity, certain components were omitted or modified. For example, Bitcoin's *scriptPubKey* and *scriptSig* were simplified to highlight ECDSA's role in verification. Despite these changes, all such modifications were explicitly noted to stay true with the original blockchain implementations as much as possible.

By providing an abstract yet functional representation, this thesis bridges the gap between conceptual understanding and real-world implementation. The models allow researchers and engineers to analyze blockchain structures without needing to navigate complex definitions, architectures or code. Additionally, this simplified approach makes the blockchain structures more accessible to students and researchers from disciplines like mathematics and economics, allowing them to study these systems without a computer science background and collaborate effectively.

Bibliography

[1] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, bitcoin.org, Oct. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[2] D. Larimer, "Delegated proof-of-stake (dpos)," *Bitshare whitepaper*, 2014.

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Tech. Rep., 2014, Accessed: 2025-01-03. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

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

[5] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International workshop on open problems in network security*, Springer, 2015, pp. 112–125.

[6] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*, 2nd. O'Reilly Media, 2017, ISBN: 978-1491954386.

[7] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018.

[8] B. Edgington, *The deposit contract*, Eth2book.info, 2018. Accessed: Mar. 21, 2025. [Online]. Available: https://eth2book.info/capella/part2/deposits-withdrawals/contract/.

[9] Hedera, *Proof of stake (pos) vs. proof of work (pow)*, Hedera, 2018. [Online]. Available: https://hedera.com/learning/consensus-algorithms/proof-of-stake-vs-proof-of-work.

[10] S. v. Hoseini, "Mathematics and data structures in blockchain and ethereum," Ph.D. dissertation, Dec. 2018. DOI: 10.13140/RG.2.2.35846.52805/1.

[11] W. Contributors, *History of bitcoin*, Wikipedia, Oct. 2019. [Online]. Available: https://en.wikipedia.org/wiki/History_of_bitcoin.

[12] E. Jiménez-Ayala and J. Medina, "The mathematics behind cryptocurrencies and blockchain," Ph.D. dissertation, 2019. Accessed: Mar. 10, 2025. [Online]. Available: https://atcm.mathandtech.org/EP2019/invited/4382019_21738.pdf.

[13] C. Grunspan and R. Pérez-Marco, "The mathematics of bitcoin," *EMS Newsletter*, vol. 2020-3, pp. 31–37, Mar. 2020. DOI: 10.4171/news/115/8. Accessed: Jan. 7, 2023.

[14] V. Juričić, M. Radošević, and E. Fuzul, "Optimizing the resource consumption of blockchain technology in business systems," *Business Systems Research Journal*, pp. 78–92, 2020.

[15] K. Suresh Kumar, R. Rajeswari, C. Vidyadhari, and B. Santhosh Kumar, "Mathematical modeling approaches for blockchain technology," *IOP Con-*

ference Series: Materials Science and Engineering, vol. 981, p. 022 001, Dec. 2020. DOI: 10.1088/1757-899x/981/2/022001. Accessed: Apr. 20, 2022.

[16] 1. Blockchains, *What is dlt (distributed ledger technology) ?* 101 Blockchains, Jul. 2021. [Online]. Available: https://101blockchains.com/what-is-dlt/.

[17] geeksforgeeks, *Cryptography in blockchain*, GeeksforGeeks, May 2022. [Online]. Available: https://www.geeksforgeeks.org/cryptography-in-blockchain/.

[18] *Intro to ethereum*, ethereum.org, 2022. [Online]. Available: https://ethereum.org/en/developers/docs/intro-to-ethereum/.

[19] B. Edgington, *Withdrawals*, Eth2book.info, 2023. Accessed: Mar. 21, 2025. [Online]. Available: https://eth2book.info/capella/part2/deposits-withdrawals/withdrawal-processing/.

[20] K. Montevirgen, *Crypto forks: What they are and how they work*, https://www.britannica.com/money/crypto-fork-explained, [Accessed 21-03-2025], 2023.

[21] *What Is OpenSea? A 101 Guide*, https://world.org/articles/scroll-through-worldcoin-beginner-guides-to-learn-the/what-is-opensea, [Accessed 20-03-2025], 2023.

[22] A. M. Ajayi, *Understanding smart contract abi, its usage, structure, and evolution*, Medium, Oct. 2024. Accessed: Mar. 21, 2025. [Online]. Available: https://ajayimike.medium.com/understanding-smart-contract-abi-its-usage-structure-and-evolution-08e257b0c32d.

[23] B. Edgington, *Ethereum BLS Signatures - Capella Edition*. Infinite Garden Publishing, 2024, Accessed January 26, 2025. Accessed: Jan. 26, 2025. [Online]. Available: https://eth2book.info/capella/part2/building_blocks/signatures/.

[24] B. Edgington, *Ethereum RANDAO - Capella Edition*. Infinite Garden Publishing, 2024, Accessed January 3, 2025. Accessed: Jan. 3, 2025. [Online]. Available: https://eth2book.info/capella/part2/building_blocks/randomness/.

[25] T. Phung, *Understanding ethereum off-chain signing, ecdsa, eip-712 and its role in permit functionality*, Accessed: 2025-02-12, 2024. [Online]. Available: https://dev.to/truongpx396/understanding-ethereum-ecdsa-eip-712-and-its-role-in-permit-functionality-26ll.

[26] R. A. S, *Merkle Tree in Blockchain: What is it and How does it work*, https://www.simplilearn.com/tutorials/blockchain-tutorial/merkle-tree-in-blockchain, [Accessed 21-03-2025], 2024.

[27] R. Santos, *Understanding ethereum structures: World state trie, transaction trie, receipts, and account storage trie, mpt*, Medium, Jul. 2024. Accessed: Mar. 21, 2025. [Online]. Available: https://medium.com/@ricore77.eth/understanding-ethereum-structures-world-state-trie-transaction-trie-receipts-and-account-d96ab74bb2ac.

[28] T. Wallet, *What is a Mempool in Crypto?* http://trustwallet.com/blog/blockchain/what-is-a-mempool-in-crypto, [Accessed 21-03-2025], 2024.

[29] G. Weston, *Blockchain technology and its impact on the global economy*, 101 Blockchains, Oct. 2024. Accessed: Mar. 5, 2025. [Online]. Available: https://101blockchains.com/blockchain-technology-impact-on-global-economy.

[30] Chainstack, *Sync committee contribution*, Chainstack Developer Portal, 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://docs.chainstack.com/reference/getsynccommitteecontribution.

[31] B. Edgington, *Committees*, Eth2book.info, 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://eth2book.info/capella/part2/building_blocks/committees/.

[32] B. Edgington, *Ethereum 2.0 rewards and incentives*, Ethereum Consensus Layer Documentation, 2025. Accessed: Jan. 17, 2025. [Online]. Available: https://eth2book.info/capella/part2/incentives/rewards/.

[33] E. Foundation. "Ethereum developer documentation." Accessed: 2025-01-26. [Online]. Available: https://ethereum.org/en/developers/docs/.

[34] A. Garnett, *Ethereum gas fees: The cost of doing (crypto) business*, www.britannica.com, 2025. [Online]. Available: https://www.britannica.com/money/ethereum-gas-fees-eth.

[35] S. R. Omanakuttan, *Ethereum logs tutorial series: Logs and filters — chainstack*, Chainstack Developer Portal, 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://docs.chainstack.com/docs/ethereum-logs-tutorial-series-logs-and-filters.

[36] *Understanding the transaction object on ethereum*, Alchemy Docs, 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://docs.alchemy.com/docs/understanding-the-transaction-object-on-ethereum.

[37] G. Walker, *P2PK — Pay To Public Key*, https://learnmeabitcoin.com/technical/script/p2pk/, [Accessed 28-02-2025], 2025.

[38] G. Walker, *ScriptPubKey — The Locking Code in a Bitcoin Transaction*, https://learnmeabitcoin.com/technical/transaction/output/scriptpubkey/, [Accessed 28-02-2025], 2025.

[39] *What is ethereum 2.0 and why does it matter?* Osl.com, 2025. Accessed: Mar. 20, 2025. [Online]. Available: https://osl.com/academy/article/what-is-ethereum-2-0-and-why-does-it-matter.

[40] Wikipedia contributors, *Elliptic-curve cryptography*, Accessed: 2025-02-06, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography.

[41] *Block reward: Definition, how they provide incentive, and future*, Investopedia. Accessed: Jun. 15, 2024. [Online]. Available: https://www.investopedia.com/terms/b/block-reward.asp.

[42] W. Contributers, *Elliptic Curve Digital Signature Algorithm - Wikipedia*, https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm, [Accessed 16-03-2025].

[43] W. Contributers, *Opcode*, https://en.wikipedia.org/wiki/Opcode, [Accessed 27-02-2025].

[44] *Difference Between SHA-256 and Keccak-256*, https://www.geeksforgeeks.org/difference-between-sha-256-and-keccak-256/, [Accessed 10-03-2025].

[45] *Difficulty - Bitcoin Wiki*, https://en.bitcoin.it/wiki/Difficulty, [Accessed 21-03-2025].

[46] S. Documentation, *Introduction to smart contracts*, Accessed January 3, 2025. Accessed: Jan. 3, 2025. [Online]. Available: https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html.

[47] *Ethash - ethereum mining algorithm*, Ethereum Foundation. Accessed: Jan. 25, 2025. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/mining/mining-algorithms/ethash/.

[48] *Ethereum accounts*, Ethereum.org. Accessed: Nov. 10, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/accounts/.

[49] *Ethereum attestations*, Ethereum.org. Accessed: May 26, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attestations/.

[50] *Ethereum blocks*, Ethereum.org. Accessed: May 26, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/blocks/.

[51] *Ethereum sync committee*, Inevitable Ethereum. Accessed: Jan. 3, 2025. [Online]. Available: https://www.inevitableeth.com/home/ethereum/network/consensus/sync-committee.

[52] *Ethereum: Transaction receipt*, Chainstack Documentation. Accessed: Jan. 3, 2025. [Online]. Available: https://docs.chainstack.com/reference/ethereum-gettransactionreceipt.

[53] *Extended finite-state machine - Wikipedia*, https://en.wikipedia.org/wiki/Extended_finite-state_machine, [Accessed 31-03-2025].

[54] Horizen.io, *What is Elliptic Curve Cryptography? - ECC — Horizen Academy*, https://www.horizen.io/academy/elliptic-curve-cryptography-ecc/, [Accessed 16-03-2025].

[55] *Proof of stake: Block proposal*, Ethereum Foundation. Accessed: Jan. 3, 2025. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/block-proposal/.

[56] *Shor's algorithm - wikipedia*. Accessed: Mar. 20, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Shor%27s_algorithm.

[57] P. Sutra Dhor, *The mathematics behind blockchain*. [Online]. Available: https://blockchain.ieee.org/images/files/pdf/techbriefs-2022-q3/the-mathematics-behind-blockchain.pdf.

[58] L. Thomas, *Ethereum block architecture*, Accessed on Ethereum Stack Exchange. Accessed: Jan. 3, 2025. [Online]. Available: https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture.

[59] *What Determines Bitcoin's Price?* https://www.investopedia.com/tech/what-determines-value-1-bitcoin, [Accessed 21-02-2025].

[60] *What is Bitcoin?* https://www.coinbase.com/en-gb/learn/crypto-basics/what-is-bitcoin, [Accessed 21-02-2025].

[61] *What is the bitcoin utxo set?* The Bitcoin Manual. Accessed: May 2, 2023. [Online]. Available: https://thebitcoinmanual.com/articles/btc-utxo-set/.