FACULTY COMPUTER SCIENCE

ULM UNIVERSITY OF APPLIED SCIENCES

Bachelor's Thesis in Computer Science

Blockchain Technology: Consensus Mechanism Proof-of-Work

Author:	Anton Gorshkov
1st Supervisor:	Prof. Dr. rer. nat. Martin Severin
2nd Supervisor:	Prof. Dr.-Ing. Philipp Graf
Submission Date:	30.11.2024

FACULTY COMPUTER SCIENCE

ULM UNIVERSITY OF APPLIED SCIENCES

Bachelor's Thesis in Computer Science

Blockchain Technology: Consensus Mechanism Proof-of-Work

Author:	Anton Gorshkov
1st Supervisor:	Prof. Dr. rer. nat. Martin Severin
2nd Supervisor:	Prof. Dr.-Ing. Philipp Graf
Submission Date:	30.11.2024

Disclaimer

I hereby declare that this thesis is entirely the result of my own work, except where otherwise indicated. I have only used the resources listed in the references.

Ulm, Germany, 30.11.2024 Signature: [illegible] Anton Gorshkov

Acknowledgments

I would like to thank everyone who supported me during the development phase of this project and the thesis as a whole.

- **Prof. Dr. rer. nat. Martin Severin** for providing a challenging and interesting topic and the constant support during the whole process.
- **Prof. Dr.-Ing. Philipp Graf** for the support provided during the elaboration phase of this thesis.

Glossary of Terms

Blockchain: A distributed ledger composed of a sequence of blocks, each cryptographically linked to its predecessor, ensuring immutability and tamper resistance.

Block: A container structure that holds transactions or data, along with metadata such as a timestamp, hash, and reference to the previous block in the chain.

Genesis Block: The first block in the blockchain, with predefined attributes to initiate the chain. In this project, the Genesis Block has a 'bit_difficulty' of 0, allowing any hash to be valid and facilitating quick initialization, even with limited computational resources.

Node: A participant in a blockchain network that maintains a copy of the blockchain and may contribute to consensus processes.

Miner: An entity (individual or organization) that participates in the blockchain network by performing computational work to validate transactions and create new blocks. Miners compete to solve cryptographic puzzles in PoW systems, receiving rewards for successfully mined blocks.

Consensus Mechanism: A protocol used to achieve agreement among distributed nodes in a blockchain network regarding the validity of transactions and blocks.

Proof-of-Work (PoW): A consensus mechanism requiring miners to perform computationally intensive tasks to validate and add blocks to the blockchain.

Proof-of-Stake (PoS): A consensus mechanism in which validators are chosen to add new blocks to the blockchain based on the amount of cryptocurrency they hold and are willing to lock (stake) as collateral.

Transaction: A unit of data recorded in a blockchain, representing an operation such as a transfer of value or information.

Hash: A fixed-length output generated by applying a cryptographic hashing algorithm (e.g., SHA-256) to input data. It uniquely identifies the block's contents.

Timestamp: A time record included in each block, indicating when it was created.

Mining: The process of finding a nonce such that the hash of the block is less than a computed target value.

Nonce: A value incremented by miners during hash computation to find a valid block hash.

Bit Difficulty: Represents the minimum number of leading zeros required for a valid hash in binary representation, allowing fractional adjustments for smoother control. The relationship with the difficulty and probability of finding a nonce is defined as:

$$\text{bit\_difficulty} = -\log_2(\text{probability}) = \log_2(\text{difficulty})$$

Difficulty: A measure of computational effort required to mine a block. It quantifies the inverse probability of finding a valid hash that satisfies the current bit difficulty. The relationship is expressed as:

$$\text{difficulty} = \frac{1}{\text{probability}} = 2^{\text{bit\_difficulty}}$$

Target: The numerical threshold below which a hash must fall to be considered valid for the current 'bit_difficulty'. It is calculated dynamically as part of the difficulty adjustment process:

$$\text{target} = 2^{\text{HASH\_BIT\_LENGTH}-\text{bit\_difficulty}}$$

e.g., HASH_BIT_LENGTH = 256 for SHA-256 (the total number of bits in the hash).

A valid hash H satisfies:

$$H < \text{Target}$$

Probability: The likelihood of a miner finding a valid hash in one attempt, given the current ‘Target‘. It is defined as:

$$\text{probability} = \frac{\text{target}}{2^{\text{HASH\_BIT\_LENGTH}}} = 2^{-\text{bit\_difficulty}}$$

Dynamic Difficulty Adjustment: A mechanism that recalibrates ‘bit_difficulty‘ based on recent mining performance to maintain a stable block production rate. In this project, it leverages fractional adjustments to avoid abrupt changes and ensure consistent mining efficiency.

Clamp Factor: A parameter that restricts the rate of change in ‘bit_difficulty‘ during dynamic adjustments. This ensures mining stability by preventing extreme difficulty fluctuations.

Stochastic Variability: The inherent randomness in mining times due to the probabilistic nature of hash computation. It is influenced by factors like network participants and computational power.

Abstract

Blockchain technology has emerged as a transformative solution for decentralized systems, offering transparency, security, and trust. At its core lies the Proof-of-Work (PoW) consensus mechanism, which ensures secure validation and transaction integrity through computational effort. Despite its strengths, PoW encounters challenges such as high energy consumption, scalability limitations, and centralization risks.

This thesis introduces a blockchain prototype designed to address these challenges by incorporating a floating-point representation of bit difficulty. Unlike integer-based bit difficulty, this approach allows for smoother and more granular adjustments to mining difficulty, enhancing stability in block production rates. The prototype also implements a dynamic difficulty adjustment mechanism to align mining performance with target parameters.

Experimental results demonstrate that the fractional bit difficulty stabilizes mining performance, maintaining block creation times close to the target average despite stochastic variability. The dynamic adjustment mechanism further prevents abrupt shifts in difficulty, ensuring operational consistency. However, the single-miner setup of the prototype highlights the impact of mining time fluctuations and the absence of competition seen in real-world scenarios.

This work bridges theoretical consensus mechanisms with practical blockchain implementations, providing insights into optimizing PoW for scalability and energy efficiency. It lays the foundation for future research into multi-miner environments, realistic transaction modeling, and hybrid consensus mechanisms, contributing to the advancement of blockchain technology.

Zusammenfassung

Die Blockchain-Technologie hat sich als transformative Lösung für dezentrale Systeme etabliert, die Transparenz, Sicherheit und Vertrauen bietet. Im Kern steht der Proof-of-Work (PoW)-Konsensmechanismus, der eine sichere Validierung und Transaktionsintegrität durch Rechenleistung gewährleistet. Trotz seiner Stärken steht PoW vor Herausforderungen wie hohem Energieverbrauch, begrenzter Skalierbarkeit und Zentralisierungsrisiken.

Diese Arbeit stellt einen Blockchain-Prototypen vor, der diese Herausforderungen durch die Einführung einer Gleitkomma-Darstellung der Schwierigkeitsstufe (bit difficulty) adressiert. Im Gegensatz zu ganzzahliger bit difficulty ermöglicht dieser Ansatz eine reibungslosere und granularere Anpassung der Mining-Schwierigkeit, was die Stabilität der Blockerzeugungsrate verbessert. Der Prototyp implementiert außerdem einen dynamischen Mechanismus zur Schwierigkeitsanpassung, um die Mining-Leistung an die Zielparameter anzupassen.

Experimentelle Ergebnisse zeigen, dass die fraktionale bit difficulty die Mining-Leistung stabilisiert und die Blockerstellungszeiten trotz stochastischer Variabilität nahe am Zielmittelwert hält. Der dynamische Anpassungsmechanismus verhindert abrupte Änderungen der Schwierigkeit und gewährleistet betriebliche Konsistenz. Allerdings hebt das Single-Miner-Setup des Prototyps die Auswirkungen von Schwankungen bei den Mining-Zeiten und das Fehlen des Wettbewerbs hervor, wie er in realen Szenarien vorkommt.

Diese Arbeit verbindet theoretische Konsensmechanismen mit praktischen Blockchain-Implementierungen und liefert Erkenntnisse zur Optimierung von PoW in Bezug auf Skalierbarkeit und Energieeffizienz. Sie bildet die Grundlage für zukünftige Forschung zu Multi-Miner-Umgebungen, realistischen Transaktionsmodellen und hybriden Konsensmechanismen und trägt damit zur Weiterentwicklung der Blockchain-Technologie bei.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Blockchain technology has emerged as one of the most disruptive innovations in the fields of finance, supply chain management, and decentralized applications. The decentralized nature of blockchain networks provides increased security, transparency, and trust, without the need for intermediaries [18]. At the heart of this technology lies the concept of consensus mechanisms, which ensure that all nodes in the network agree on the state of the blockchain. Among these mechanisms, Proof-of-Work (PoW) stands as one of the most widely adopted protocols [11].

The importance of PoW cannot be understated, as it is foundational to the security and immutability of major blockchain networks like Bitcoin. It ensures that malicious actors are deterred from manipulating the ledger by making it computationally expensive to alter past transactions. However, as the use of blockchain grows, it is essential to critically analyze PoW's efficiency and explore how it can be improved or replaced to meet the demands of scalability, energy efficiency, and overall performance [19].

1.2 Problem Statement

Despite its popularity and robust security guarantees, Proof-of-Work presents several significant challenges that hinder its scalability and sustainability. The most prominent issues include:

- **Energy Consumption:** PoW demands immense computational power, leading to substantial energy use, which has raised concerns about its environmental impact [6].
- **Scalability:** As more users join a blockchain network, the demand for faster and more efficient consensus increases, but PoW's high computational requirements limit transaction throughput.
- **Mining Centralization:** Over time, mining operations have become increasingly centralized, with large-scale mining farms dominating the landscape. This centralization threatens the decentralization principles of blockchain [4].

These challenges have led to the exploration of alternative consensus mechanisms, such as Proof-of-Stake (PoS), Delegated Proof-of-Stake (DPoS), and Proof-of-Authority (PoA), which aim to reduce energy consumption and improve scalability. However, the transition from PoW to these alternatives introduces its own set of trade-offs, such as decreased security or increased complexity [23].

1.3 Goal of the Project

The goal of this project is to design and analyze a blockchain prototype that implements key features such as block creation, Proof-of-Work consensus, and dynamic difficulty adjustment with floating-point 'bit_difficulty'. This approach enables smoother and more granular adjustments to mining difficulty, enhancing stability.

1.4 Research Questions

This project aims to address the following research questions:

- How does the floating-point representation of 'bit_difficulty' improve blockchain performance and stability?
- What are the trade-offs between simplicity (e.g., single miner) and scalability (e.g., multiple miners) in blockchain prototypes?
- How does dynamic difficulty adjustment influence the stability of block mining time?

1.5 Thesis Structure

This thesis is organized into eight chapters, along with an appendix, as outlined below:

- **Chapter 1: Introduction**
  This chapter provides the background and motivation for the research, articulates the problem statement, defines the goal of the project, identifies key research questions, and presents the structure of the thesis.

- **Chapter 2: Related Work**
  This chapter reviews the state of the art in blockchain technology, consensus mechanisms, and the practical implementation of Proof-of-Work in Bitcoin and Ethereum.

- **Chapter 3: Fundamentals**
  This chapter introduces the foundational concepts necessary for understanding blockchain technology. Topics include cryptographic hashing, blockchain architecture, the concept of floating-point bit difficulty, and the principles of decentralization and security.

- **Chapter 4: Methods**
  This chapter explains the theoretical basis of the research, focusing on the Proof-of-Work algorithm, its process, energy concerns, and a comparison with Proof-of-Stake as an alternative consensus mechanism.

- **Chapter 5: Implementation**
  This chapter details the development of the blockchain prototype. It covers environment setup, implementation of core components such as blocks and blockchain, the Proof-of-Work mechanism, hash utilities, and the dynamic adaptation mechanism. Additionally, it discusses prototype simplifications, the code repository, and UML diagrams.

- **Chapter 6: Results**
  This chapter presents the analysis of the prototype's performance, including mining statistics, the impact of the difficulty adjustment mechanism, and blockchain performance metrics. It also highlights the prototype's limitations and summarizes key conclusions from the results.

- **Chapter 7: Conclusion**
  This chapter summarizes the key achievements and contributions of the thesis, offering insights and lessons learned. It concludes with final remarks on the research outcomes.

- **Chapter 8: Future Work**
  This chapter proposes enhancements to the prototype based on mentioned above simplifications, such as implementing multiple miners, transitioning to continuous mining, incorporating a mining reward system, and using realistic transaction data.

- **Appendix A: Cryptographic Hashing**
  This appendix provides an extended discussion on cryptographic hash functions, including their origins, development, applications, and a visual representation.

Chapter 2

Related Work

2.1 Blockchain Technology

Blockchain technology has been the subject of extensive research over the past decade, with studies examining its potential to revolutionize various industries, including finance, supply chain management, healthcare, and governance. Nakamoto's [18] pioneering work on Bitcoin introduced the concept of a decentralized ledger maintained through cryptographic means, which laid the foundation for blockchain technology as we know it today.

Following this foundational work, research has focused on enhancing the decentralization, security, and scalability of blockchain systems. One of the primary challenges in achieving these goals is known as the Blockchain Trilemma, a concept popularized by Vitalik Buterin [3]. The Blockchain Trilemma refers to the inherent difficulty in simultaneously optimizing three key attributes of blockchain systems: decentralization, security, and scalability.

Decentralization refers to the degree to which control is distributed among network participants.

Scalability relates to the ability of the network to handle an increasing number of transactions efficiently.

Security is a fundamental property that ensures the integrity, authenticity, and immutability of the data stored on the blockchain, preventing malicious actors from compromising the network.

The trilemma implies that optimizing any two of these characteristics often comes at the cost of the third. For instance, Bitcoin's Proof-of-Work consensus provides high levels of security and decentralization, but its scalability is limited, as reflected in the relatively low number of transactions it can process per second.

Figure 2.1 illustrates the trade-offs represented by the Blockchain Trilemma, showing that improvements in one dimension may result in compromises in the others [3].

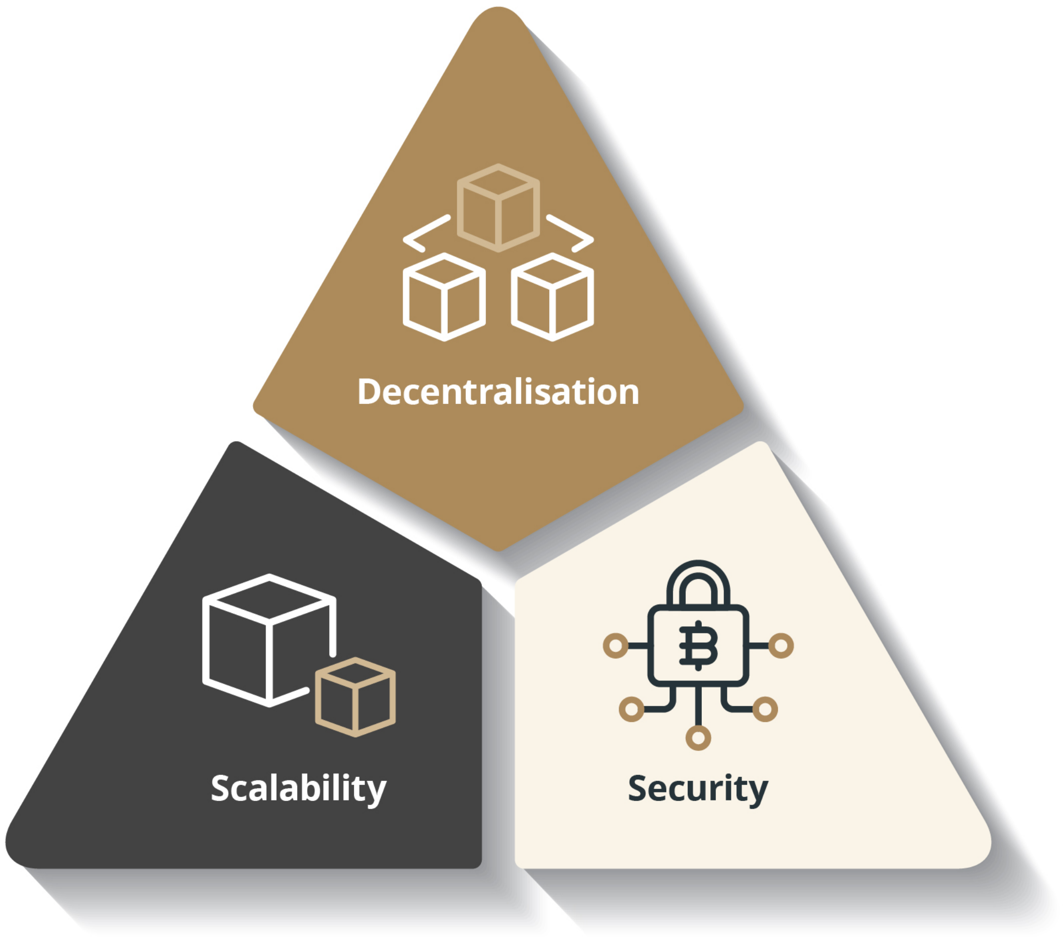


Figure 2.1: The Blockchain Trilemma: Trade-offs Between Decentralization, Security, and Scalability

To address the trilemma, several innovative approaches have emerged.

Solutions such as Layer 2 scaling (e.g., the Lightning Network for Bitcoin [21]) and sharding (used by Ethereum 2.0 [3]) aim to improve scalability while preserving security and decentralization. However, each approach has its own set of limitations and trade-offs, which are still under active research and development.

Other studies have focused on the application of blockchain in specific sectors. In the realm of finance, for instance, studies like [20] examine the potential for blockchain to streamline cross-border payments and reduce the reliance on traditional banking institutions. Research in supply chain management [22] highlights blockchain's ability to enhance transparency, traceability, and trust across global supply chains.

2.2 Consensus Mechanisms

Consensus mechanisms are crucial to the functioning of decentralized blockchain networks, as they ensure all participating nodes agree on the state of the distributed ledger. Proof-of-Work, introduced by Nakamoto [18], has been the predominant consensus mechanism, especially in early blockchain networks such as Bitcoin. PoW relies on computational resources to solve cryptographic puzzles, making it highly secure but also resource-intensive.

Numerous studies have explored the limitations and potential improvements of PoW. For instance, Garay et al. [11] analyzed the security model of PoW in blockchain systems, demonstrating its robustness in a decentralized setting but also highlighting scalability challenges. Research has also focused on reducing the energy consumption of PoW. One notable example is the introduction of hybrid models that combine PoW with less energy-intensive methods [7].

Alternative consensus mechanisms, such as Proof-of-Stake (PoS), have gained attention due to their reduced energy consumption. In PoS, validators are selected based on the number of tokens they hold, rather than computational power. Buterin [4] proposed the Casper protocol as an adaptation of PoS for Ethereum, arguing that it offers a more scalable and sustainable alternative to PoW. Studies like [23] provide a comparative analysis of PoW and PoS, concluding that while PoS addresses some of the energy concerns of PoW, it introduces challenges related to centralization and security.

2.3 Proof-of-Work in Practice

Two of the most prominent case studies of Proof-of-Work in action are Bitcoin and Ethereum, both of which have adopted PoW as their primary consensus mechanism, albeit with differing implementations and long-term goals.

2.3.1 Bitcoin

Bitcoin, the first successful implementation of blockchain, uses PoW as its core consensus mechanism to secure the network. The mining process, wherein participants compete to solve cryptographic puzzles, is integral to the creation of new blocks and the verification of transactions. Studies such as [24] provide a detailed analysis of Bitcoin's PoW mechanism, illustrating its effectiveness in securing the network against attacks but also highlighting the growing problem of energy consumption. Research has consistently shown that Bitcoin's PoW system, while secure, is increasingly energy-intensive, with mining operations consuming more electricity than some entire countries [6].

2.3.2 Ethereum

Ethereum, initially launched with a PoW consensus similar to Bitcoin, sought to extend blockchain's application through the introduction of smart contracts. Unlike Bitcoin, Ethereum's developers have acknowledged the scalability and energy limitations of PoW and are actively transitioning to a Proof-of-Stake consensus through the Ethereum 2.0 upgrade. Buterin's proposal of Casper [4] outlines the shift from PoW to PoS in Ethereum, aiming to make the network more energy-efficient and scalable. The hybrid nature of Ethereum's consensus mechanism during the transition has been a subject of significant study, with researchers examining the trade-offs between security, decentralization, and efficiency [5].

These case studies show that while PoW has proven effective in ensuring network security, its scalability and sustainability are ongoing challenges. The exploration of alternative mechanisms, such as PoS in Ethereum, represents a critical evolution in blockchain technology.

Chapter 3

Fundamentals

3.1 Cryptographic Hashing

Cryptographic hashing is a fundamental concept in blockchain technology that provides data integrity and security. A cryptographic hash function is a mathematical algorithm that takes an input (or ”message”) and returns a fixed-size string of bytes, usually in the form of a hexadecimal value. The output is called the hash value or digest. The unique properties of cryptographic hash functions are crucial for ensuring the security of blockchain systems.

3.1.1 Properties of Cryptographic Hash Functions

Cryptographic hash functions have several important properties that make them suitable for use in blockchain systems [1]:

- **Deterministic**: The same input will always produce the same hash value. This property ensures that any identical data entered into the hash function yields the same output.
- **Fast Computation**: The hash function can quickly produce the hash value for any given input. This is essential for blockchain networks, which need to process numerous transactions rapidly.
- **Pre-image Resistance**: It is computationally infeasible to reverse-engineer the original input from its hash value. This property ensures the security of the data.

- **Small Changes Yield Large Differences (Avalanche Effect)**: A slight modification in the input results in a drastically different hash value. This property makes it easy to detect any tampering with data.
- **Collision Resistance**: It is infeasible to find two different inputs that produce the same hash value. Collision resistance is critical for preventing malicious attempts to create fraudulent data that matches an existing hash.
- **Fixed Output Size**: Regardless of the size of the input, the output hash value has a fixed length. For example, SHA-256 always produces a 256-bit (64 hexadecimal characters) hash.

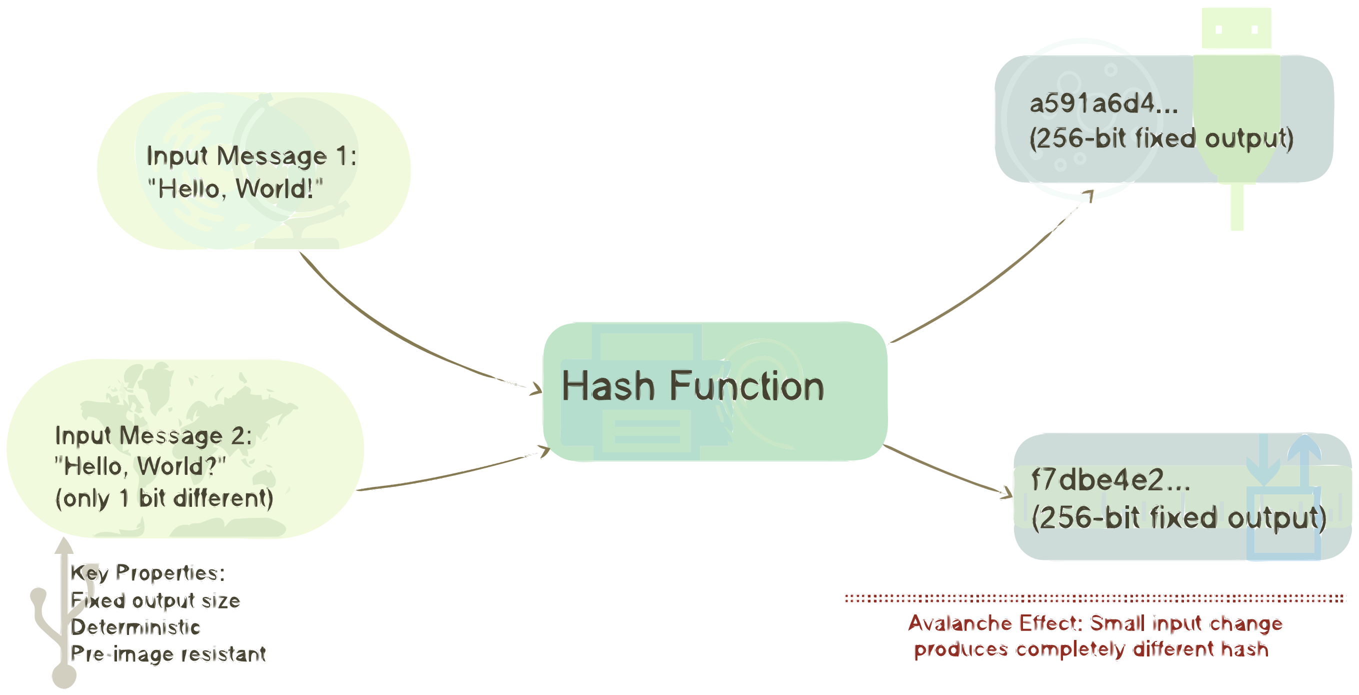


Figure 3.1: A simplified illustration of how a hash function works [9].

Figure 3.1 illustrates the process of applying a hash function to an arbitrary input (e.g., a message or data file). The output is a fixed-size hash value, regardless of the size or complexity of the input. This output is unique to the input and serves as a digital fingerprint for data verification or cryptographic processes.

The diagram also highlights the **avalanche effect**, where even a small change in the input drastically alters the hash output, ensuring the robustness and security of cryptographic hash functions.

3.1.2 Role of Cryptographic Hashing in Blockchain

In blockchain, cryptographic hashing plays several key roles:

- **Data Integrity**: Each block in the blockchain contains a hash of the previous block. This chaining mechanism ensures that altering one block would require recalculating the hashes of all subsequent blocks, making tampering practically impossible.
- **Proof-of-Work (PoW)**: In PoW-based blockchain systems, miners must solve computational puzzles that involve finding a hash value below a given target. This process ensures consensus and prevents unauthorized modifications.
- **Address Generation**: Cryptographic hashes are also used to generate addresses for blockchain wallets, ensuring a level of anonymity for participants.
- **Digital Signatures**: Cryptographic hashing is used in conjunction with public-key cryptography to create digital signatures, which verify the authenticity and integrity of transactions.

3.1.3 Common Cryptographic Hash Functions

Several cryptographic hash functions are used in blockchain technology, each with different properties and use cases:

- **SHA-256**: The *Secure Hash Algorithm 256-bit* is one of the most commonly used hash functions in blockchain, especially in Bitcoin. It generates a 256-bit hash and provides strong security guarantees.

- **RIPEMD-160**: This hash function is used primarily for creating shorter addresses, providing a 160-bit output. Bitcoin uses RIPEMD-160 in combination with SHA-256 to generate public keys and addresses.

- **SHA-3**: A newer member of the Secure Hash Algorithm family, SHA-3 offers enhanced security and is used in some blockchain projects to ensure resistance against more advanced cryptographic attacks.

Cryptographic hashing is a cornerstone of blockchain technology, enabling secure data storage, ensuring immutability, and maintaining consensus. The properties of cryptographic hash functions make blockchain systems resilient to data manipulation, fraud, and unauthorized access. These properties are fundamental to the operation of consensus mechanisms, transaction verification, and maintaining the integrity of the blockchain ledger. More information see in Appendix A

3.2 Blockchain Architecture

Blockchain architecture is a foundational aspect of blockchain technology that determines how the network operates, stores data, and secures transactions. Essentially, a blockchain is a distributed ledger technology (DLT) that maintains an immutable chain of data records, called blocks, which are linked through cryptographic hashes.

3.2.1 Components of Blockchain

A blockchain system is comprised of several key components that ensure the secure and efficient functioning of the network. These components include:

- **Transactions**: A transaction represents the transfer of value or information between participants within the blockchain network. It is the fundamental unit of activity on a blockchain, containing details such as the sender, recipient, and value being transferred. Transactions are validated and grouped into blocks, ensuring consistency and integrity within the distributed ledger.

- **Blocks**: The fundamental unit of a blockchain, each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. The connection between blocks creates the "chain" structure, ensuring that all data is sequentially and securely stored.

- **Nodes**: Nodes are computers that participate in the blockchain network by validating transactions and maintaining copies of the ledger. Nodes can be either *full nodes*, which maintain a complete copy of the blockchain, or *light nodes*, which store only part of the chain. Some nodes can also take on specialized roles, such as *miners* or *validators*. Miners are responsible for solving cryptographic puzzles to add new blocks in Proof-of-Work (PoW) blockchains, while validators participate in the consensus process by staking tokens in Proof-of-Stake (PoS) systems.

- **Consensus Mechanism**: A consensus mechanism is a process used to achieve agreement on a single version of the blockchain among nodes in the network. Examples include Proof-of-Work (PoW) and Proof-of-Stake (PoS), which are used to validate blocks and maintain network integrity.

- **Cryptographic Hash Functions**: These functions are used to secure transactions and link blocks together. A hash function produces a unique, fixed-length output from an input, ensuring data integrity and making it nearly impossible to alter any information without detection. The key properties of cryptographic hash functions—such as pre-image resistance, collision resistance, and the avalanche effect—are discussed in Section 3.1, which play a vital role in maintaining the security and integrity of blockchain systems.

- **Distributed Ledger**: The distributed ledger is a shared, decentralized database that records all transactions that occur on the blockchain. Each node holds a copy of the ledger, ensuring transparency and reducing the risk of a single point of failure [13].

3.2.2 How Blockchain Works

The process begins when a user initiates a transaction. The transaction is broadcast to the network of nodes for validation. Once verified, the transaction is grouped with other transactions to form a block. The block is then added to the existing chain through a consensus mechanism. Each block references the hash of the previous block, creating a secure link and ensuring that no one can alter historical data without affecting all subsequent blocks, thus providing immutability [14].

3.2.3 Consensus Mechanisms

Consensus mechanisms are at the heart of blockchain networks, ensuring that nodes agree on the validity of transactions and the overall state of the ledger. Two commonly used consensus mechanisms are:

- **Proof-of-Work (PoW)**: This mechanism requires nodes to solve complex mathematical puzzles to validate transactions and create new blocks. The process is computationally intensive, which helps to secure the network from attacks.
- **Proof-of-Stake (PoS)**: In PoS, validators are chosen to create new blocks based on the number of tokens they hold and are willing to "stake." This mechanism is more energy-efficient compared to PoW and reduces the need for expensive computational resources [10].

3.2.4 Floating-Point `bit_difficulty`

For simplicity of explanation, understanding, and visually `bit_difficulty` seems to be an integer that represents the minimum number of leading zeros required for a valid block hash. This project extends the concept by introducing a floating-point `bit_difficulty`, allowing for fractional control over difficulty levels.

Mathematical Representation

The relationship between `bit_difficulty`, target, and difficulty is expressed as:

$$\text{Target} = 2^{\text{HASH\_BIT\_LENGTH} - \text{bit\_difficulty}}$$

$$\text{bit\_difficulty} = \log_2(\text{difficulty})$$

A hash H is valid if $H < \text{Target}$.

Advantages

- **Smooth Adjustments:** Enables incremental changes to difficulty.
- **Enhanced Control:** Reduces abrupt changes in mining performance.
- **Improved Realism:** Simulates real-world conditions where difficulty is fine-tuned.

Implementation

This concept is implemented in:

- `adjust_difficulty`: Dynamically recalculates difficulty based on block mining time.
- `validate_hash`: Verifies if a block hash meets the floating-point difficulty target.

3.2.5 Benefits of Blockchain Architecture

Blockchain architecture provides several significant benefits that make it suitable for a wide range of applications, such as finance, healthcare, and supply chain management. Key advantages include:

- **Decentralization**: Blockchain operates on a peer-to-peer network, with no central authority. This ensures that no single entity has control over the data, enhancing security and transparency.
- **Immutability**: Once data is recorded in a blockchain, it cannot be altered without consensus from the majority of nodes, ensuring the integrity and trustworthiness of the information.
- **Security**: Blockchain employs advanced cryptographic techniques to secure transactions and protect against fraud. Hash functions ensure that each block is linked securely to the previous block.
- **Transparency**: The distributed nature of blockchain means that all nodes maintain a copy of the ledger, providing transparency and making it easy for participants to verify transactions.

Blockchain architecture has fundamentally reshaped the way data is stored and shared, providing a secure, transparent, and decentralized solution that has the potential to transform numerous industries [15].

3.3 Decentralization and Security

One of the primary benefits of PoW is its contribution to decentralization and security in blockchain networks. By requiring significant computational power to validate transactions, PoW ensures that no single participant can easily control the network. This decentralization is key to maintaining the security of blockchain systems, as it prevents malicious actors from gaining control over the ledger and manipulating transactions.

In decentralized systems, security is achieved through a combination of cryptographic techniques and consensus protocols. The computational difficulty of PoW serves as a deterrent against attacks, such as double-spending, where a user attempts to spend the same cryptocurrency more than once. Research has shown that blockchain systems employing PoW are highly secure as long as the majority of the network's computational power is distributed among honest participants [23].

The use of PoW also promotes fairness, as it allows anyone with sufficient computational resources to participate in the network. However, over time, mining has become more centralized, with large mining pools dominating the landscape. This trend poses risks to decentralization, as it concentrates power in the hands of a few entities, potentially threatening the security and fairness of the network [6].

Chapter 4

Methods

4.1 Proof-of-Work Algorithm

Proof-of-Work (PoW) is a consensus algorithm used to validate blocks in a blockchain network. In PoW-based systems, such as Bitcoin, nodes (also known as miners) compete to solve complex cryptographic puzzles. The puzzle is computationally expensive to solve but easy for other nodes to verify, a property that is central to the security of the system. Each miner aims to find a nonce value that, when hashed with the block data, results in a hash that meets the network's difficulty requirements [18].

Once a miner finds a valid hash, the block is added to the blockchain, and the miner is rewarded, typically with newly mined cryptocurrency. This process is energy-intensive, requiring significant computational resources to solve the cryptographic puzzle, which leads to concerns about the environmental impact of PoW systems [6]. The difficulty of the PoW puzzle is periodically adjusted to ensure that new blocks are added to the blockchain at a relatively constant rate, regardless of fluctuations in mining power [5].

4.1.1 Process Diagram for Proof-of-Work

Figure 4.1 shows the process flow for PoW. In PoW, miners compete to solve a complex cryptographic puzzle that requires significant computational power. Once a miner successfully solves the puzzle, they broadcast the new block to the network. Other nodes validate the block, and if a consensus is reached, the block is added to the blockchain. Miners are then rewarded with cryptocurrency for their computational effort.

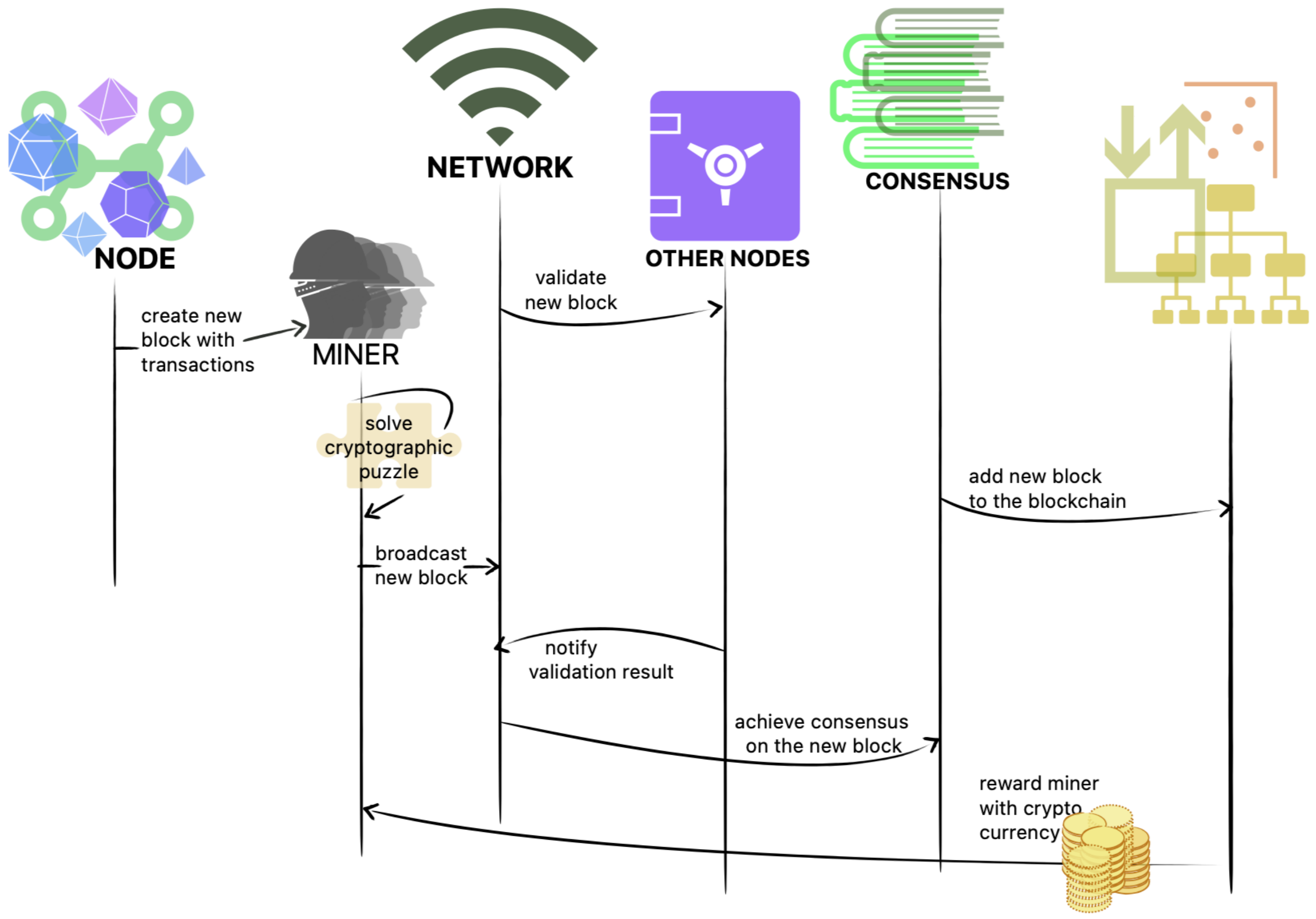

NODE
NETWORK
OTHER NODES
CONSENSUS
MINER
create new block with transactions
solve cryptographic puzzle
validate new block
add new block to the blockchain
broadcast new block
notify validation result
achieve consensus on the new block
reward miner with crypto currency

Figure 4.1: Process Diagram for Proof-of-Work (PoW)

4.1.2 Energy Concerns

One of the most significant criticisms of the PoW consensus mechanism is its high energy consumption. As miners compete to solve cryptographic puzzles, they utilize vast amounts of computational power, leading to high electricity consumption. Studies have shown that Bitcoin mining alone consumes more energy than many small countries [6]. This energy usage has raised concerns about the environmental sustainability of PoW-based systems, especially as global energy demand increases.

The energy cost of PoW is often justified by its role in securing the network; however, this security comes at a substantial environmental cost. The reliance on fossil fuels for electricity generation in many parts of the world exacerbates the issue, contributing to the carbon footprint of PoW-based cryptocurrencies. Some mining operations have attempted to mitigate this by moving to regions with access to renewable energy, but the overall environmental impact remains a significant concern [4].

4.2 Proof-of-Stake as an alternative

Proof-of-Stake (PoS) is an alternative consensus mechanism that has been proposed to address some of the energy concerns associated with Proof-of-Work (PoW). In PoS, instead of relying on computational power, validators are selected based on the number of tokens they hold and are willing to "stake" as collateral. This system reduces the need for energy-intensive computations, making it more environmentally friendly than PoW [23].

PoS also promotes decentralization in different ways compared to PoW. While PoW can lead to the centralization of mining power in regions with cheaper energy or specialized hardware, PoS allows anyone with tokens to participate in the validation process. This lowers the barriers to entry and can help distribute control more evenly across the network [4]. However, PoS systems are not without their challenges, including concerns about security and the potential for wealth centralization, as those with more tokens are more likely to be selected as validators.

While PoS is considered more energy-efficient and potentially more decentralized, its security properties have been the subject of ongoing research, particularly regarding the risk of "nothing at stake" attacks, where validators have little incentive to act honestly if they face no penalties for validating multiple competing chains. Both PoW and PoS have trade-offs in terms of security, decentralization, and energy efficiency, and the choice between them depends on the specific goals and constraints of a given blockchain network [5].

4.2.1 Process Diagram for Proof-of-Stake

Figure 4.2 illustrates the process flow for PoS. In PoS, validators are selected based on the number of tokens they are willing to stake. Once a validator is chosen, they create a new block and broadcast it to the network. Other validators verify the block's validity, and once a consensus is reached, the block is added to the blockchain. Validators are then rewarded or penalized based on their actions during the validation process.

4.3 Proof-of-Work and Proof-of-Stake Comparison

The key differences between PoW and PoS are evident in their respective process diagrams. In PoW, miners must expend computational power to solve cryptographic puzzles, which requires a significant amount of energy. The reward system incentivizes miners to participate, but it also contributes to the centralization of mining in regions with cheaper energy.

In contrast, PoS relies on validators staking their tokens, which eliminates the need for energy-intensive computations. This makes PoS more environmentally friendly and potentially more accessible to a broader group of participants. However, the risk of wealth centralization remains a concern, as those with more tokens have a greater likelihood of becoming validators.

Both consensus mechanisms have their strengths and weaknesses, and the choice between PoW and PoS ultimately depends on the goals and requirements of the blockchain network. While PoW is highly secure due to its reliance on computational power, PoS offers a more sustainable alternative that may be better suited for future blockchain applications that prioritize energy efficiency and broader participation.

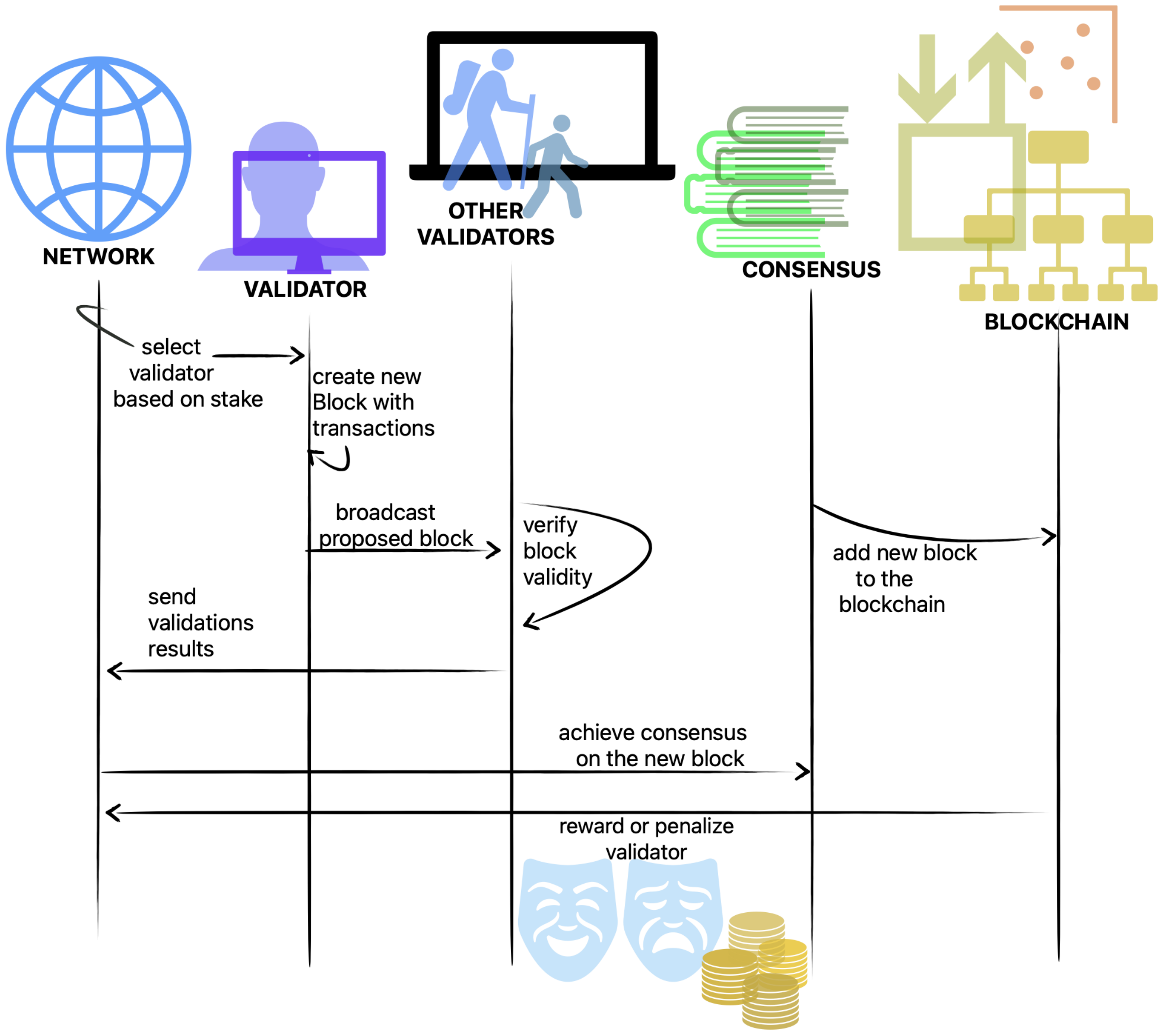

NETWORK
VALIDATOR
OTHER VALIDATORS
CONSENSUS
BLOCKCHAIN
select validator based on stake
create new Block with transactions
broadcast proposed block
verify block validity
add new block to the blockchain
send validations results
achieve consensus on the new block
reward or penalize validator

Figure 4.2: Process Diagram for Proof-of-Stake (PoS)

Chapter 5

Implementation

5.1 Introduction to Blockchain Programming

The goal of this chapter is to implement a blockchain prototype to demonstrate how the core features of a blockchain can be programmed. In the implementation, the prototype will include block validation through Proof-of-Work and basic testing to verify the functionality of the blockchain, measuring its stochastic properties.

This prototype was developed entirely from scratch, without relying on any blockchain-specific libraries, such as `web3.py`, `pyethereum`, or similar frameworks. As a result, the implementation is relatively simple compared to real-world blockchain projects, which often include extensive networking, advanced consensus mechanisms, and optimized performance.

The primary intention behind developing this prototype was to gain a deeper understanding of how core blockchain components work, including block creation, linking blocks, and ensuring data integrity through cryptographic hashes. By implementing these elements manually in Python, this project provides a hands-on exploration of blockchain fundamentals in a straightforward and controlled environment. While this approach lacks the complexity and scalability of production-level blockchains, it offers valuable insights into their underlying principles and mechanisms.

5.2 Environment Setup

In order to set up the environment and run the application, follow the steps of Installation and Usage of the github repository README.md:

```
https://github.com/antoniooreany/Blockchain-PoW-Consensus/
blob/main/README.md
```

To implement the blockchain prototype, the following Python libraries are used:

- **hashlib:** This Python library is used to generate cryptographic hashes for block creation. It supports several hashing algorithms, including SHA-256, which is fundamental to Proof-of-Work-based blockchains like Bitcoin [19].
- **logging:** The logging module in Python is used to record runtime information, such as debug- and info-messages, errors, and warnings, during the program execution. It helps in debugging, monitoring, and understanding the flow of the application.
- **math:** The math library provides mathematical functions such as logarithmic and power calculations, which are essential in dynamic difficulty adjustment mechanisms to control the mining difficulty.
- **random:** The random library is used to generate random numbers, which can be helpful in various scenarios within the blockchain, such as randomly choosing the enter point for the Proof-of-Work nonce calculations.
- **time:** This library is used to record timestamps, which are crucial for tracking when blocks are created, as well as for calculating the time taken for mining for adjusting the difficulty dynamically.
- **matplotlib:** Matplotlib is a popular plotting library in Python used to create visualizations. In this project, it is used to generate graphs showing the mining process, difficulty levels, and performance metrics for better understanding and analysis.

- **numpy:** Numpy is a fundamental library for numerical operations in Python. It is used for efficient array manipulations, statistical calculations, and generating data points during the analysis of blockchain performance.
- **screeninfo:** Screeninfo is a utility that provides information about the available screens and their dimensions. It helps to adapt the visualizations and plots generated by the application to fit optimally on the user's screen.

5.3 Block Implementation

The `Block` class serves as a fundamental component of the blockchain system, encapsulating data and cryptographic attributes that ensure the security, immutability, and orderly linkage of blocks within the chain. Key attributes of the `Block` class include:

- **Index:** Specifies the sequential position of the block in the blockchain, ensuring chronological order.
- **Bit Difficulty:** Denotes the level of mining difficulty associated with the block, influencing the computational effort for Proof-of-Work.
- **Timestamp:** Records the creation time of the block as a Unix timestamp.
- **Data:** Contains the information or transactions stored within the block.
- **Previous Hash:** Stores the hash of the preceding block, creating a cryptographic link that secures the blockchain's integrity.
- **Nonce:** A numerical value that is incremented during the mining process to satisfy Proof-of-Work conditions.
- **Hash:** A unique identifier for the block, generated by hashing its attributes using a cryptographic algorithm.

5.3.1 Block Initialization

The `Block` class initializes these attributes to establish the structural and cryptographic foundations of a blockchain. Each attribute contributes to the block's security, making unauthorized modifications easily detectable.

5.3.2 Hash Calculation

A notable feature of the `Block` class is the `calculate_block_hash` method, which computes the block's hash. The process is as follows:

1. Combines the block's attributes (`index`, `timestamp`, `data`, `previous_hash`, and `nonce`) into a single string.

2. Encodes the string using a specified encoding format, such as UTF-8.

3. Applies the SHA-256 hashing algorithm to the encoded string, generating a secure, fixed-length hash.

The generated hash ensures data integrity and serves as a tamper-proof identifier. Any alteration to the block's attributes results in a completely different hash, signaling potential tampering and safeguarding the blockchain's immutability.

5.3.3 Integration with Blockchain System

The `Block` class integrates seamlessly with other components of the blockchain system, such as the `ProofOfWork` and `Blockchain` classes. The `previous_hash` attribute ensures that each block is cryptographically linked to its predecessor, while the `calculate_block_hash` method supports mining and validation processes.

The structure of the `Block` class is as follows:

```python
#   Copyright (c) 2024, Anton Gorshkov
#   All rights reserved.
#   This code is for a block and its unit tests.
#   For any questions or concerns, please contact Anton
    Gorshkov at antoniooreany@gmail.com

import logging
import random
import time

from src.constants import MAX_NONCE
from src.utils.hash_utils import calculate_block_hash


class Block:
    def __init__(
            self,
            bit_difficulty: float,
            index: int,
            data: str,
            previous_hash: str,
    ) -> None:
        """
        Initialize a new block with its attributes.

        Args:
            bit_difficulty (float): The difficulty level
                of the block.
            index (int): The position of the block in the
                blockchain.
            data (str): The data contained within the
                block.
            previous_hash (str): The hash of the previous
                block in the chain.
        """
        # Set the difficulty level for the block
        self.bit_difficulty: float = bit_difficulty

        # Assign the block index
        self.index: int = index

        # Store the block's data
        self.data: str = data

        # Record the timestamp of block creation
        self.timestamp: float = time.time()  # Here we use
             the current time as the timestamp

        # Store the hash of the previous block
        self.previous_hash: str = previous_hash

        # Initialize the nonce with a random value within
            the possible range
        self.nonce: int = random.randint(0, MAX_NONCE)
        logging.info(f"Block {self.index} initial nonce
            search enter point: {self.nonce}")

        # Compute the hash of the block
        self.hash: str = calculate_block_hash(
            index=self.index,
            timestamp=self.timestamp,
            data=self.data,
            previous_block_hash=self.previous_hash,
            nonce=self.nonce,
        )
```

Listing 5.1: Block Implementation

This implementation reinforces the blockchain's security and immutability by leveraging cryptographic hashing and attribute validation. The `Block` class is indispensable for constructing a robust and tamper-proof blockchain system.

5.4 Blockchain Implementation

The `blockchain.py` module implements the foundational `Blockchain` class, which manages block creation, mining, validation, and dynamic difficulty adjustment to ensure the integrity, scalability, and security of the blockchain system. The class includes the following key methods:

- `__init__`: Initializes the blockchain with critical parameters such as initial difficulty, target mining time, and difficulty adjustment intervals. It creates the genesis block and prepares the blockchain for efficient block management.
- `get_latest_block`: Retrieves the most recent block in the chain, ensuring proper linkage for new blocks.

- `add_block`: Mines and appends a new block to the chain, validates its Proof-of-Work, and adjusts mining difficulty dynamically.
- `get_average_mining_time`: Calculates the average mining time over a specific number of blocks, offering insights into network performance.
- `adjust_difficulty`: Dynamically modifies the difficulty level by analyzing recent mining times, ensuring adaptability to computational power changes.
- `log_difficulty_anomalies`: Detects and logs inconsistencies in difficulty adjustments to maintain mining stability.

5.4.1 Blockchain Initialization

The `__init__` method sets up the blockchain by initializing key parameters such as difficulty and mining time targets. It also generates the genesis block, the first block in the chain, with predefined attributes. The genesis block is logged to ensure transparency and traceability.

5.4.2 Block Management and Mining

The `add_block` method is responsible for securely mining and appending new blocks to the chain. It operates as follows:

1. Sets the `previous_hash` of the new block based on the latest block in the chain.
2. Invokes the `find_nonce` method from the `ProofOfWork` class to mine the block.
3. Updates the block's timestamp upon successful mining.
4. Validates the mined block using the `validate_proof` method.
5. Appends the block to the chain and updates mining statistics.

This method ensures that all blocks meet the Proof-of-Work requirements before being added.

5.4.3 Dynamic Difficulty Adjustment

The `adjust_difficulty` method recalibrates the mining difficulty by analyzing the average mining time of recent blocks. The process includes:

- Calculating the average mining time over a predefined interval.
- Adjusting the difficulty using a clamping mechanism to maintain stability.
- Logging any anomalies detected in the difficulty adjustment process.

This functionality ensures the blockchain adapts to fluctuating computational resources.

5.4.4 Integration with Proof-of-Work

The `Blockchain` class heavily relies on the `ProofOfWork` class for mining operations and validation. Functions like `find_nonce` and `validate_proof` are pivotal for ensuring blocks adhere to the Proof-of-Work criteria.

The structure of the `Blockchain` class is as follows:

```python
#   Copyright (c) 2024, Anton Gorshkov
#   All rights reserved.
#   This code is for a blockchain.py and its unit tests.
#   For any questions or concerns, please contact Anton
    Gorshkov at antoniooreany@gmail.com

# path: src/model/blockchain.py

import logging
import math
import time

from src.model.block import Block
from src.constants import DEFAULT_PRECISION,
    AVERAGE_MINING_TIME_ADJUSTMENT_INTERVAL_KEY, \
     REVERSED_ADJUSTMENT_FACTOR_KEY,
         GENESIS_BLOCK_BIT_DIFFICULTY
from src.utils.logging_utils import configure_logging
from src.utils.logging_utils import log_validity
from src.controller.proof_of_work import ProofOfWork
from src.constants import GENESIS_BLOCK_PREVIOUS_HASH,
    GENESIS_BLOCK_DATA
from src.utils.logging_utils import log_mined_block


class Blockchain:
    def __init__(
            self,
            initial_bit_difficulty: float,
            target_block_mining_time: float,
            adjustment_block_interval: int,
            number_blocks_to_add: int,
            clamp_factor: float,
            smallest_bit_difficulty: float,
            number_blocks_slice: int,
    ) -> None:
        """
        Initialize a new blockchain with the given
            parameters.

        Parameters:
            initial_bit_difficulty (float): The initial
                difficulty level of the blockchain.
            target_block_mining_time (float): The target
                time to mine a block in seconds.
            adjustment_block_interval (int): The number of blocks to wait before adjusting the difficulty.
            number_blocks_to_add (int): The number of blocks to add to the blockchain.
            clamp_factor (float): The factor to clamp the adjustment of the difficulty.
            smallest_bit_difficulty (float): The smallest bit difficulty that we can adjust to.
            number_blocks_slice (int): The number of blocks to slice the list of blocks to calculate the statistics.

        Notes:
            The number of blocks to add is the number of blocks to add to the blockchain after the Genesis Block.
            The number of blocks slice is the number of blocks to slice the list of blocks to calculate the statistics.
        """

        self.logger: logging.Logger = configure_logging()

        self.initial_bit_difficulty: float = initial_bit_difficulty  # The initial difficulty level of the blockchain.
        self.target_block_mining_time: float = target_block_mining_time  # The target time to mine a block in seconds.
        self.adjustment_block_interval: int = adjustment_block_interval  # The number of blocks to wait before adjusting the difficulty.
        self.number_blocks_to_add: int = number_blocks_to_add  # The number of blocks to add to the blockchain.
        self.clamp_factor: float = clamp_factor  # The factor to clamp the adjustment of the difficulty.
        self.smallest_bit_difficulty: float = smallest_bit_difficulty  # The smallest bit difficulty that we can adjust to.
        self.number_blocks_slice: int = number_blocks_slice  # The number of blocks to slice the list of blocks to calculate the
```

```python
statistics.
        self.bit_difficulties: list[float] = [initial_bit_difficulty]  # The list of bit difficulties in the blockchain.
        self.proof_of_work: ProofOfWork = ProofOfWork()  # Create an instance of ProofOfWork

        # Create the Genesis Block
        start_time: float = time.time()
        genesis_block: Block = Block(
            bit_difficulty=GENESIS_BLOCK_BIT_DIFFICULTY,  # todo it might be initial_bit_difficulty
            index=0,
            data=GENESIS_BLOCK_DATA,
            previous_hash=GENESIS_BLOCK_PREVIOUS_HASH,
        )
        self.blocks: list[Block] = [genesis_block]  # The list of blocks in the blockchain.

        log_mined_block(genesis_block)
        log_validity(self)

        self.mining_times: list[float] = [genesis_block.timestamp - start_time]  # avoid the check for the Genesis Block
        # todo ugly, calculate the mining time for the Genesis Block in generic way.

        logging.debug("\n")

    def add_block(self, new_block: Block, clamp_factor: float, smallest_bit_difficulty: float) -> None:
        """
        Add a new block to the blockchain, validate it and update the blockchain state.
        """
        # Record the start time of mining
        mining_start_time = time.time()

        # Set the previous hash of the new block to the hash of the latest block in the blockchain
        new_block.previous_hash = self.get_latest_block().hash if self.blocks else GENESIS_BLOCK_PREVIOUS_HASH

        # Find a nonce for the new block to satisfy proof of work
        self.proof_of_work.find_nonce(new_block, self.bit_difficulties[-1])

        # Update the timestamp to the end of mining
        new_block.timestamp = time.time()
        actual_mining_time = new_block.timestamp - mining_start_time

        # Validate the new block's proof of work
        if not self.proof_of_work.validate_proof(new_block, self.bit_difficulties[-1]):
            self.logger.error(f"Block {new_block.index} was mined with an invalid hash")
            return

        # Add the new block to the blockchain
        self.blocks.append(new_block)

        # Append the actual mining time
        self.mining_times.append(actual_mining_time)

        # Adjust difficulty and log results
        self.bit_difficulties.append(self.bit_difficulties[-1])
        self.adjust_difficulty(clamp_factor, smallest_bit_difficulty)
        self.log_difficulty_anomalies()

        # Log validity and actual mining time
        log_validity(self)
        self.logger.debug(
            f"Actual mining time for block {new_block.index}: {actual_mining_time:.{DEFAULT_PRECISION}f} seconds\n")

    def get_latest_block(self) -> Block | None:
        """
        Retrieve the latest block from the blockchain.

        Returns:
            Block | None: The latest block if available, otherwise None if the blockchain is empty.

        Notes:
            This function provides a way to access the most recent block in the blockchain.
            If the blockchain has no blocks, it returns None.
        """
        # Check if there are any blocks in the blockchain
        if self.blocks:
            # Return the last block in the list, which is the latest block
            return self.blocks[-1]
        # Return None if the blockchain is empty
        return None

    def get_average_mining_time(self, num_last_blocks: int) -> float:
        """
        Calculate the average mining time for the last `num_last_blocks` blocks.

        Args:
            num_last_blocks (int): The number of blocks to calculate the average mining time for.

        Returns:
            float: The average mining time for the last `num_last_blocks` blocks.

        Notes:
            If the number of blocks in the blockchain is less than or equal to 1,
            or if the blockchain has less than `num_last_blocks+1` blocks, then
            the average mining time for all blocks (except the Genesis Block) is returned.
        """
        # If the number of blocks in the blockchain is less than or equal to 1,
        # or if the blockchain has less than `num_last_blocks+1` blocks, then
        # the average mining time for all blocks (except the Genesis Block) is returned
        if len(self.blocks) <= 1:  # in the case of the Genesis Block
            return 0.0  # return 0.0 as the average mining
            time
        if len(self.blocks) < num_last_blocks + 1:  # (+1) to exclude the Genesis Block from the calculation
            return sum(self.mining_times[1:]) / (len(self.mining_times) - 1)
        total_time: float = sum(
            self.mining_times[-num_last_blocks:])  # sum the mining times of the last `num_last_blocks` blocks
        return total_time / num_last_blocks  # calculate the average mining time

    def adjust_difficulty(self, bit_clamp_factor: float, smallest_bit_difficulty: float) -> None:
        """
        Adjust the difficulty of the blockchain.

        This function is called every time a new block is added to the blockchain.
        It checks if the number of blocks in the blockchain is a multiple of the
        adjustment block interval. If it is, it calculates the average mining time
        of the last adjustment block interval and adjusts the difficulty of the
        blockchain accordingly.

        Args:
            bit_clamp_factor (float): The maximum allowable adjustment factor.
            smallest_bit_difficulty (float): The smallest bit difficulty that we can adjust to.

        Returns:
            None
        """
        if (len(self.blocks) - 1) % self.adjustment_block_interval == 0:
            # Calculate the average mining time of the last adjustment block interval
            average_mining_time_adjustment_interval: float = self.get_average_mining_time(
                self.adjustment_block_interval)
            # Calculate the reversed adjustment factor
            reversed_adjustment_factor: float = average_mining_time_adjustment_interval / self.target_block_mining_time

            # Log the average mining time and the reversed adjustment factor
            self.logger.debug(f"{AVERAGE_MINING_TIME_ADJUSTMENT_INTERVAL_KEY}: "
                f"{average_mining_time_adjustment_interval:.{DEFAULT_PRECISION}f}"
                              f" seconds")
            self.logger.debug(f"{REVERSED_ADJUSTMENT_FACTOR_KEY}: "
                              f"{reversed_adjustment_factor:.{DEFAULT_PRECISION}f}")

            # Get the last bit difficulty
            last_bit_difficulty: float = self.bit_difficulties[-1]

            if reversed_adjustment_factor > 0:
                # Calculate the bit adjustment factor
                bit_adjustment_factor: float = math.log2(reversed_adjustment_factor)
                # Clamp the bit adjustment factor
                clamped_bit_adjustment_factor: float = self.proof_of_work.clamp_bit_adjustment_factor(
                    bit_adjustment_factor, bit_clamp_factor)
                # Calculate the new bit difficulty
                new_bit_difficulty: float = max(
                    smallest_bit_difficulty,
                    last_bit_difficulty -
                    clamped_bit_adjustment_factor
                )
            else:
                # Set the new bit difficulty to the smallest bit difficulty if the reversed adjustment factor is 0 or negative
                new_bit_difficulty: float = smallest_bit_difficulty

            # Update the last bit difficulty in the blockchain
            self.bit_difficulties[-1] = new_bit_difficulty

    def log_difficulty_anomalies(self) -> None:
        """
        Logs any anomalies in the difficulty adjustments by comparing the actual bit difficulty values
        to the expected values calculated from the average mining times.

        The function iterates over the blocks in chunks of the adjustment block interval, calculates the
        average mining time for each chunk and the expected bit difficulty from the average mining time.
        If the actual bit difficulty value differs from the expected value, the function logs a warning.

        Returns:
            None
        """
        interval: int = self.adjustment_block_interval
        anomalies_detected: bool = False

        for i in range(interval, len(self.blocks), interval):
            avg_mining_time: float = sum(self.mining_times[i - interval + 1:i + 1]) / interval
            expected_factor: float = avg_mining_time / self.target_block_mining_time

            # Handle invalid expected_factor values
            if expected_factor <= 0:
                self.logger.error(f"Invalid expected_factor: {expected_factor}. Skipping log calculation.")
                continue

            expected_adjustment: float = math.log2(expected_factor)
            clamped_adjustment: float = self.proof_of_work.clamp_bit_adjustment_factor(
                expected_adjustment,

            expected_difficulty: float = max(self.smallest_bit_difficulty,
                self.bit_difficulties[i - interval] - clamped_adjustment)

            if abs(self.bit_difficulties[i] - expected_difficulty) > 1e-6:
                self.logger.critical(
                    f"Anomaly detected for blocks {i - interval + 1} to {i}:\n"
                    f"  Average Mining Time: {avg_mining_time:.6f}s\n"
                    f"  Expected Difficulty: {expected_difficulty:.6f}\n"
                    f"  Actual Difficulty: {self.bit_difficulties[i]:.6f}\n"
                )
                anomalies_detected = True

        if not anomalies_detected:
            self.logger.info("No adjust_difficulty anomalies detected")
```

Listing 5.2: Blockchain Implementation

This module supports secure and efficient blockchain operations by combining Proof-of-Work, dynamic difficulty adjustment, and comprehensive logging to uphold system integrity and transparency.

5.5 Proof-of-Work Implementation

The `proof_of_work.py` module implements the Proof-of-Work (PoW) algorithm, a cornerstone of blockchain security and integrity. This implementation ensures computational effort for mining blocks, deterring unauthorized modifications. It comprises the following key methods:

- `find_nonce`: Iteratively determines a nonce for a block such that the block's hash satisfies the difficulty threshold. The process continues until a valid nonce is found, with intermediate logging for debugging and progress tracking.

- `validate_proof`: Verifies that the block's hash meets the difficulty requirements, ensuring correctness and validity of the mined block.
- `clamp_bit_adjustment_factor`: Limits the bit adjustment factor within a predefined range, ensuring stability in mining difficulty.

5.5.1 Nonce Discovery

The `find_nonce` method is central to the mining process. It calculates the hash of a block repeatedly, adjusting the nonce until the hash meets the difficulty target. The target is derived from the `bit_difficulty`, with higher difficulty values necessitating greater computational effort. The process includes:

1. Initializing the target value based on the `bit_difficulty`.
2. Iteratively generating block hashes by updating the nonce.
3. Logging the mining progress every `NONCE_INCREMENT` attempts.
4. Stopping when the computed hash satisfies the difficulty requirement.

This method ensures secure and tamper-proof block creation while maintaining transparency through detailed logging.

5.5.2 Proof Validation

The `validate_proof` method evaluates whether a block's hash complies with the specified difficulty. It involves:

- Calculating the target value from the `bit_difficulty`.
- Converting the block's hash to a numerical value and comparing it with the target.
- Logging detailed validation results, including aligned hash and target values for clarity.

- Returning `True` if the block is valid, otherwise `False`.

This method upholds blockchain integrity by validating mined blocks rigorously.

5.5.3 Difficulty Adjustment Clamping

The `clamp_bit_adjustment_factor` method stabilizes mining operations by constraining difficulty adjustments. It achieves this by:

- Ensuring the adjustment factor is a numerical value within the permissible range.
- Preventing abrupt changes in difficulty that could destabilize the mining process.

By maintaining controlled adjustments, this method ensures a consistent and reliable blockchain network.

5.5.4 Integration with Hash Utilities

The `ProofOfWork` class leverages the `hash_utils.py` module for computing and validating block hashes during mining and proof validation. The `calculate_block_hash` function, invoked during nonce discovery, ensures secure and consistent hash generation across the blockchain.

The structure of the `ProofOfWork` class is as follows:

```python
#   Copyright (c) 2024, Anton Gorshkov
#   All rights reserved.
#   This code is for a pow and its unit tests.
#   For any questions or concerns, please contact Anton Gorshkov at antoniooreany@gmail.com

import math
from venv import logger

from src.model.block import Block
from src.constants import HASH_BIT_LENGTH, BASE, HEXADECIMAL_BASE, NONCE_INCREMENT
from src.utils.hash_utils import calculate_block_hash
from src.utils.logging_utils import log_mined_block


class ProofOfWork:
    def __init__(self) -> None:
        """
        Initializes a new instance of the ProofOfWork class.

        This class provides a proof of work algorithm to be used in a blockchain.
        It provides methods to find a nonce for a given block and to validate a
        block's proof of work.

        Returns:
            None
        """
        pass  # todo add any initialization logic here


    def find_nonce(self, block: Block, bit_difficulty: float) -> None:
        """
        Finds a nonce for a given block such that its hash is smaller than the target value.

        Args:
            block (Block): The block to find a nonce for.
            bit_difficulty (float): The difficulty level of the block.

        Returns:
            None
        """
        if block is None:
            raise ValueError("Block cannot be None")

        target_value: float = math.pow(BASE, HASH_BIT_LENGTH - bit_difficulty) - 1

        while True:
            block.hash = calculate_block_hash(
                index=block.index,
                timestamp=block.timestamp,  # Ensure consistent timestamp
                data=block.data,
                previous_block_hash=block.previous_hash,
                nonce=block.nonce,
            )
            if int(block.hash, HEXADECIMAL_BASE) < target_value:
                logger.debug(f"Found nonce for block {block.index}: {block.nonce}, Hash: {block.hash}")
                break
            block.nonce += 1
            if block.nonce % NONCE_INCREMENT == 0:
                logger.debug(f"Trying another {NONCE_INCREMENT} nonce {block.nonce} for block {block.index}, Hash: {block.hash}")


    def validate_proof(self, block: Block, bit_difficulty: float) -> bool:
        """
        Validate the proof of work for a given block.

        Args:
            block (Block): The block to validate.
            bit_difficulty (float): The difficulty level for the block's proof of work.

        Returns:
            bool: True if the block's hash meets the required difficulty, False otherwise.
        """
        # Calculate the target value based on bit
            difficulty
        target_value: float = pow(BASE, HASH_BIT_LENGTH -
            bit_difficulty) - 1

        # Convert the hash from hexadecimal to a numerical
             value
        hash_value: int = int(block.hash, HEXADECIMAL_BASE
            )

        # Format values with commas and ensure equal
            padding
        hash_value_str: str = f"{hash_value:,}"
        target_value_str: str = f"{int(target_value):,}"

        # Find the length of the longest string for
            alignment
        max_length: int = max(len(hash_value_str), len(
            target_value_str))

        # Add padding to align values
        hash_value_padded: str = hash_value_str.rjust(
            max_length)
        target_value_padded: str = target_value_str.rjust(
            max_length)

        # Log the values
        is_valid: bool = hash_value < target_value
        logger.debug(
            f"Validating Block {block.index}:\n"
            f"  Hash Value (int):   {hash_value_padded}\n"
            f"  Target Value (int): {target_value_padded}"
        )
        if is_valid:
            logger.info(f"Block {block.index} Validation:
                PASS")
            log_mined_block(block)
        else:
            logger.critical(f"Block {block.index}
                Validation: FAIL\n")

        # Return validation result
        return is_valid

```

```python
    def clamp_bit_adjustment_factor(self,
        bit_adjustment_factor: float, bit_clamp_factor:
        float) -> float:
        """
        Clamp the bit adjustment factor within the range
            determined by the bit clamp factor.

        This function takes two parameters, the bit
            adjustment factor and the bit clamp factor.
        The bit adjustment factor is the factor by which
            the bit difficulty is adjusted.
        The bit clamp factor is the maximum allowable
            adjustment factor.

        The function first calculates the minimum of the
            bit adjustment factor and the bit clamp factor,
        and then calculates the maximum of the result and
            the negative of the bit clamp factor.
        The final result is the clamped bit adjustment
            factor.

        Args:
            bit_adjustment_factor (float): The factor by
                which the bit difficulty is adjusted.
            bit_clamp_factor (float): The maximum
                allowable adjustment factor.

        Returns:
            float: The clamped bit adjustment factor.
        """
        if not isinstance(bit_adjustment_factor, (int,
            float)) or not isinstance(bit_clamp_factor, (
            int, float)):
            raise TypeError("bit_adjustment_factor and
                bit_clamp_factor must be numbers")
        if bit_clamp_factor < 0:
            raise ValueError("bit_clamp_factor must be a
                positive number")

        clamped_bit_adjustment_factor: float = max(-
            bit_clamp_factor, min(bit_adjustment_factor,
            bit_clamp_factor))
        return clamped_bit_adjustment_factor
```

Listing 5.3: Proof-of-Work Implementation

This module ensures secure and efficient mining while maintaining blockchain integrity through robust validation and controlled difficulty adjustments.

5.6 Hash Utilities Implementation

The `hash_utils.py` module provides essential utility functions for hash computation and validation within the blockchain system. It ensures that each block is uniquely identifiable and tamper-proof by generating secure cryptographic hashes. The module includes the following key functions:

- `calculate_block_hash:` Computes the hash of a block by combining its attributes (index, timestamp, data, previous block hash, and nonce). It employs the SHA-256 hashing algorithm to generate a unique hexadecimal string that serves as the block's hash.
- `validate_block_attributes:` Ensures the validity of block attributes by checking that they are not null or empty. This validation step prevents the creation of invalid or incomplete blocks.

5.6.1 Hash Computation

The `calculate_block_hash` function plays a pivotal role in the Proof-of-Work process by generating the hash required for mining and validation. The function works as follows:

1. It combines the block's attributes (index, timestamp, data, previous hash, and nonce) into a single string.
2. The string is encoded using the specified `ENCODING` (defined in the constants module).
3. The SHA-256 hashing algorithm processes the encoded string to produce a secure, fixed-length hash.

This hash acts as the block's unique identifier, ensuring that any modification to the block's attributes results in a completely different hash.

While the method for block hash calculation can vary across different blockchain implementations, it must remain consistent within a particular blockchain. This consistency ensures that all participants in the network can validate blocks reliably, maintaining the integrity of the blockchain.

5.6.2 Attribute Validation

The `validate_block_attributes` function ensures data integrity by verifying that all block attributes are valid. It checks each attribute as follows:

- Ensures the attribute is not `None`.
- For string attributes, confirms that they are not empty.

By enforcing these checks, the function ensures that all blocks contain complete and consistent data before undergoing the hashing process.

5.6.3 Integration with Proof-of-Work

The `hash_utils.py` module integrates seamlessly with the `ProofOfWork` class, enabling efficient and secure mining operations. During the mining process, the `calculate_block_hash` function is invoked repeatedly as the nonce is adjusted, ensuring the hash meets the required difficulty level. The computed hash is then validated to confirm adherence to the blockchain's security standards.

The structure of the `hash_utils.py` module is as follows:

5.7 Difficulty Adjustment Mechanism

The blockchain dynamically adjusts its mining difficulty to maintain consistent block generation times, utilizing the `adjust_difficulty` method in the `blockchain.py` module. This mechanism ensures the blockchain remains adaptive and resilient to variations in computational power. The core aspects of this mechanism are as follows:

- **Hash Calculation:** The `calculate_block_hash` function, described in the `hash_utils.py` module, generates a unique hash for each block by concatenating its attributes (index, timestamp, data, previous hash, and nonce) and applying the SHA-256 algorithm. This ensures that each block has a tamper-evident identifier, critical for blockchain security.

- **Average Mining Time Computation:** The system computes the average mining time for a defined adjustment interval, excluding the genesis block. This metric reflects the network's current computational capacity and guides difficulty adjustments.

- **Difficulty Adjustment:** The `adjust_difficulty` method calculates a reversed adjustment factor based on the computed average mining time and the target block mining time. The factor is clamped within a predefined range to prevent drastic difficulty fluctuations. Finally, the bit difficulty is logarithmically adjusted to align mining challenges with the blockchain's performance goals.

This adaptive mechanism ensures that block mining rates remain consistent even as the computational environment changes. The hash generation process, powered by the `calculate_block_hash` function, serves as the foundation of the Proof-of-Work algorithm, ensuring that the blockchain remains secure and resistant to tampering.

The implementation of the calculate block hash function is shown in Listing 5.4, demonstrating its role in validating and generating secure block hashes.

```python
#   Copyright (c) 2024, Anton Gorshkov
#   All rights reserved.
#   This code is for a pow and its unit tests.
#   For any questions or concerns, please contact Anton
    Gorshkov at antoniooreany@gmail.com

# path: src/utils/hash_utils.py

import hashlib
from src.constants import ENCODING

def calculate_block_hash(
        index: int, timestamp: float, data: str,
            previous_block_hash: str, nonce: int
) -> str:
    """Compute the hash of a block by combining its
        attributes.

    Args:
        index: int: The position of the block in the
            blockchain.
        timestamp: float: The time at which the block was
            created.
        data: str: The data contained within the block.
        previous_block_hash: str: The hash of the previous
             block in the chain.
        nonce: int: The nonce for the proof of work
            algorithm.

    Returns:
        str: The hash of the block as a hexadecimal string
            .
    """
    # Validate the arguments
    validate_block_attributes(
        index=index,
        timestamp=timestamp,
        data=data,
        previous_block_hash=previous_block_hash,
        nonce=nonce
    )

    # Combine the arguments into a single string
    data_to_hash: bytes = (
        f"{index}{timestamp}{data}{previous_block_hash}{
            nonce}"
    ).encode(ENCODING)

    # Compute the hash of the string
    hash_object: hashlib._Hash = hashlib.sha256()
    hash_object.update(data_to_hash)

    # Return the hash as a hexadecimal string
    return hash_object.hexdigest()


def validate_block_attributes(
        index: int, timestamp: float, data: str, previous_block_hash: str, nonce: int
) -> None:
    """Validate that all block attributes are not null or empty.

    Validate that all block attributes are not null or empty.
    This ensures that all blocks have valid attributes.

    Args:
        index: int: The position of the block in the blockchain.
        timestamp: float: The time at which the block was created.
        data: str: The data contained within the block.
        previous_block_hash: str: The hash of the previous block in the chain.
        nonce: int: The nonce for the proof of work algorithm.

    Returns:
        None
    """
    # Validate each attribute
    for attribute, name in [
        (index, "index"),
        (timestamp, "timestamp"),
        (data, "data"),
        (previous_block_hash, "previous_block_hash"),
        (nonce, "nonce"),
    ]:
        # Check if the attribute is None
        if attribute is None:
            raise ValueError(f"{name}: {attribute!r}
                cannot be null")
        # Check if the attribute is an empty string
        if isinstance(attribute, str) and not attribute:
            raise ValueError(f"{name}: {attribute!r}
                cannot be empty")
```

Listing 5.4: Hash Utilities

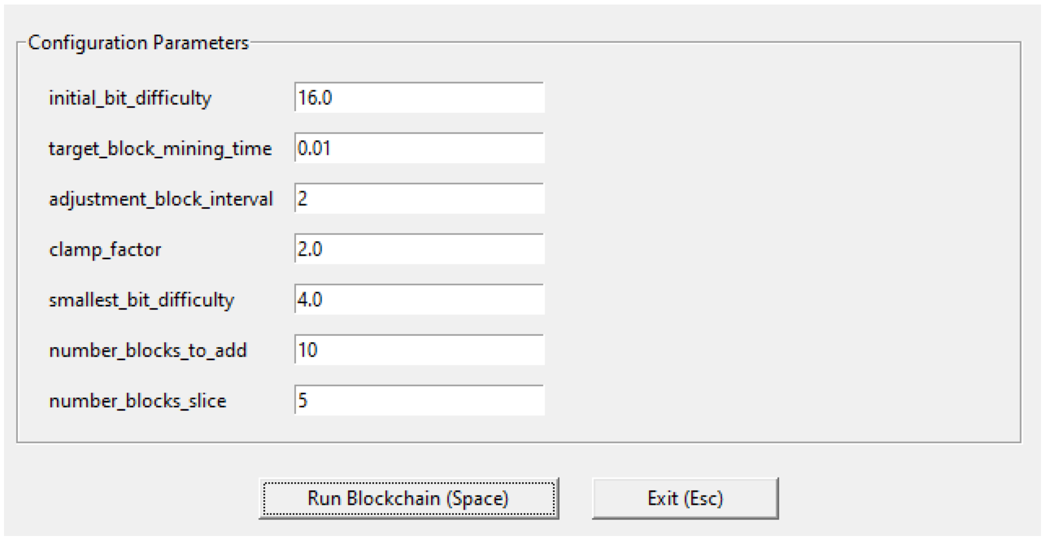
By combining robust hash generation, accurate mining time computation, and controlled difficulty adjustments, the blockchain prototype achieves a realistic and secure implementation. These features make it a valuable tool for understanding blockchain dynamics and experimenting with adaptive mechanisms.

5.8 Executing the Blockchain with Configurable Parameters

The blockchain prototype provides a graphical user interface (GUI) for configuring and running the blockchain simulation. The updated GUI, depicted in Figure 5.1, allows users to set key parameters dynamically, facilitating experimentation with blockchain behavior.

The following parameters are available for configuration:

- **Initial Bit Difficulty:** The starting difficulty level for mining blocks. This establishes the baseline challenge for solving cryptographic puzzles.
- **Target Block Mining Time:** The desired average time for mining each block. This value ensures a rapid mining environment suitable for simulation purposes.
- **Adjustment Block Interval:** The interval, at which mining difficulty is recalculated based on recent mining times. This ensures the blockchain remains responsive to changes in computational power.
- **Clamp Factor:** Limits the maximum allowable adjustment of difficulty, stabilizing the mining process by preventing extreme fluctuations.


Configuration Parameters

initial_bit_difficulty	16.0
target_block_mining_time	0.01
adjustment_block_interval	2
clamp_factor	2.0
smallest_bit_difficulty	4.0
number_blocks_to_add	10
number_blocks_slice	5

Figure 5.1: Graphical User Interface (GUI) for Running the Blockchain Prototype

- **Smallest Bit Difficulty:** Defines the minimum allowable difficulty, ensuring mining complexity does not drop to insecure levels.
- **Number of Blocks to Add:** Specifies the total number of blocks to be mined during the session.
- **Number Blocks Slice:** Determines the number of blocks used for performance calculations.

Users can initiate the blockchain simulation by selecting the `Run Blockchain` button in the GUI. This triggers block mining using the Proof-of-Work algorithm, leveraging the configuration parameters.

5.9 Intentional Simplifications of the Prototype

The developed blockchain prototype is a simplified representation of a real-world Proof-of-Work (PoW) system. The following simplifications were introduced to focus on key functionalities and reduce complexity:

- **Single Miner:** The prototype simulates a blockchain with only one miner, eliminating competition and network effects seen in real-world blockchains.
- **Limited Blocks:** Only a fixed number of blocks are mined to analyze performance and measure stochastic parameters, rather than continuous mining as seen in operational blockchains.
- **No Mining Reward System:** The prototype does not implement a reward mechanism for mining, as the focus is on validating the difficulty adjustment and mining process.
- **Mock Data:** Blocks are mined with mock data rather than actual transactions. Transaction creation and validation are not implemented.

5.10 Code Repository

The complete code for the blockchain prototype, including all implementations discussed in this chapter, is available on GitHub. You can find the full source code, along with additional resources for setup and usage instructions, at the following link:

`https://github.com/antoniooreany/Blockchain-PoW-Consensus`

This repository includes all necessary files for setting up and running the blockchain prototype, as well as detailed documentation for further exploration and experimentation.

5.11 UML Diagram of the Blockchain Prototype

To understand the structure and relationships of the blockchain components, the UML diagram below illustrates the main classes in this prototype and their interactions:

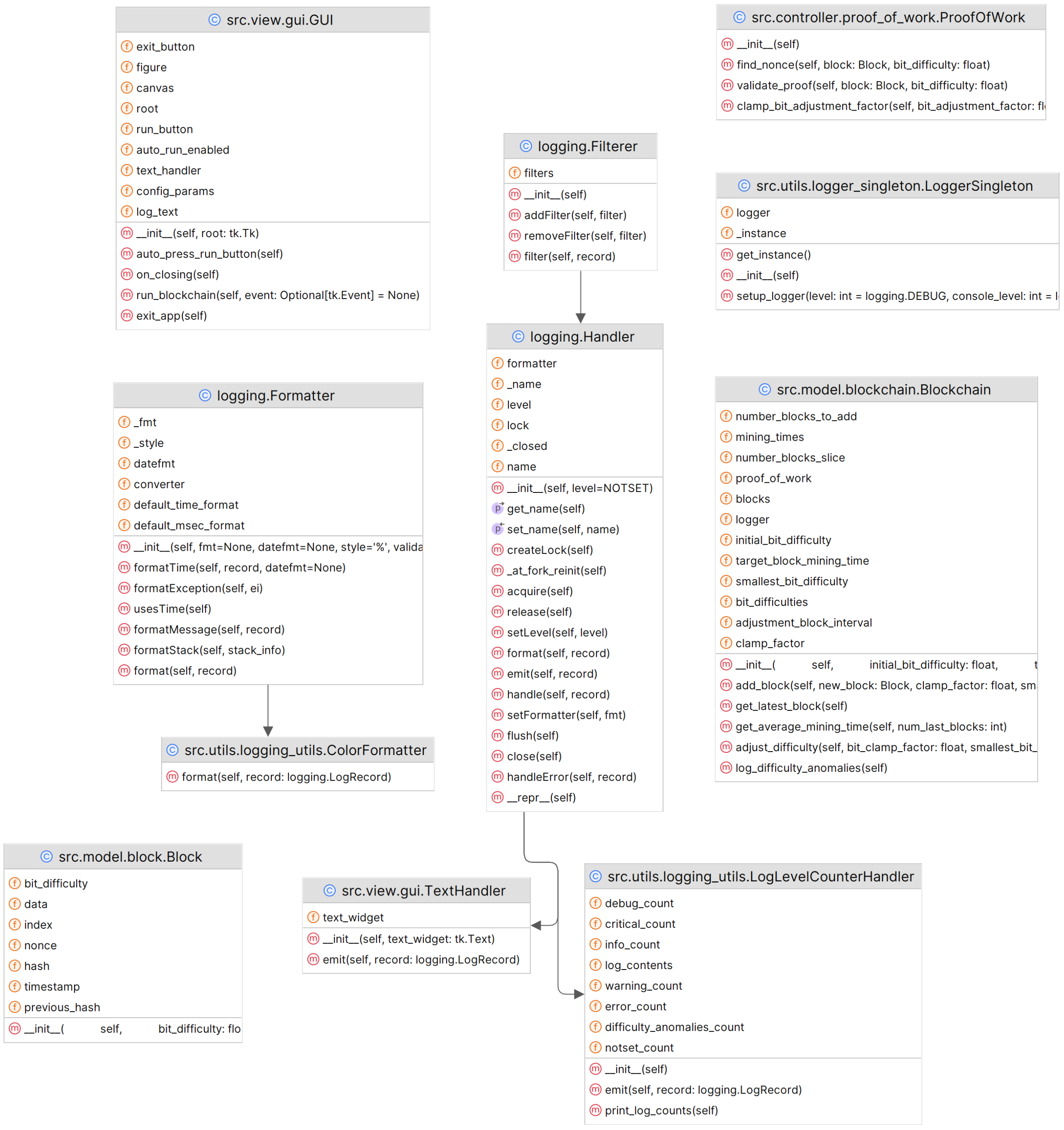

src.view.gui.GUI
exit_button
figure
canvas
root
run_button
auto_run_enabled
text_handler
config_params
log_text
__init__(self, root: tk.Tk)
auto_press_run_button(self)
on_closing(self)
run_blockchain(self, event: Optional[tk.Event] = None)
exit_app(self)
src.controller.proof_of_work.ProofOfWork
__init__(self)
find_nonce(self, block: Block, bit_difficulty: float)
validate_proof(self, block: Block, bit_difficulty: float)
clamp_bit_adjustment_factor(self, bit_adjustment_factor: fl
logging.Filterer
filters
__init__(self)
addFilter(self, filter)
removeFilter(self, filter)
filter(self, record)
src.utils.logger_singleton.LoggerSingleton
logger
_instance
get_instance()
__init__(self)
setup_logger(level: int = logging.DEBUG, console_level: int = l
logging.Handler
formatter
_name
level
lock
_closed
name
__init__(self, level=NOTSET)
get_name(self)
set_name(self, name)
createLock(self)
_at_fork_reinit(self)
acquire(self)
release(self)
setLevel(self, level)
format(self, record)
emit(self, record)
handle(self, record)
setFormatter(self, fmt)
flush(self)
close(self)
handleError(self, record)
__repr__(self)
logging.Formatter
_fmt
_style
datefmt
converter
default_time_format
default_msec_format
__init__(self, fmt=None, datefmt=None, style='%', valida
formatTime(self, record, datefmt=None)
formatException(self, ei)
usesTime(self)
formatMessage(self, record)
formatStack(self, stack_info)
format(self, record)
src.utils.logging_utils.ColorFormatter
format(self, record: logging.LogRecord)
src.model.blockchain.Blockchain
number_blocks_to_add
mining_times
number_blocks_slice
proof_of_work
blocks
logger
initial_bit_difficulty
target_block_mining_time
smallest_bit_difficulty
bit_difficulties
adjustment_block_interval
clamp_factor
__init__(self, initial_bit_difficulty: float, t
add_block(self, new_block: Block, clamp_factor: float, sm
get_latest_block(self)
get_average_mining_time(self, num_last_blocks: int)
adjust_difficulty(self, bit_clamp_factor: float, smallest_bit_
log_difficulty_anomalies(self)
src.model.block.Block
bit_difficulty
data
index
nonce
hash
timestamp
previous_hash
__init__(self, bit_difficulty: flo
src.view.gui.TextHandler
text_widget
__init__(self, text_widget: tk.Text)
emit(self, record: logging.LogRecord)
src.utils.logging_utils.LogLevelCounterHandler
debug_count
critical_count
info_count
log_contents
warning_count
error_count
difficulty_anomalies_count
notset_count
__init__(self)
emit(self, record: logging.LogRecord)
print_log_counts(self)

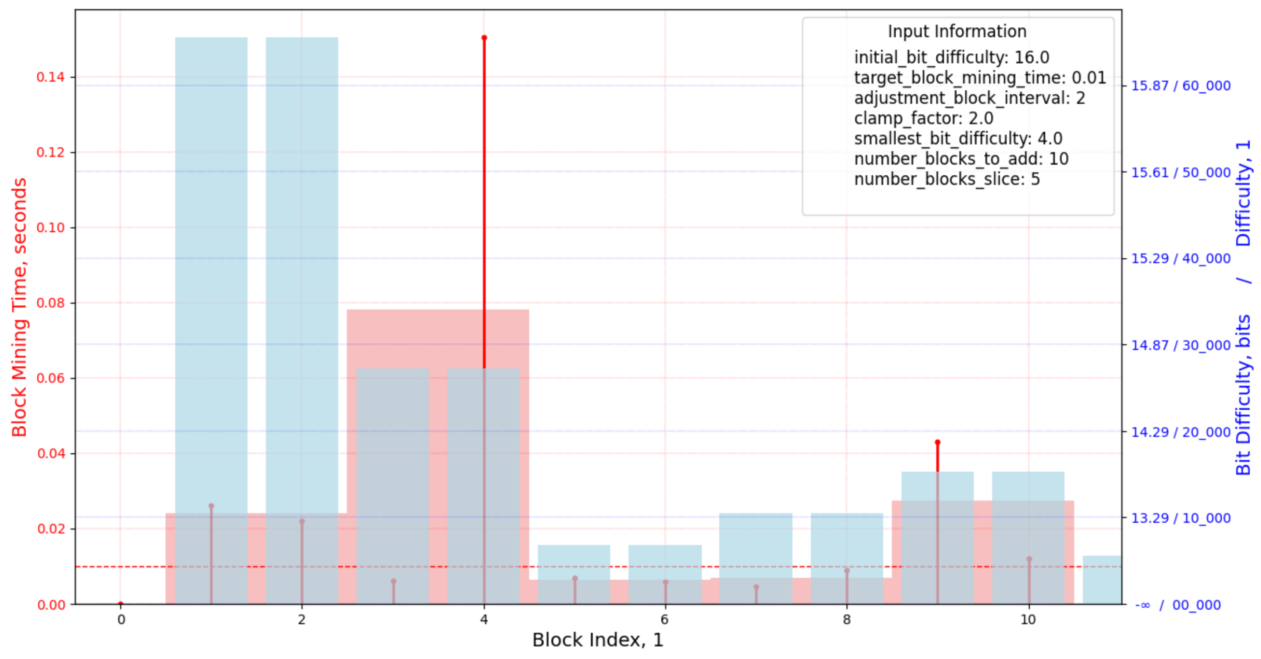
Figure 5.2: UML Diagram of the Blockchain Prototype

Chapter 6

Results

6.1 Analyzing Blockchain Mining

Upon completion, the system generates a Blockchain Mining Statistics graph, as shown in Figure 6.1. This visualization highlights key metrics, such as the target mining time for the current blockchain execution (depicted as a horizontal dashed line), the actual mining time per block (shown in red), the average mining time within difficulty adjustment intervals, and the evolving bit difficulty levels (shown in blue).



Input Information
initial_bit_difficulty: 16.0
target_block_mining_time: 0.01
adjustment_block_interval: 2
clamp_factor: 2.0
smallest_bit_difficulty: 4.0
number_blocks_to_add: 10
number_blocks_slice: 5

Figure 6.1: Blockchain Mining Plot: Difficulty and Mining Time as functions of Block Index

6.1.1 Interpretation of the Graph

The graph provides insights into the blockchain's mining performance and dynamic difficulty adjustments:

- **Block Mining Time:** The red bars indicate the time in seconds required to mine each block. While fluctuations occur due to the stochastic nature of the mining process, the average mining time aligns closely with the target value of 0.01 seconds, demonstrating the blockchain's ability to adjust dynamically.

- **Difficulty Adjustments:** The blue bars show the bit difficulty recalibration that occurs every 2 blocks, as configured. These adjustments ensure that the average mining time remains consistent with the target.

- **Performance Evaluation Slice:** Using the `Number Blocks Slice` parameter, mining performance is evaluated over the last 5 blocks. The performance evaluation is made only on this number of the last blocks to check the properties of the blockchain without any involvement of the external data, such as initial difficulty, etc.

6.2 Impact of Fractional `bit_difficulty`

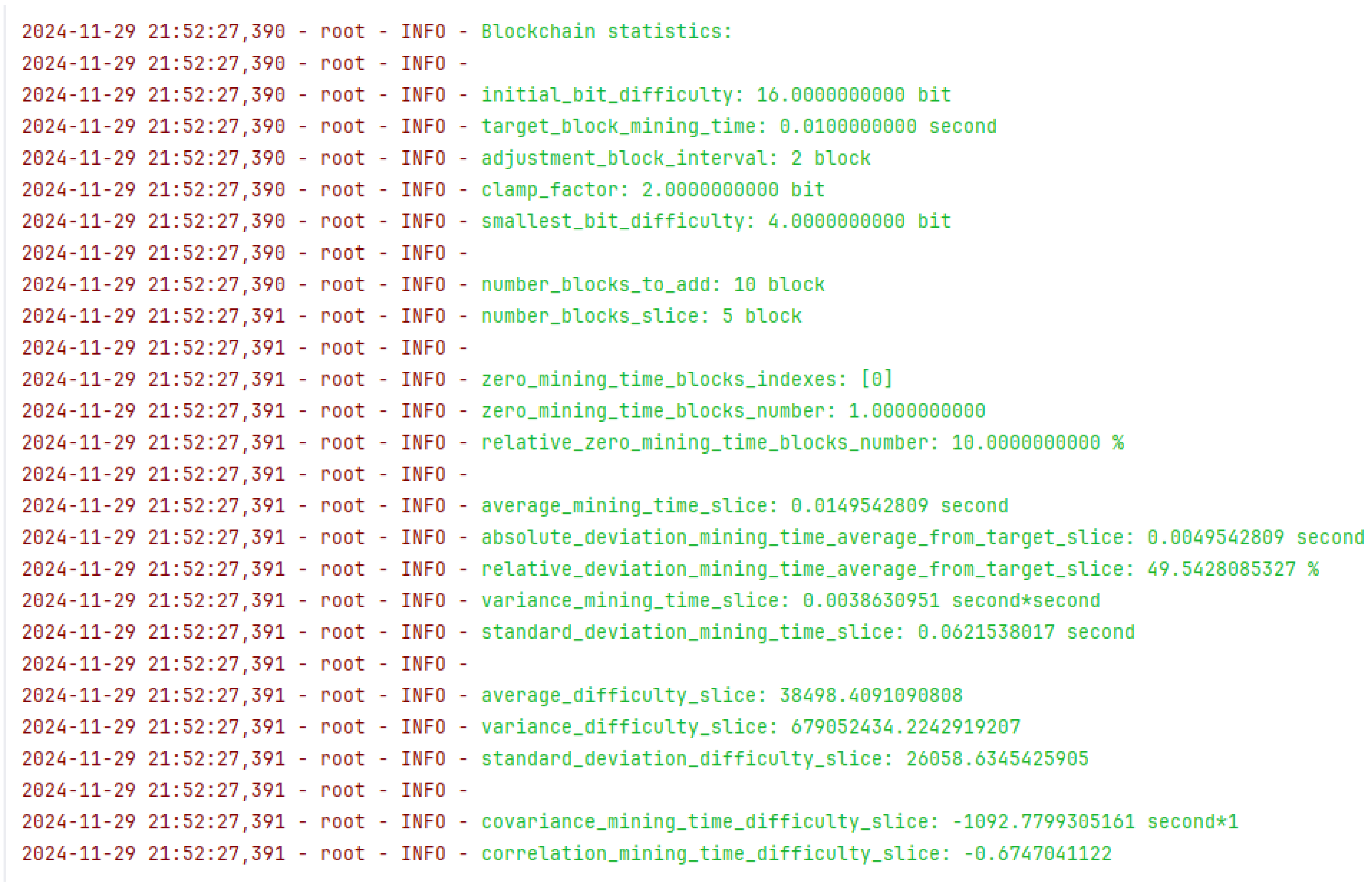
The fractional ‘bit_difficulty‘ demonstrated its effectiveness in maintaining stable mining times. The following observations were made:

- **Alignment with Target Time:** The average mining time aligned closely with the target time, as shown in the Blockchain Mining Statistics graph (Figure 6.1).
- **Stability in a Single-Miner Setup:** The dynamic adjustment mechanism effectively prevented abrupt changes in difficulty, even in the simplified single-miner setup used in this prototype.
- **Smooth Transitions:** The fractional ‘bit_difficulty‘ allowed for more granular adjustments, resulting in smoother transitions in difficulty levels and reducing fluctuations in block production rates compared to traditional integer-based adjustments.

This approach demonstrates the advantages of fractional ‘bit_difficulty‘ in improving blockchain stability and aligning mining performance with desired targets.

6.2.1 Blockchain Performance Metrics

The blockchain execution generates detailed performance metrics, as shown in Figure 6.2. These statistics, taken directly from the console log, provide a comprehensive understanding of the mining process, including stochastic variability, dynamic adjustments, and overall stability.



```
2024-11-29 21:52:27,390 - root - INFO - Blockchain statistics:
2024-11-29 21:52:27,390 - root - INFO - 
2024-11-29 21:52:27,390 - root - INFO - initial_bit_difficulty: 16.0000000000 bit
2024-11-29 21:52:27,390 - root - INFO - target_block_mining_time: 0.0100000000 second
2024-11-29 21:52:27,390 - root - INFO - adjustment_block_interval: 2 block
2024-11-29 21:52:27,390 - root - INFO - clamp_factor: 2.0000000000 bit
2024-11-29 21:52:27,390 - root - INFO - smallest_bit_difficulty: 4.0000000000 bit
2024-11-29 21:52:27,390 - root - INFO - 
2024-11-29 21:52:27,390 - root - INFO - number_blocks_to_add: 10 block
2024-11-29 21:52:27,391 - root - INFO - number_blocks_slice: 5 block
2024-11-29 21:52:27,391 - root - INFO - 
2024-11-29 21:52:27,391 - root - INFO - zero_mining_time_blocks_indexes: [0]
2024-11-29 21:52:27,391 - root - INFO - zero_mining_time_blocks_number: 1.0000000000
2024-11-29 21:52:27,391 - root - INFO - relative_zero_mining_time_blocks_number: 10.0000000000 %
2024-11-29 21:52:27,391 - root - INFO - 
2024-11-29 21:52:27,391 - root - INFO - average_mining_time_slice: 0.0149542809 second
2024-11-29 21:52:27,391 - root - INFO - absolute_deviation_mining_time_average_from_target_slice: 0.0049542809 second
2024-11-29 21:52:27,391 - root - INFO - relative_deviation_mining_time_average_from_target_slice: 49.5428085327 %
2024-11-29 21:52:27,391 - root - INFO - variance_mining_time_slice: 0.0038630951 second*second
2024-11-29 21:52:27,391 - root - INFO - standard_deviation_mining_time_slice: 0.0621538017 second
2024-11-29 21:52:27,391 - root - INFO - 
2024-11-29 21:52:27,391 - root - INFO - average_difficulty_slice: 38498.4091090808
2024-11-29 21:52:27,391 - root - INFO - variance_difficulty_slice: 679052434.2242919207
2024-11-29 21:52:27,391 - root - INFO - standard_deviation_difficulty_slice: 26058.6345425905
2024-11-29 21:52:27,391 - root - INFO - 
2024-11-29 21:52:27,391 - root - INFO - covariance_mining_time_difficulty_slice: -1092.7799305161 second*1
2024-11-29 21:52:27,391 - root - INFO - correlation_mining_time_difficulty_slice: -0.6747041122
```

Figure 6.2: Detailed Blockchain Statistics Log

Key metrics and their interpretations are as follows:

- **Initial Parameters:**
  - *Initial Bit Difficulty:* 16 bits, establishing the initial computational challenge, meaning that at least 16 leading binary zeros (or 4 hexadecimal zeros) are required for the valid hash to mine the first block. However, for the block with index 0, which is Genesis Block, the 'bit_difficulty' is chosen to be 0.
  - *Target Block Mining Time:* 0.01 seconds, defining the desired average mining time.

  - *Adjustment Block Interval:* Recalibrations occur every 2 blocks to maintain target mining times.
  - *Clamp Factor:* Limits difficulty changes to 2 bits per adjustment interval to prevent abrupt shifts.
  - *Smallest Bit Difficulty:* A minimum of 4 bits ensures continued operation during low computational demand.

- **Zero Mining Time Blocks:**
  - *Indexes:* Blocks with indexes [0] showed a zero mining time. The index 0 is an index of the Genesis block, which in our implementation normally mined instantly, because of its 'bit_difficulty' is 0. Therfore it's not an anomaly.
  - *Count:* One block showed zero mining time due to the fact that the bit_difficulty was 0.
  - *Relative Proportion:* 10% is quite high because of the small amount of the blocks have been mined.

This behavior, while rare, can be attributed to the single-miner setup of the prototype, which limits the stochastic variability of mining times. In a distributed network, multiple miners would contribute more variability, reducing the likelihood of such anomalies.

- **Mining Time Analysis:**
  - *Average Mining Time for the Slice:* 0.0149 seconds, slightly exceeding the target due to small inefficiencies. By "slice," this analysis refers to the last specified number of mined blocks (e.g., 5 blocks) that are considered for statistical research and performance evaluation.
  - *Deviation from Target:*
    - Absolute Deviation: 0.0049 seconds.
    - Relative Deviation: 49.54%, reflecting fluctuations caused by the mining process's stochastic nature in complex with the single-miner system.
  - *Variance and Standard Deviation:*
    - Variance: 0.00836 seconds^2.

        - Standard Deviation: 0.0622 seconds, indicating moderate variability.

- **Difficulty Analysis:**
    - *Average Difficulty for the Slice:* 38498.41, reflecting dynamic adjustments to maintain consistent mining performance. Similarly, "slice" refers to the interval of blocks used to calculate these statistical values, ensuring a meaningful average over recent performance. These values serve as intermediate metrics to provide context for key analyses, such as correlation and covariance, which validate the correctness of the difficulty adjustment mechanism.
    - *Variance and Standard Deviation:*
        - Variance: 679052434.
        - Standard Deviation: 26058.63.
- **Correlation Analysis:**
    - *Mining Time and Bit Difficulty:*
        - Covariance: -1092.78 seconds*bit.
        - Correlation: -0.6747, indicating a moderate inverse relationship. This behavior aligns with the system design: as mining time increases, the difficulty decreases to stabilize the mining process. The term "slice" here denotes the blocks considered for deriving this relationship, emphasizing that the results are focused on a defined and consistent subset of blocks.

6.2.2 Prototype Limitations

The results presented in this chapter are based on a simplified blockchain prototype. The following limitations of the implementation should be considered when interpreting the findings:

- **Single Miner System:** The single-miner setup eliminates competition and limits the stochastic diversity of mining times.
- **Finite Block Set:** Mining is performed on a limited number of blocks, focusing on performance metrics and stochastic behavior within this subset.

- **Absence of Rewards:** Mining rewards are not implemented, removing economic incentives typically associated with PoW systems.
- **Mock Data:** Blocks are mined using mock data without real transactions, excluding the effects of transaction size and complexity on performance.

6.2.3 Conclusions from the Results

The results validate the blockchain prototype's ability to dynamically adjust difficulty, maintaining target mining times. Key conclusions include:

- The negative correlation between mining time and difficulty demonstrates that the adjustment mechanism is functioning the right way, stabilizing performance.
- The single-miner setup simplifies the development and testing but leads to deviations in the stochastic properties of the blockchain, limiting the realism of certain metrics.
- Despite limitations, the usage of the clamp factor and the smallest bit difficulty effectively control difficulty fluctuations, ensuring a more stable and predictable mining environment.

These findings highlight the prototype's success in replicating core blockchain behaviors while identifying areas for potential enhancements in scalability and realism.

Chapter 7

Conclusion

The development and execution of this blockchain prototype have successfully demonstrated the essential components and principles of blockchain technology. By implementing and analyzing features such as block creation, Proof-of-Work validation, and dynamic difficulty adjustment, this project has provided practical insights into the interplay of these mechanisms in maintaining a secure and efficient blockchain system. Despite the simplified nature of this prototype, it serves as an effective educational and experimental tool for understanding blockchain dynamics.

7.1 Key Achievements and Contributions

This prototype successfully implemented a Python-based blockchain, incorporating critical elements that illustrate the functionality and adaptability of blockchain systems. Key contributions include:

- **Robust Blockchain Structure:** The `Block` and `Blockchain` classes created a secure and immutable chain where each block is cryptographically linked to its predecessor. This ensures data integrity and serves as the foundation of trust in the blockchain.

- **Efficient Proof-of-Work Implementation:** The `ProofOfWork` class effectively demonstrated the computational effort required to mine blocks, reinforcing blockchain security by making tampering computationally prohibitive. The algorithm dynamically adjusted the nonce to find hashes that satisfied the target difficulty.

- **Dynamic Difficulty Adjustment:** The adaptive difficulty mechanism recalibrated every 2 blocks, aligning the mining difficulty with the target block production rate. This feature illustrated how blockchain systems maintain stability despite fluctuating conditions.

- **Comprehensive Performance Analysis:** Through data visualization and console-based statistical logs, the prototype provided valuable insights into mining performance, stochastic variability, and difficulty adjustment behavior over a session of 10 mined blocks.

7.2 Insights and Lessons Learned

The execution and analysis of this prototype revealed several important insights into blockchain technology, as well as challenges that arise in simplified experimental setups:

- **Impact of Dynamic Difficulty Adjustment:** The difficulty adjustment mechanism effectively stabilized the mining process, ensuring the average mining time remained close to the target of 0.01 seconds. The use of a clamp factor prevented extreme fluctuations in difficulty, demonstrating its importance for maintaining blockchain stability.

- **Limitations of a Single-Miner Setup:** The prototype relied on a single miner, simplifying implementation but limiting stochastic variability in mining times. Real-world blockchains with multiple miners introduce additional complexity and variability, making the metrics observed in this prototype more deterministic.

- **Significance of Negative Correlation:** The observed negative correlation between mining time and difficulty validated the proper functioning of the difficulty adjustment mechanism. As expected, increased mining times resulted in reduced difficulty, aligning the mining process with the target block production rate.

- **Performance Challenges with Anomalies:** It seems like an anomaly was observed where the block exhibited zero mining time. This occurred at index 0, which corresponds to the Genesis block. The bit difficulty for the Genesis block was deliberately set to 0 to allow the blockchain to start quickly, even in scenarios with a small amount of miners. In this case, any nonce is valid, resulting in zero mining time, meaning: that was not an anomaly. While this decision simplifies initialization, it highlights the need for robust mechanisms to handle such edge cases in more complex implementations, especially where computational diversity is limited.

- **Scalability Concerns:** Although the prototype successfully mined 10 blocks with consistent performance, scaling this system would require optimizations to handle larger blockchains. Efficient data management and improved algorithms would be essential for practical implementations.

7.3 Final Remarks

This blockchain prototype illustrates the core principles of blockchain technology, such as Proof-of-Work, dynamic difficulty adjustment, and cryptographic block linking, within a highly simplified framework. It excludes real-world complexities like multi-node communication, decentralized consensus, and realistic transaction handling, opting for a single-node setup and generic data structures.

Simplifications include streamlined difficulty adjustments, single-process mining, and the absence of variability in mining power or external conditions. These reductions focus on core functionalities, offering a clear, educational model to understand blockchain mechanics while omitting the intricacies of large-scale, distributed systems.

Chapter 8

Future Work

While the current blockchain prototype successfully demonstrates fundamental blockchain concepts, certain simplifications were implemented to focus on key functionalities and reduce complexity. These simplifications, as outlined in Section 5.9, provide a clear roadmap for future enhancements. Addressing these areas will help bridge the gap between the prototype and real-world blockchain systems.

8.1 Implementing Multiple Miners

Introducing multiple miners would simulate the competitive environment of real-world blockchains, allowing for:

- Analysis of mining competition and its impact on performance.
- Implementation of network synchronization and consensus mechanisms.
- Resolution of forks caused by simultaneous block production.

8.2 Continuous Mining

Transitioning from mining a limited number of blocks to real-time continuous mining would enable:

- Long-term performance evaluation.
- Robust analysis of blockchain behavior under sustained activity.

8.3 Mining Reward System

Adding a mining reward mechanism would provide:

- Realistic incentives for miners.
- The foundation for integrating a cryptocurrency into the prototype.

8.4 Realistic Transaction Data

Replacing mock data with actual transactions would involve:

- Implementing a transaction pool (mempool) for miners to process.
- Introducing validation and ordering mechanisms for transactions.
- Simulating user interactions to generate realistic blockchain activity.

8.5 Conclusion

Addressing these features would bridge the gap between the current prototype and real-world blockchain systems. These improvements provide a logical pathway for enhancing the prototype's realism, scalability, and functionality, making it a valuable tool for further research and experimentation.

Appendix A

Cryptographic Hashing

A.1 What is a Hash Function?

A **hash function** is a deterministic mathematical function that takes an arbitrary-sized input (also referred to as a *message*) and produces a fixed-size output, commonly referred to as a *hash value* or *digest*. This digest is a representation of the input data, and even a small change in the input (e.g., changing one character or bit) results in a drastically different hash value, a property known as the **avalanche effect**. The output size is usually much smaller than the input data, making hash functions efficient for summarizing and identifying large data sets.

Cryptographic hash functions have several important properties that make them suitable for use in securing information:

- **Deterministic**: The same input always produces the same output.
- **Fast computation**: Hash values can be computed efficiently, even for large inputs.
- **Pre-image resistance**: It should be computationally infeasible to reverse the process, i.e., to determine the original input given only the hash output.
- **Small changes in input result in large changes in output**: A single bit difference in the input should cause the hash output to change completely, making it difficult for similar inputs to have related hash values.

- **Fixed output size**: Regardless of the input size, the output length is fixed. For example, SHA-256 always produces a 256-bit hash. [1]

A.2 Origins of Hash Functions

Hash functions initially emerged in the field of computer science, particularly for use in **hash tables**, a data structure that allows for efficient data retrieval [16]. The concept of hashing was first formalized in the 1950s as a way to map data of arbitrary size to a fixed-size array. The primary goal of hash tables is to minimize access time and efficiently locate or store data based on a unique identifier generated by the hash function.

In the 1970s, the importance of hash functions grew beyond data indexing with the rise of **cryptography**. As digital communication expanded, so did the need for tools to ensure secure data transmission, integrity, and authenticity. Early cryptographic hash functions, such as MD5 and SHA-1, were developed in response to these needs. Cryptographers realized that hash functions could be used not only for data storage but also for securing sensitive information.

A.3 History and Development of Cryptographic Hash Functions

Hash functions have existed for several decades, with their roots tracing back to the early stages of computer science. Over time, they evolved to play a critical role in modern cryptographic protocols.

- **1950s**: The concept of hash functions emerged as a means of improving data retrieval speeds in hash tables. Early uses were primarily for data indexing and quick lookup purposes in databases and memory management.

- **1970s-1980s**: The first cryptographic hash functions, such as MD2, MD4, and MD5, were developed by Ronald Rivest as part of the **Message Digest (MD)** family. These early cryptographic hash functions laid the groundwork for more advanced cryptographic protocols, including digital signatures and encryption.

- **1990s**: The **Secure Hash Algorithm (SHA)** family was developed by the National Security Agency (NSA) and adopted as a federal standard by NIST. SHA-1 was introduced in 1993 and became widely used for securing data in digital signatures and certificates.

- **2000s**: Due to vulnerabilities discovered in SHA-1 and MD5, newer hash functions such as **SHA-256** and **SHA-3** were developed, offering increased security and resistance to modern cryptographic attacks [2].

A.4 Applications of Hash Functions

Cryptographic hash functions are fundamental components of modern cryptography. They are used in a wide range of applications, including:

- **Data Integrity Verification**: Hash functions are used to verify that data has not been tampered with [12]. For example, when downloading a file, a hash value (also known as a checksum) may be provided. The user can compute the hash of the downloaded file and compare it with the provided value. If the two hashes match, the file is verified as intact.

- **Digital Signatures**: In digital signature schemes, the hash of a message is signed rather than the message itself [17]. This ensures that the signature is efficient (since the hash is much smaller than the message) and secure (since the hash is unique to the message).

- **Password Hashing**: Instead of storing plaintext passwords, systems store hashes of passwords ??? [8]. When a user attempts to log in, the system hashes the entered password and compares it to the stored hash. This approach enhances security, as it prevents attackers from obtaining user passwords even if they access the database.

- **Blockchain and Cryptocurrencies**: Hash functions are integral to the functioning of blockchain technology, where they ensure data immutability and are used in proof-of-work mechanisms. In blockchain, each block contains the hash of the previous block, creating a secure and tamper-proof chain.

A.5 Visual Representation of a Hash Function

A.6 Conclusion

Cryptographic hash functions are an essential tool in modern computing, enabling secure communication, data integrity, and verification. The development of these functions has evolved significantly from their early use in data indexing to their current applications in cryptography. As threats to digital security continue to evolve, cryptographic hash functions remain at the forefront of safeguarding sensitive information.

Bibliography

[1] FIPS PUB 180-4. Secure hash standard (shs), 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak sha-3 submission to nist (round 3), 2009. https://eprint.iacr.org/2009/200.pdf.

[3] Vitalik Buterin. Sharding faqs. https://ethereum.org/en/roadmap/danksharding/#what-is-sharding, 2016. Ethereum Foundation.

[4] Vitalik Buterin and Virgil Griffith. Casper, the friendly finality gadget, 2017. https://arxiv.org/pdf/1710.09437.

[5] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains, 2016. https://link.springer.com/chapter/10.1007/978-3-662-53357-4_8.

[6] Alex de Vries. Bitcoin's growing energy problem, 2018. https://www.sciencedirect.com/science/article/pii/S2542435118301776.

[7] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network, 2013. https://ieeexplore.ieee.org/document/6688704.

[8] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail, 1992. https://doi.org/10.1007/3-540-48071-4_10.

[9] Eduonix. Know how hash functions works with blockchain and cryptocurrency. YouTube video, 2024. Available at https://www.youtube.com/watch?v=jvPfuh-HYik.

[10] Ethereum Foundation. Proof-of-stake vs proof-of-work. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/pos-vs-pow/, 2023. Accessed: 2024-11-30.

[11] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications, 2015. https://eprint.iacr.org/2014/765.pdf.

[12] Praveen Gauravaram. On the collision resistance of md5 and sha-1, 2007. https://eprint.iacr.org/2007/155.pdf.

[13] GeeksforGeeks. Blockchain structure. https://www.geeksforgeeks.org/blockchain-structure/, 2023. Accessed: 2024-11-30.

[14] GeeksforGeeks. How does the blockchain work? https://www.geeksforgeeks.org/how-does-the-blockchain-work/, 2023. Accessed: 2024-11-30.

[15] IBM. What are the benefits of blockchain? https://www.ibm.com/topics/benefits-of-blockchain, 2023. Accessed: 2024-11-30.

[16] Donald E. Knuth. The art of computer programming, volume 3: Sorting and searching, 1997. https://dl.acm.org/doi/book/10.5555/265594.

[17] Alfred Menezes, Scott Vanstone, and Paul Oorschot. Handbook of applied cryptography, 1996. https://cacr.uwaterloo.ca/hac/.

[18] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.

[19] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies: A comprehensive introduction, 2016. https://press.princeton.edu/books/hardcover/9780691171692/bitcoin-and-cryptocurrency-technologies.

[20] Gareth W Peters and Efstathios Panayi. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money, 2016. https://arxiv.org/pdf/1511.05740.

[21] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. `https://cdn.nakamotoinstitute.org/docs/lightning-network.pdf`.

[22] Sara Saberi, Mahtab Kouhizadeh, Joseph Sarkis, and Le Shen. Blockchain technology and its relationships to sustainable supply chain management, 2019. `https://www.tandfonline.com/doi/full/10.1080/00207543.2018.1533261`.

[23] Fahad Saleh. Blockchain without waste: Proof-of-stake, 2021. `https://www.researchgate.net/publication/325891130_Blockchain_Without_Waste_Proof-of-Stake`.

[24] John Smith and Jane Doe. Deciphering the blockchain: A comprehensive analysis of bitcoin's proof-of-work mechanism, 2023. `https://arxiv.org/pdf/2304.02655`.