Imperial College London
Faculty of Engineering
Department of Computing

Equitable Proof-of-Work Mining Rewards

Ramy Abdelmageed Ebrahim Khalil

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London
September 2022

Abstract

We present Reward-All Nakamoto-Consensus (Reward-All), a Proof-of-Work cryptocurrency that rewards each miner with a number of coins that is directly proportional to its individual mining power, rather than in proportion to its relative share of the entire network's mining power as done in Bitcoin. Unlike their Bitcoin counterparts, miners in Reward-All do not have to win the leader-election process to earn coins, and only lose earned coins after block reorganizations of a configurable minimum length occur. We present a detailed specification of Reward-All, along with a prototype implementation, and an evaluation of its practicality and efficiency. Additionally, we provide an analysis of the security of Reward-All, where mining is modeled as a Markov Decision Process, and the advantages of optimal mining strategies are quantified. Under reasonable configurations, Reward-All achieves near-perfect incentive compatibility, and near-zero censorship susceptibility, for adversarial mining shares up to 45%, while retaining the same chain quality as Bitcoin's Nakamoto Consensus (Nakamoto). However, Reward-All pays for these advantages with a regression in subversion gain resilience compared to Nakamoto. Furthermore, under Reward-All's approach, the growth rate of the total coin supply correlates closely with the growth rate of mining power invested in the network. This enables miners to mint coins at a stable hash-based cost of production, and enables all rewarded coins to correspond to an approximately equal number of hashing attempts on average. Consequently, depending on the network transaction-fees, Reward-All improves miners' waiting times for rewards, and incentivizes forming mining pools smaller than required in Bitcoin for an equal level of reward stability. Moreover, rewards in Reward-All exhibit significantly lower variance for non-majority miners compared to Nakamoto, enabling unprecedented reward stability.

Dedications

This thesis is dedicated to my mother – Mervat, to my wife – Sherine, and to my friends, all without whom I would not have been able to complete this work.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Naranker Dulay, for enabling me to carry on researching a subject I am profoundly passionate about, and for guiding me throughout my PhD. I am also thankful to Dr. Amani El-Kholy, to Prof. William Knottenbelt, and to the other members of the Department of Computing who supported me during my time at Imperial College London. This work would not have been made possible without the generous funding and support of the Imperial College London President's PhD Scholarship program.

Declarations

I, Rami Khalil, hereby confirm the following: I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisors. I have not knowingly committed any plagiarism against the works of others. I have documented all methods, data and processes truthfully to the best of my knowledge. I have not intentionally manipulated or misrepresented any data. I have appropriately mentioned to the best of my judgement any and all persons who were significant facilitators of or contributors to this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis proposes an alternative to Bitcoin's reward and punishment directives by presenting the Reward-All Nakamoto Consensus (Reward-All) protocol. Reward-All casts light on and remedies many of the elusive drawbacks of Bitcoin that restrict the equitable compensation of miners, and hold back its coinage from attaining a properly measurable cost-of-production that can be used to determine its purchasing power.

Bitcoin is by far the world's leading cryptocurrency in user adoption and market capitalization, which exceeded over one trillion USD in the first quarter of 2021. At Bitcoin's core lies Nakamoto Consensus (Nakamoto), an effective consensus algorithm based on Proof-of-Work (PoW) and a cryptographically authenticated data-structure. For over a decade, Nakamoto has successfully motivated thousands of Bitcoin miners to maintain consensus on the contents of Bitcoin's ledger, where over 42 million account addresses and over 720 million transactions have been recorded so far [Blo].

The crux of Nakamoto's incentive mechanism is that miners who invest their computational resources to maintain the ledger are issued rewards in the form of cryptocurrency units, or coins, as compensation for their efforts towards advancing the state of the ledger [N$^+$08]. While Bitcoin attests to the resilience and capability of PoW-based consensus in running a permissionless distributed ledger, we show in this thesis that it still suffers from limitations that affect the ledger's ability to support a usable currency that can be adopted by a significant population of individuals.

The limitations we focus on can be summarized in three main points. Namely, only a single miner is issued rewards every time a block of transactions is executed by the network, while all other miners are left uncompensated for their work. Furthermore, the amount of compensation is not adjusted based on the number of participants, which leads the expected reward of each miner to decrease as more miners participate. Lastly, stale blocks, which occur naturally in the protocol, can cause miners to lose their rewards even when all miners are compliant, and when such blocks can be created intentionally by an adversary, they can lead to dangerous attacks. We elaborate further on the consequences of these limitations.

1.1 Winner-Takes-All Dynamics

To appreciate the need for an alternative coin creation procedure for PoW ledgers, consider the current state of the mining process in Nakamoto Consensus. Nakamoto incentivizes network participants, called miners, to take part in a periodic leader-election process. In each leader-election round, the miner who first finds a satisfactory PoW is considered as the elected leader, and can publish a sequence of transactions, in the form of a block, for the entire network to execute and confirm. For this block to be considered valid, it must be constructed such that it extends all previously published valid blocks so far, as to form an authenticated append-only ledger, commonly referred to as a Blockchain.

Remarkably, taking on the role of a miner and abandoning it when needed is done in a permissionless peer-to-peer fashion. Once connected to any peer in the network and synchronized with the latest block, any new miner can begin to propose blocks through solving PoW puzzles and can cease to do so, in both cases without the need to notify other miners in the network, or any trusted third party (TTP). Despite its permissionless design, Nakamoto effectively prevents Sybil attackers from overcrowding the leader-election process through tying participation in consensus to solving computationally expensive PoW puzzles. A miner's representation in this leader election process is thus capped by the miner's computational resources, or by how many times the miner can attempt to solve the PoW puzzle per round.

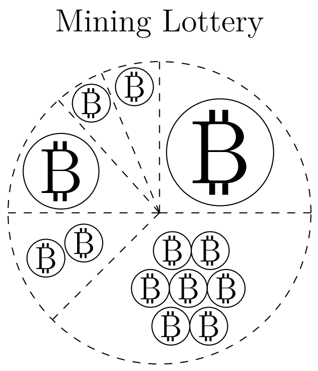


Figure 1.1: Visualization of the winning proportions of the nodes of different block mining capabilities in a figurative Bitcoin network. Each node's size depicts its relative computational power in the network. Nodes with a larger relative share of the network's computational power receive rewards much more frequently than those with smaller shares. This creates an incentive for nodes with relatively small shares to consolidate their computational powers into a mining pool, as they would receive rewards more frequently. Such mining pool are depicted by nodes grouped together in the same pie chart section.

This induces a lottery each round, an example of which is shown in Figure 1.1, where the probability of winning is proportional to investment. However, for a miner with relatively small resources, the expected number of rounds lost before winning can be substantial, and may very well exceed the lifetime of the living beneficial owner of the miner due to this winner-takes-all approach employed in Nakamoto. One may be fine with such an arrangement if deciding the next sequence of transactions to be executed by the network is not of concern, but what about remuneration for the computational resources that a miner expends out of pocket? What would incentivize any individual to burn away their computational resources with no hope of being paid for doing so in their lifetime? And is the size of the reward proportional to the resources invested in the mining process?

Fundamentally, only the elected leader of a round who successfully publishes a block receives a reward, while all other participants receive nothing for that round at all. Consequently, coalitions of miners, called mining pools, form to alleviate the impoverishment of independent mining with relatively small resources [Ros11]. Miners who take part in such pools receive a share of the rewards won by the entire pool, with this share being proportional to the miner's relative contribution to the pool's mining power. Joining such pools offers a more continuous reward stream, but at the cost of increased centralization of mining power.

1.2 Unsustainable Compensation

Surprisingly, the number of coins issued to the winning leader in a round does not correlate with how much total mining power is invested by the network towards mining at the time [CPR19]. This means that for a miner to receive a non-decreasing expected return on investment in cryptocurrency units, it must grow its computational resources by at least as much as the remainder of the network is growing its resources, or risk being gradually phased out and receiving a smaller return in coins over time [KLK+19]. Additionally, in Bitcoin, the number of coins issued as a reward for mining a block is split in half every $210,000$ blocks, making smaller mining rewards inevitable, and artificial scarcity of coins imminent. This, at the very least, is the tip of a colossal iceberg.

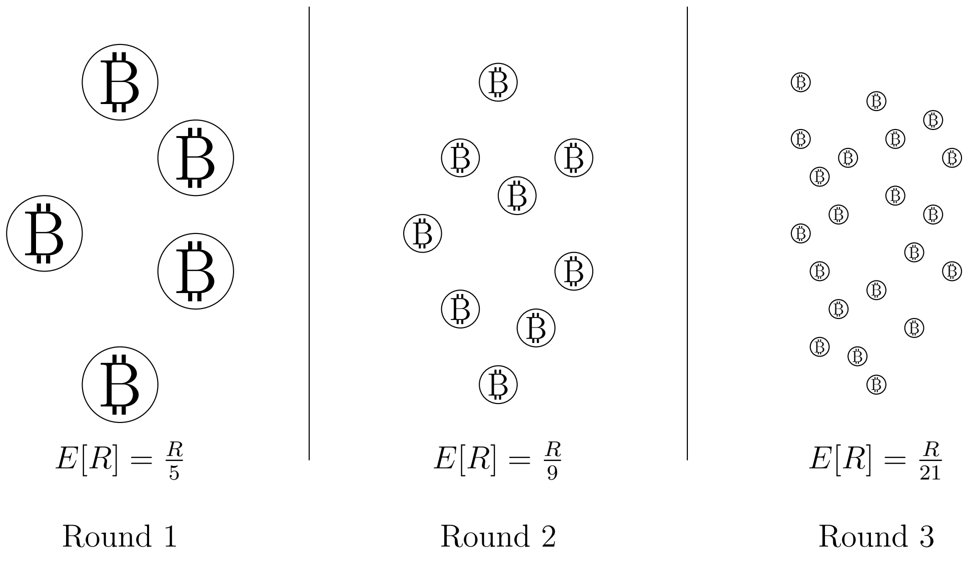


Figure 1.2: Three different mining rounds where all miners have equal computational resources invested. The number of miners in rounds 1, 2 and 3 are 5, 9 and 21 respectively. In each round, R coins are issued to the winning miner. As more miners join to compete for the same set of R reward coins, the expected reward, $E[R]$, of each miner decreases.

For the profitability of mining to be sustainable, under this disproportionality between total rewards and total mining power, demands that mining costs have to continuously decrease, transaction fees have to continuously rise, or the value of Bitcoin itself has to keep increasing.

Ultimately, this fuels a competitive coin creation process, visualized in Figure 1.2 that does not thrive under the simultaneous stability of all of the aforementioned processes, which are all desirable to stabilize for the system to support a practical currency! Surprisingly so, even as mining power increases, this artificial scarcity leads to a discrepancy in miner compensation that indirectly rewards owners of coins created earlier than others. With these concerns in mind, increased adoption of Bitcoin or any other system built on the same mechanism may very well be unsustainable.

1.3 Reward Instability

So-called stale blocks [VG17] can occur as a result of temporary disagreements in the network on the state of the ledger [NH19]. In such cases, more than one miner each finds a satisfactory solution to the PoW puzzle of the current leader election round within a short amount of time, which results in several valid proposals for the latest block. Ultimately, Nakamoto allows all miners to reach consensus on which block to adopt, causing the other blocks proposed at that round to be discarded, becoming stale [GKL15]. However, the miners of such stale blocks, and any blocks created on top of them, lose the rewards they would have otherwise been issued, as shown in Figure 1.3, whether the miners were innocently mistaken in creating such blocks or maliciously intent on inducing failures in consensus.

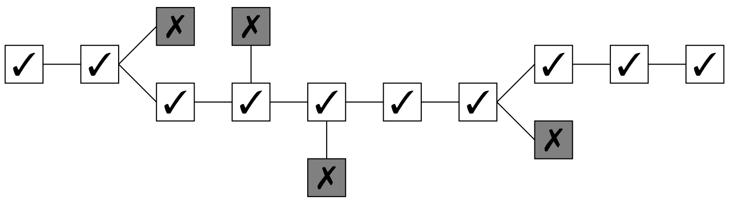


Figure 1.3: Example mining scenario with stale blocks. *Squares* represent mined blocks. A gray interior block shade means that the block is stale. No interior block shade means that the block is part of the main chain. The sign (✓) means that the creation of the containing block is rewarded. The sign (✗) means that the creation of the containing block is not rewarded.

Even worse, adversarial miners, whether as pool members [Eya15, KKS$^+$17] or independently, with a non-negligible share of the network hashing power may even abuse stale blocks to increase their relative rewards per block, harm the revenue of compliant miners, or double-spend coins [KAC12, BMC+15, NKMS16, SSZ16, ZP19, MJP+20]. This ultimately leads to protocol-compliant miners becoming vulnerable to incurring losses in mining rewards because of miners who follow adversarial mining strategies.

These downsides and weaknesses of mining rewards in Nakamoto are detrimental impediments towards equitable coin acquisition. Without an equitable means of distributing their mining rewards, Proof-of-Work permissionless distributed ledgers may never realize the full utility of a currency. This thesis aims to aid that realization by constructing a more equitable mechanism for rewarding Proof-of-Work miners.

1.4 Objectives

The primary goal of this thesis is to construct a Proof-of-Work protocol which remedies, or fixes, the aforementioned shortcomings of Nakamoto. To achieve this goal in a measured and organized manner, the effectiveness of the constructed protocol against the aforementioned downsides must be clearly demonstrated, and the constructed protocol must at least satisfy the following thesis objectives:

1. Establish a fixed mining cost per coin. Ensuring that the protocol rewards miners directly in proportion to their individual mining powers and does not induce any artificial scarcity in the supply of the coin allowing it to establish a relatively stable cost of production in terms of the number of hashing computations required to create a new coin.

2. Minimize value leakage from mining rewards to existing coin holders. Safeguarding miners from any undeserved deduction or retraction of mining rewards, which increases the value of the existing coin supply, ensures that rewards are only sent where due, and hidden mechanisms which draw value from newly mined coins and indirectly feed it to existing stakeholders are minimized.

3. Normalize reward variance across mining powers. Reducing the gap in reward stability between miners with nominal and significant shares of the mining power is pertinent to preventing smaller miners from suffering a disadvantage, and minimizing the incentives for forming large central mining pools.

4. Retain the mutual peer distrust of Nakamoto. The decentralized permissionless ledger setting implies that no single party in the system may be trusted to behave correctly, and consequently, all objectives must be satisfied without such an assumption. However, a majority of the population of miners may be assumed to comply with the protocol as in Nakamoto.

5. Minimize the impact of stale blocks on miner rewards. Fortifying the protocol against the effects of stale blocks protects it against adversarial miners that aim to detract from the number of coins earned by protocol-compliant miners, or degrade the ability of the network to process transactions. Intentionally introducing stale blocks threatens both the production cost stability and circulation speed of the protocol coin.

1.5 Contributions

Primarily, to improve the security of acquiring cryptocurrency coins through mining against leakage of value and manipulation, we design and specify in detail Reward-All Nakamoto-Consensus (Reward-All), our novel variant of Nakamoto which enforces that:

A. Miner rewards are based on their individual PoW solving throughputs. We designed Reward-All to reward miners with a number of coins that only depend on their absolute mining power, regardless of the total amount of mining power in the network. This allows us to establish a fixed hash-based cost per coin that is independent from block mining difficulty, satisfying our first objective. Consequently, Reward-All does not directly disadvantage participants which mine at a relatively later stage of the protocol's lifetime than other miners, or those which mine with only a small amount of hashing power, as per our second and third objectives. To demonstrate the effectiveness of our approach, we implement and evaluate a functioning prototype of Reward-All, and quantify the overheads associated with redeeming mining rewards in our system under different ledger conditions. Our prototype enables two or more mutually distrusting peers to operate a Reward-All payment blockchain.

B. Receiving mining rewards is not conditional on being elected leader. Our Reward-All protocol directly extends the Proof-of-Work mining process from Nakamoto, while allowing participants to claim reward coins only after having securely proven the amount of mining work they have performed. Consequently, we retain the well-studied characteristics of Nakamoto block creation, while introducing no new trust assumptions between peers to enable our Reward-All approach, as per our fourth objective. Furthermore, this allows us to compensate Reward-All miners even when they attempt to extend stale block chains, limited to a preconfigured chain length. Consequently, we achieve our fifth objective, and quantify the resulting resilience our Reward-All design achieves against adversarial miners by performing a security analysis of mining in Reward-All, where we model mining as a Markov-Decision-Process (MDP) with the different adversarial proof-of-work mining objectives compiled in the framework of Zhang et al. [ZP19]. Our analyses yield highly favorable results that demonstrate Reward-All enjoys stronger incentive compatibility and censorship resilience than all current state-of-the-art protocols, while meeting the state-of-the-art in chain quality, and falling behind in double-spending resilience under certain conditions.

1.6 Publications

Parts of this thesis are submitted or published, in part or in whole, in the following papers:

1.6.1 PoSH in Practice

- Khalil, Rami, and Naranker Dulay. 2021. "PoSH in Practice: Implementing Proof of Staked Hardware Consensus with Limited Storage." The 3rd IEEE International Conference on Blockchain and Cryptocurrency (ICBC).

This paper introduces the cornerstone concept of operating a permissionless distributed ledger that rewards miners in proportion to their absolute mining power, and focuses on the technical challenges associated with storing the data required by the miners throughout the mining process. Since its debut, we have introduced methods which require even less storage and improve performance. This paper is motivated by Chapter 6.

1.6.2 Adaptive layer-two dispute cutoffs in smart-contract blockchains

- Khalil, Rami, and Naranker Dulay. 2021. “Adaptive layer-two dispute cutoffs in smart-contract blockchains.” 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). **(Best Paper Award)**
- Khalil, R.A. and Dulay, N., 2022. AC/DC: Adaptive Cutoffs and Disputable Cutoffs for robust critical transactions in smart-contracts. IEEE Transactions on Network and Service Management. **(Extended Journal Publication)**

This paper introduces the necessary functionality a layer-one blockchain must provide for so-called layer-two solutions to remain secure during times of high transaction fees. This paper borrows from the blockchain structure presented in Chapter 6 and Appendix A.

1.6.3 Towards Equity in Proof-of-Work Mining Rewards

- Khalil, Rami, and Naranker Dulay. 2022. “Towards Equity in Proof-of-Work Mining Rewards.” The 3rd International Conference on Mathematical Research for Blockchain Economy (MARBLE). **(Best Paper Award)**

This paper presents our framework for assessing reward distribution equity, where we introduce our key definitions that allow us to measure the per-hash coin reward value of a blockchain over time, and set the key constraints necessary for equitable reward distribution. We focus Chapter 3 on this paper.

1.6.4 Reward-All Nakamoto Consensus

- Rami A. Khalil and Naranker Dulay. 2022. “RANC: Reward-All Nakamoto Consensus.” The 37th ACM/SIGAPP Symposium on Applied Computing (SAC).

This paper presents the Reward-All protocol specification and demonstrating its efficacy in protecting against stale block attacks and reducing waiting times for rewards. Much of the content in this thesis can be found in this paper, albeit in an abridged manner for publication purposes, and without any dedicated explanation of Reward-All’s implementation. We draw on the contents of Chapters 4, 5, and 6 for this paper.

1.7 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 presents reviews background literature and related work.
- Chapter 3 identifies the sources of inequity in Nakamoto Consensus mining rewards and introduces Reward-All’s solution for equitable mining.
- Chapter 4 presents the full technical specification of Reward-All, detailing its design methodology and architecture.
- Chapter 5 presents an analysis of Reward-All’s security against block withholding attacks.
- Chapter 6 presents the details of our Reward-All prototype implementation and its evaluation in terms of the costs and overheads associated with mining rewards.
- Chapter 7 discusses several noteworthy aspects regarding Reward-All.
- Chapter 8 concludes this thesis, and outlines directions for future work.

Chapter 2

Background

Bitcoin is a complex distributed system with several layers. Its base is a **ledger layer**, which is a collection of cryptographically endorsed data designed to fully describe the state of ownership of a set of coins. Above that layer lies a **consensus layer**, which allows a distributed network of mutually-distrusting peers to agree on the contents of the ledger and update it according to a predefined set of rules. Further up is an **incentivization layer**, which motivates participants to participate in the consensus protocol and fully replicate the ledger, contributing to the security of updating its contents. These layers are designed to coalesce to form "*a purely peer-to-peer version of electronic cash*", where no central party, such as a Bank, is required to maintain the ledger, or to govern the supply of currency being managed [N$^+$08].

We begin this chapter in Section 2.1 with a review of the existing data schemes of fully replicated distributed ledgers. Then, in Section 2.2 we review the state-of-the-art consensus protocols designed to keep permissionless distributed networks synchronized on the contents of their ledgers. Subsequently, in Section 2.3, we examine existing coin reward mechanisms that are built above the reviewed consensus protocols in order to motivate participants to comply with the protocol rules. Lastly, we conclude this chapter in Section 2.4 with a summary of the presented literature.

2.1 Ledger Design

At the base of the permissionless distributed ledgers lies a data scheme for storing and retrieving the information required to maintain a ledger that is replicated across multiple nodes. The different data schemes used throughout various permissionless ledgers are all designed to satisfy one or more of the design goals discussed in this section. We discuss and describe each goal from the perspective of a single node, or replica, in the distributed ledger.

2.1.1 Coins, Addresses, Transactions

A central goal for permissionless distributed ledgers is to fully maintain the history of coin creation and expenditure. This goal is relevant for a **cryptocurrency** in which records are maintained that describe the state of all coins recorded in the ledger. Ultimately, the ledger details where each coin lies. How coins have moved so far, where each movement, or update to coin locations, is the result of a **transaction**. Notably, coin locations are denoted by hexadecimal strings called **addresses**.

Because coins are deemed **fungible**, i.e. interchangeable or equivalent, batches of coins are indexed by a single address, such that the ledger specifies how many coins are located at each hexadecimal address, instead of redundantly specifying where each individual coin lies. These addresses have largely been cryptographic commitments to information that defines the constraints which must be met for coins to leave the addresses. Such addresses have largely been of two main types.

Single-use addresses. Most prominently known as transaction-output addresses, these types of addresses are used in Bitcoin, where each address is a commitment to an executable program. For coins to be moved from an address, a transaction must specify input data to and reveal the full code of the program the address commits to. Using the input, the program must be successfully executed for coins to be moved [ABLZ18].

Transactions must specify each input address they move coins from, and each amount and output address they send coins to, as we illustrate in Figure 2.1. Interestingly, output addresses are not manually specified in a transaction. Instead, output addresses are derived from the commitment of the entire transaction. Only the contents of, or a commitment to, the script of the output address are specified in the transaction, and are used to determine how this output can later be spent.

Once a transaction is inserted into the ledger, all its input addresses are marked as spent, such that no further transactions can move coins from its input addresses. Atzei et al. [ABLZ18] create and analyze a formal abstract model of Bitcoin transactions, and prove that the semantics of the system forbid coins from being located in more than one address simultaneously.

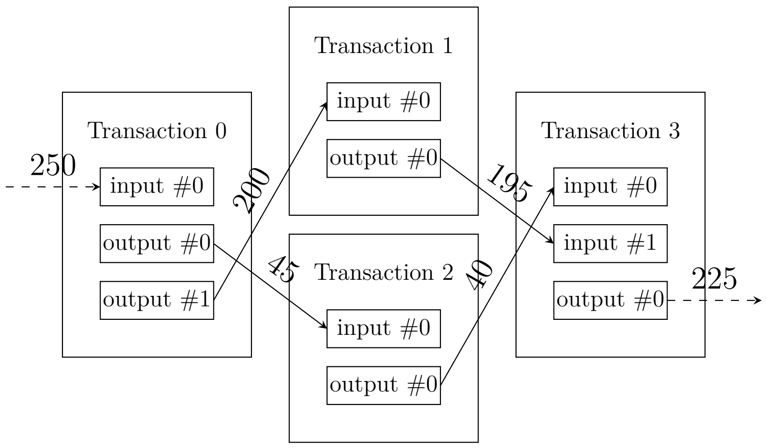


Figure 2.1: Single-use transaction output address example. The input of Transction 0 contains 250 coins, which it uses to create two outputs of 45, and 200 coins respectively. The 5 coins difference are unused, and can be collected as fees for including Transaction 0 in the ledger. Transaction 1 uses the second output of Transaction 0 to create a single output of 195 coins, similarly leaving 5 coins as fees. Similarly, Transaction 2 uses the first output created by Transaction 0 to create one output of 40 coins, leaving another 5 coins as a fee. Lastly, Transaction 3 uses the outputs of Transactions 1 and 2 to create a single output of 225 coins, leaving 10 coins as fees.

Remark. This addressing scheme is useful when providing highly restricted transactions which fully determine the changes they make in the ledger. However, since each transaction must fully specify the inputs it is spending coins from, when two or more transactions attempt to spend coins from the same address, only one transaction can be executed, while the other transaction will become invalidated. This causes a race condition problem for applications such as shared addresses with multiple users.

Multi-use addresses. Multi-use addresses, also known as account addresses, were introduced in Ethereum [W+14]. Such addresses either denote a commitment to the public key of an identity, or a program that is stored in full in the ledger. In the former case, the address is referred to as a wallet address, and in the latter case, it is referred to as a contract address.

Transactions that move coins from a wallet address must specify a single destination and the amount that will be transferred to it. The untransferred coins, and any future incoming coins, remain in the same wallet address [W+14], and can be spent by future transactions signed by the underlying private key of the wallet.

When specified as the destination of a transaction, a contract address may also utilize any additional input data specified in the transaction for the execution of its program. Moreover, a contract's program can invoke a cascade of transactions to other addresses to be executed as part of the transaction in the block, and specify an input for each destination if it is a contract address. However, a transaction may only stem from a wallet address, as we illustrate in Figure 2.2.

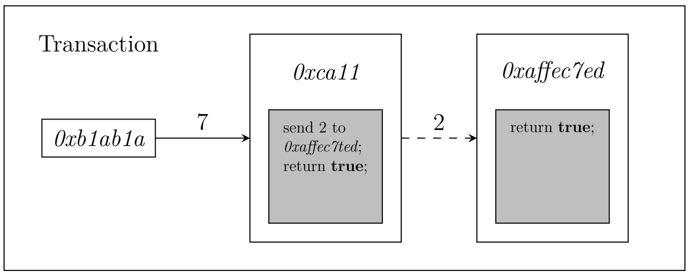


Figure 2.2: Multi-use transaction example. A single transaction from wallet address *0xb1ab1a* explicitly invokes contract address *0xca11* while sending it 7 coins. Because of *0xca11*'s program code, this results in the implicit invocation of contract *0xaffec7ed* with 2 coins within the same transaction. While only the transfer of 7 coins from *0xb1ab1a* to *0xca11* was explicitly mentioned in the transaction, 2 coins were added to *0xaffec7ed*'s balance.

These rules mean that a single transaction may result in the execution of several programs across several contract addresses in a ledger, leading to potentially much more complex changes in coin locations than those possible with Bitcoin addresses [CCM+20]. Notably, the transaction's execution must be successful, as in single-use addresses, for any of the movements it causes directly or indirectly to be permanently stored in the ledger.

Remark. While this scheme does not suffer from the same race condition problem as Single-use addresses, it prevents parallel transaction execution in the ledger, as the sequential order in which transactions are executed determines the final state of the ledger.

2.1.2 Transaction Termination

Another key property in permissionless distributed ledgers is that transactions may only take measurable amounts of computation and memory to be interpreted and fully executed, an aspect that is crucial to maintaining predictable read-write ledger throughputs.

In Bitcoin, transactions are specified in a stack-based scripting language designed to keep transaction processing overheads minimal [ABLZ18]. All language instructions are designed to take a negligible number of CPU cycles, such that the total number of instructions can be used as an estimate of how much computational power is required to execute a transaction.

On the other hand, in the Ethereum Virtual Machine [W^{+}14] (EVM), the computational and storage resources required to process a transaction are characterized by its **gas** consumption, where gas is a unit designed to capture the total cost of execution of a transaction. This more elaborate measurement scheme handles the added complexities of the EVM, which enables a much more feature-rich transaction language under a multi-use address scheme.

In the EVM, a transaction that is very small in byte-size may take a considerable amount of CPU time to execute if it invokes a computationally intensive program stored at a contract address. Consequently, EVM transactions specify explicit limits of how much gas each may consume, and each EVM instruction has a predefined gas cost to execute. As a transaction is executed, the amount of gas its instructions have tallied so far is counted, and if this amount exceeds the transaction's limit, the entire transaction is aborted and its coin movements are not saved in the ledger. Ensuring that the gas unit cost of instructions adequately corresponds with the number of CPU cycles required to execute them is essential to preventing denial-of-service attacks [PL22].

Transactions incur a cost in currency units to be executed. In Bitcoin, this cost is pre-defined in full prior to transaction execution, while in Ethereum, the total cost is calculated based on the total gas consumed after execution. In both systems, sufficient currency units must be spendable in a transaction to pay its transaction fee. This means that the total units available in the Input addresses of a Bitcoin transaction must be sufficient to cover those allocated to the output addresses, in addition to the transaction fee, as shown in Figure 2.1. Similarly, the units available in an Ethereum wallet address used to initiate a transaction must be enough to pay for any transferred currency units and also for the total gas incurred in the transaction.

However, because the total gas consumed by an Ethereum transaction cannot always be known in advance, the transaction must specify a gas price in currency units, which is multiplied by the total gas consumed by the end of a transaction to derive its total due fee. To ensure that the final fee is payable, the maximum fee that can be due in an Ethereum transaction is always known in advance, because the transaction must specify the maximum gas that it can consume, and the balance of the transaction initiator must cover said maximum fee [W$^+$14]. Transactions which exceed their specified gas limit are marked as failed, which leads their sending wallet to be debited for the incurred fees without moving any coins to the destination address or executing any code in its contract.

Remark. Despite relying on transaction pricing mechanics, current ledger designs do not provide mechanisms for transactions to access historical data associated with transaction fees. Because such data can be used to measure the demand placed on the ledger, it can be used to improve the security of applications that depend on timely access to the ledger [KD21].

2.1.3 Append-only Updates

Lastly, another fundamental record-keeping objective in permissionless distributed ledgers is that once information is inserted into the ledger it should no longer be removed or changed. Instead, an append-only ledger which contains every incremental update is kept. Namely, in a blockchain, batches of transactions are recorded in the ledger as blocks. When inserted into the blockchain, each block is appended to another preceding block, denoting that the sequence of transactions in the appended block logically follows, or is executed after, the sequence of transactions in the preceding block[1]. In this model, the blockchain is only a subset of a directed tree of blocks, where different branches denote divergent versions of the ledger [N+08].

Notably, each block in this design contains a cryptographic commitment to the block it is appended to, as shown in Figure 2.3, which provides a means of **tamper evidence**[2]. In this tamper evidence model, using the commitment to one block in the stored tree of blocks as a reference, one can examine the data of any preceding block, even if it is retrieved from an external party, and verify that this data was not modified. This verification is done through using all blocks starting from the preceding block, and leading up to the block of the reference commitment, to recompute the reference commitment [N+08]. If the recomputed commitment matches the reference commitment, then one can be certain that, with overwhelming probability, the preceding block was not modified since the block of the reference commitment was appended.

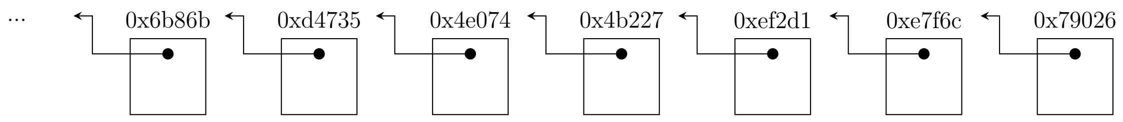


Figure 2.3: Example blockchain diagram, where the chain proceeds from left to right. Each block is represented by a square. The hash of each block is written directly above it. Each block commits to the entire chain preceding it by including the hash of its immediate parent.

A core functional feature for this append-only approach is that it enables the ledger to be safely rolled-back in time, and enables the maintenance of several consistent views of the ledger simultaneously. The matter of which consistent view constitutes the canonical view is the core concern of the consensus mechanism, and deciding which view should be adopted as the main one is essential to ensuring that the ledger can process transactions quickly [SZ15]. Consequently, the blockchain structure must support flexible switching between different views of the ledger, so that once consensus on the canonical view is reached, it is possible

[1]More general append-only logs exist where one block may be appended to more than one preceding block, forming a directed acyclic graph called a **blockDAG** [LSZ15].

[2]This is different from tamper proofness, which is an objective of the consensus mechanism.

to efficiently adopt said canonical view.

Remark. While several work has been dedicated to providing efficient tamper-evidence proofs [KLS16, BKLZ20, KMZ20] using current blockchain commitments which refer only to the preceding block, little attention has been given to constructing block designs which utilize commitments that can offer more efficient tamper-evidence proofs. For example, using Merkle-Tree commitments rather than hash chains, which would allow efficient tamper-evidence proofs to be generated for older blocks given newer reference blocks.

2.2 Blockchain Consensus

Consensus on the canonical chain is a fundamental functionality to achieve in a permissionless distributed blockchain ledger. The primary aim behind establishing a canonical view of the ledger's contents is to prevent the act of **double-spending**, which would occur when multiple transactions move the same set of coins from the same address to different destinations [N^+08], as illustrated in Figure 2.4.

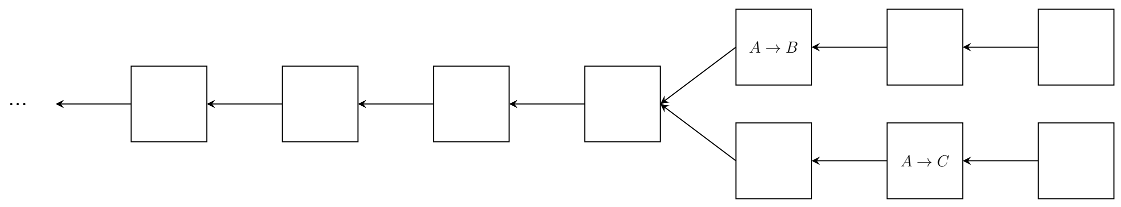


Figure 2.4: Example conflicting blockchain forks. Each block is represented by a square, with an arrow pointing to its parent in the directed acyclic graph of blocks. The two forks contain conflicting information about A's coins, whereby in the upper fork, A sends all its coins to B, while in the lower fork, A sends all its coins to C.

While no valid single chain in the tree of blocks maintained in the ledger can contain such double-spending transactions, divergent branches in the tree, or different chains, can contain transactions which move the same coins from the same source addresses, but to different destinations in each branch. Consequently, if peers disagree on which chain is the canonical one, the originators of double-spending transactions can spend their coins more than once, which is impractical for the recipients as they obtain no guarantee of being considered as the sole owners of their coins by all nodes.

Generally, consensus enables a distributed system of independent nodes to function almost as if it were one cohesive node. Several consensus protocols can achieve such unity under a multitude of different conditions. e.g. only under the strong assumption that all nodes fully comply with the protocol, or that all nodes are known in advance and identifiable. Furthermore, this unison may be achieved by partitioning data across nodes, such that, for example, some nodes contain one half of the ledger, while others are only required to retain the other half. Several other assumptions can be placed by the consensus mechanism on the structure of the distributed system and how communication between its nodes takes place.

Notably, a central assumption for permissionless distributed ledger protocols is that any participant may enter or exit the system at any point in time without a fixed predetermined identity or role, and a non-majority of nodes may deviate from the protocol arbitrarily.

2.2.1 Network Topologies

In this section, we review mechanisms that allow nodes to provide their ledger contents to interested peers in the system, allowing a permissionless distributed ledger to reach consensus on a canonical view of a ledger.

Typically, these data distribution mechanisms are designed with consideration for a **peer-to-peer** network topology, as shown in Figure 2.5. In such a network structure, each node of the system is connected only to a relatively small, or limited, number of other random nodes. Ideally, no central node that must be connected to is assumed to exist, and nodes can connect to and disconnect from any of their peers freely.

The propagation of information in this layout is established via **gossip** protocols [DGH$^+$87], whereby once a node learns new information, such as a transaction or a block, it passes it on to its peers. Gossip protocols can employ information advertising and information pushing mechanisms. In the former case, the availability of information is what is first sent by a node to its peers, and the full information itself is only transmitted to peers that request it. In the later case, new information is immediately transferred by a node to its peers.

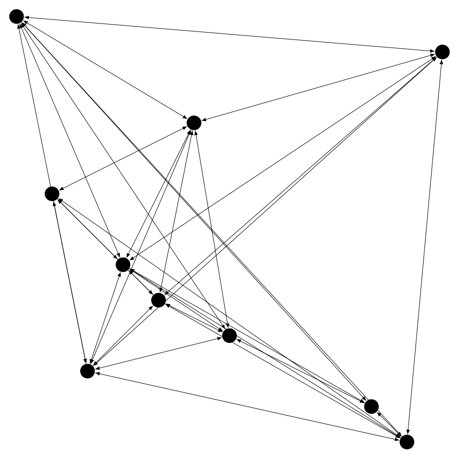


Figure 2.5: Example peer-to-peer overlay network. Each node is represented by a black dot, while bi-directional communication links are represented by double-arrow connections between nodes. The length of each link indicates connection latency.

Notably, as no nodes are assumed to retain fixed identities, each node keeps its own records of which peers are **well behaved**, i.e. which peers provide valid information and do not disconnect erratically. This localized reputation system enables each node to establish its own list of peers that are useful to be connected to. Furthermore, peers can exchange information about other peers, allowing nodes to explore the network and dynamically change their connections, as often done in Ethereum [KSL+21].

Interestingly, the disruption of connectivity and of information propagation in the network of a permissionless distributed ledger is crucial. Attackers can disrupt the entire consensus process altogether not only by isolating portions of the network from each other [CDG+02], but also by inducing severe delays in information propagation between nodes [WWK19].

Remark. Despite their importance towards establishing consensus, there are very few rules in place that force nodes to propagate newly learned information to other peers. As we discuss in Section 2.3, there are incentives in place to motivate a node to publish its own blocks to the network, but there are no guarantees that a node must provide any useful information to its peers. Furthermore, state-of-the-art consensus algorithms do not offer defenses against network-layer attackers [HKZG15, AZV17], such as requiring periodic proofs that network peers are mining on the canonical chain.

2.2.2 Longest-chain Protocols

Longest-chain protocols allow a node to use all information it has learned so far to take a local decision about which view constitutes the canonical ledger view. In this context, information is comprised of blocks, and the canonical view of the ledger is embodied by a single branch in the tree of blocks stored by a node. This branch essentially forms a chain of blocks, or blockchain, and is uniquely identifiable by the last block in the chain, referred to as the **tip**. Consequently, longest-chain protocols aim for all of its nodes to reach the same decision on which tip characterizes the canonical view.

Hinted at by its name, a longest-chain protocol typically aims to guide nodes compliant with the protocol specification towards a state whereby they regard the **longest** blockchain, in terms of its number of blocks, as the canonical chain. However, the length of a chain is not always the only factor taken into consideration. In addition, a **weight** may be attributed to each block according to the protocol, and consequently regarding the **heaviest** chain, in terms of effort needed to create it, as canonical becomes the target for consensus. Unsurprisingly, said protocols also take into consideration cases where two or more chains have the same total length, or weight, by introducing **tie-breaking** rules which lead to the selection of only one canonical tip [Hei14, ES14, SZ15, ZP17].

Notably, the scope of longest-chain protocols does not end with only a specification of a procedure to select a tip from a block tree, but also extends to specifying how nodes can participate in block creation, a process which must be regulated in pace so that the canonical tip does not frequently change across divergent branches [CDE+16].

Broadly speaking, longest-chain protocols regulate block creation by facilitating a periodic leader election process in regularly intervaled rounds, where the product of each round is usually intended to be a single block, or series of blocks by a single entity [EGSVR16], to be appended to the canonical chain. Many such protocols have been successfully designed and implemented to achieve different trade-offs.

Proof-of-Work (PoW). Nakamoto Consensus is characteristically the seminal PoW-based longest-chain protocol. In this design, a system inspired by Hashcash [B$^+$02] is adopted to throttle and regulate new block creation, such that a block can only be appended to another block if the hash of the appended block is less than or equal to a certain numeric value, which we refer to as the block target.

To create a block that meets the block target, a node must search for a nonce value, which when inserted into the block's data, causes the block's hash value to meet the block target. Finding such a nonce requires a brute-force search to be performed by the node, and nodes which undergo this search are commonly referred to as **miners**. Notably, the PoW approach is very computationally intensive, and consequently requires a significant amount of energy to be expended by the distributed system nodes.

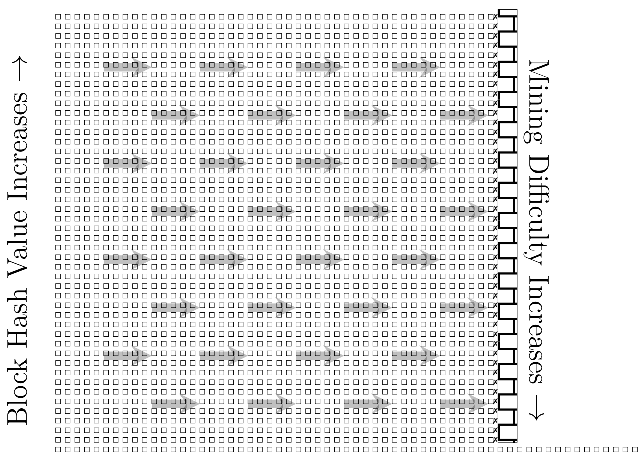


Figure 2.6: Proof-of-Work mining difficulty acts as a means to throttle block production to a stable average rate, such that each block is given adequate time to fully propagate across the overlay network. Only blocks whose hash value falls below the mining target value may pass. The mining target value is inversely proportional to the mining difficulty, leading fewer blocks to be accepted as mining difficulty increases.

The number of search attempts required to create a block whose hash meets the block target follows a geometric distribution, the expected value of which is referred to as the block mining difficulty. Consequently, the time between block arrivals in this process also follows an exponential distribution, but whose mean is based on the mining difficulty and the total search speed of all miners combined. In order to keep block production at a stable rate, which is currently approximately 1 block every 10 minutes in the Bitcoin network, the block target is adjusted based on the average amount of time taken to create the most recent N

blocks, where N is a predetermined constant [N+08]. However, the timestamps which indicate the time at which blocks were created are not precise. Furthermore, a noticeable reduction in the search power of the distributed system may take a significant amount of time to be reflected in the mining difficulty under this approach [IWSK21].

Furthermore, while originally intended to enable miners to have somewhat equal shares of the computational power of the network, high performance application-specific integrated circuit (ASIC) hardware was commercially developed for PoW [RD17]. This has lead to a severe imbalance in favor of miners that employ ASICs, compared to those that only utilize a general purpose CPU to search for a nonce. Nonetheless, families of so-called ASIC-resistant hashing functions, which cannot be cost-effectively directly implemented in hardware, are under constant research and development, with so-called Memory-bound PoW and Programmatic PoW retaining a foothold against hardware acceleration [KKKS20].

Remark. The intensive computational resource requirements of PoW lead to significant energy usage by its nodes [OM14, DV18, GKS20]. This reason has so far been the main motive for designing alternative schemes that demand less energy intensive resources than computation, namely storage space, or have only purely virtual resource demands.

Proof-of-Space (PoSpace). PoSpace protocols [ABFG14, PKF+18, PPA+15] operate in a similar fashion to PoW, but instead of only significantly demanding computational resources from a miner, they also require a miner to have access to a large amount of storage. In some protocols, the intensive computational requirements are dropped altogether in favor of demanding that storage be the dominant requirement for a miner to participate in consensus. Creating a block under such protocols requires that the block be accompanied by a PoSpace, which attests that the miner creating the block has exclusive access to some amount of storage.

Alternative forms of PoSpace are Proof-of-Retrievability (PoR), Proof-of-Space-Time (PoST), and Proof-of-Capacity (PoC). A PoR [SW08, BJO09] additionally attests that the space in question is used to store a certain file's contents, while a PoST [MO16] additionally attests that the contents of the storage space used to generate the proof have been kept unchanged for some amount of time. A PoC [HM19] on the other hand attests that the storage space is being used to store hard-to-compute hash-tables that require intensive computational resources to create.

Remark. While storage-based approaches require smaller energy demands than PoW, they do not guarantee that the storage resources are being exclusively used to mine on top of one chain. The Chia blockchain specification refers to this issue as **double-dipping**, and remedies the advantages which an adversary can gain from it by specifying that honest miners may also attempt to extend 3 chains concurrently [CP19]. Nonetheless, even when honest miners extend multiple chains at once, an attacker can completely dominate the blockchain with fewer resources than required in Proof-of-Work [DKT$^+$20].

Proof-of-Stake (PoS). PoS protocols [KN12, KRDO17] rely purely on each miner owning a **stake** in the distributed system to enable block production. A miner's stake ownership is embodied by a coin balance that is recorded in favor of the miner within the distributed ledger. Under this approach, and compared to PoW and PoSpace, miners do not perform any resource-intensive tasks, such as continuous hashing or storage maintenance, in order to create blocks. Instead, the state of the longest-chain of the ledger decides which miner, or committee of miners, can create the next block.

However, because creating a block in a PoS system is inexpensive, PoS protocols face unique challenges that their block creation processes must address [LABK17]. First, the **nothing-at-stake problem** occurs when a miner finds itself next-in-line to create a block on more than one divergent chain and seizes that opportunity. Second, a **grinding attack** occurs when said miner searches for a block which would update the ledger's state such that the same miner remains next in line to create a block. Third, a **long-range attack** occurs when miners can be bribed to reveal their private keys to an adversarial miner who can use them to rewrite a significant suffix of the canonical chain.

Remark. Despite their acknowledgement as blockchain consensus protocols, PoS designs fall short in their support for permissionless participation. Not only must new participants acquire coins in order to take part in the consensus lottery, but also existing participants are guaranteed to maintain the percentage they own of the coin supply if they consistently increase their stake using the rewards they earn. These issues can be problematic when a majority of the staked coin supply falls within the hands of a single party, which can then dominate the consensus process and even prevent new participants from partaking in consensus.

2.2.3 Stale Blocks

The objective of longest-chain protocols would be straightforward to achieve in a synchronous network setting where new blocks propagate to all nodes simultaneously, and the local procedure used by each node deterministically selects the same new tip after the block is appended. However, connectivity via today's internet does not provide such a perfectly synchronous and dependable means of communication, and any practical consensus protocol design must take into consideration delays in information propagation across the system.

For this reason, longest-chain protocols offer mechanisms for a node to accept blocks at much later times than their creation times, and to consider any branch these blocks form as the canonical chain if it indeed forms the longest, or heaviest, chain seen so far by the node. Consequently, blocks which become abandoned by a majority of protocol-compliant miners are referred to as **stale**. Stale blocks are an inevitable unwanted byproduct of all state-of-the-art longest-chain protocols, even when all miners in their systems are protocol-compliant.

Stale Block Attacks. However, and more importantly, intentionally forcing compliant miners to switch over to a new view of the canonical chain, and causing the divergent branch of the prior canonical chain to become stale, is a core adversarial behavior in longest-chain protocols [SSZ16]. An adversary in control of some miners in the distributed system can intentionally withhold newly found blocks by those miners from the remainder of the system, and subsequently release these blocks once they form a chain that is long enough to be adopted as the canonical chain by the compliant miners, while causing the branch that was previously considered canonical by the compliant miners to become stale.

Depending on the objective of the adversary, causing the blocks generated by compliant miners to go stale can have adverse consequences [ZP19]. Most notably, it can enable the adversary to execute double-spending transactions by spending its coins in the compliant miners' blocks, and then replacing those blocks with its own blocks that spend the same coins differently. Alternatively, and regardless of the reward mechanism employed, the adversary can slow down the overall transaction execution throughput of the distributed system by replacing transaction-filled blocks with its own empty blocks.

For an adversary to create a long enough chain that can cause the compliant miner's blocks to go stale, it essentially enters into a block creation race with the compliant miners. An adversary can only win this race when it is in possession of a divergent branch that can cause the compliant miners to adopt that branch as the canonical chain once they receive it. The probability of an adversary winning such a race depends both on the adversary's share of the resources being used to create blocks, and the speed with which blocks created by compliant miners can propagate to other nodes [ES14].

To understand the limits of longest-chain protocols against such attacks, Garay et al. [GKL15] analyze the Bitcoin protocol and isolate in it what they term as the Bitcoin backbone protocol. They demonstrate that the backbone protocol is able to maintain two central properties when compliant miners control a majority of the block creation resources and under relatively fast block propagation. The first property is **Persistence**, which states that, with overwhelming probability, a confirmed transaction will become permanently adopted by all compliant nodes after some number of blocks have been added to the chain after its confirmation. The second property is **Liveness**, which states that transactions by compliant miners will eventually be included in the canonical chain. Their work intentionally focuses on the block creation and propagation process, and excludes an analysis of the rationality of compliant miner behavior or the rewards offered by the block creation process.

Dembo et al. [DKT$^+$20] later present a more general analysis of longest-chain protocols which yields tighter security bounds in slower block propagation settings, and extends beyond PoW.

They demonstrate that the block withholding attack is the worst case attack for longest-chain protocols, including PoW, PoSpace and PoS variants. Most notably, they demonstrate that some PoSpace protocols are less resilient against stale block attacks than PoW and PoS protocols, mainly due to the ability of the examined PoSpace miners to use their storage resources to extend multiple branches simultaneously.

Remark. Many of the limitations of Proof-of-Work and other longest-chain protocols against adversaries that cause blocks to go stale have become clearer as research has progressed. Nonetheless, such protocols still remain as a viable option against deterring adversarial behavior, due to the scale of resources required.

Checkpointing Defenses. The leading mechanism in effectively preventing a node from considering a block to have gone stale under any circumstances is referred to as **checkpointing**. Under this approach, a block is effectively considered permanent once a sufficiently long chain of blocks has been created on top of it. Even if later presented with a longer, or heavier, chain than the one currently known by the node, the node would not adopt that newly presented chain as the canonical chain if the new chain causes any checkpointed blocks to become stale.

These checkpoints, which form exceptions to the longest-chain rule, can come in the form of hard-coded software upgrades to the nodes' software. The need for such checkpoints arose from the fact that while longest-chain protocols provide a stable form of consensus to each node, they offer no absolute guarantees on the permanence of a block in a node's canonical view if that node suddenly discovers a new network of nodes with a longer, or heavier, canonical chain.

In essence, the objectivity of longest-chain protocols in determining a canonical chain detracts from the practicality of considering that canonical chain as the only legitimate source of truth about the transactions that have occurred in the ledger, because said chain can always be replaced if not checkpointed.

One notable objective for checkpoints is to ensure that a node that is new to the network, and uses the same software, does not end up adopting a different canonical view from the node network. This objective is to be met even if an adversary has been secretly building a longer, or heavier, chain than the canonical chain that can replace it altogether. For example, checkpointing protects the Bitcoin network from a hypothetical attacker that may have been mining blocks to date since, or prior to, the network's inception in 2009 and has created a heavier chain than the current canonical chain. For this attacker to attract new miners which want to participate in the Bitcoin network it would have to provide them with client software that contains different checkpoints than those provided by the standard Bitcoin client.

Alternatively, in [KK21], Karakostas and Kiayias propose both a federated, and a fully decentralized checkpointing mechanisms. Their works protects against a block lead attack, whereby an adversary with more than half of the total network mining power can break the liveness property of the ledger across different checkpoint periods. However, their work relies on an external set of parties to checkpoint blockchains.

Remark. While the problem of creating an objectively acceptable checkpointing procedure is interesting, we believe that it is more promising to avoid the attack altogether. This can be accomplished, for example, through creating a procedure to safely merge two conflicting blockchains together to form a third longer, or heavier, chain. However, this procedure must resolve the conflicts not only between double-spending transactions, but also between smart-contract state updates. To the best of our knowledge, this approach has not been taken in a permissionless setting.

2.3 Mining Incentives

Primarily, the main reward offered by all such protocols comes in the form of coins within the distributed ledger in favor of miners. These rewards are intended as compensation for the miners' use of block creation resources to extend the canonical chain and maintain stable consensus on the ledger contents. Inversely, miners which create divergent branches are not meant to receive rewards because they impede the stability of the consensus process.

However, identifying with certainty which divergences were intentional or not is not always feasible, and therefore protocols cannot decisively consider every branching as non-compliant. Consequently, many state-of-the-art protocols take coarse approaches to issuing rewards and punishments. In this section we examine how these approaches operate under three scenarios. In Section 2.3.1 we review the state-of-the-art in granting coins to compliant nodes. Subsequently, in Section 2.3.2 we review how different protocols punish nodes which perform explicitly faulty behavior. Lastly, we give a review in Section 2.3.3 of how nodes can deviate from the protocol specification to maximize their gains beyond compliance rewards.

2.3.1 Compliance Payoffs

The fundamental reward scheme introduced in Nakamoto Consensus is to issue a **block reward** to the miner which successfully finds a block. This reward creates new coins and credits them to the miner within the block that it created. Additionally, a secondary reward, in the form of **transaction fees**, is granted to miners that create blocks. These fees are specified and paid by the issuers of the transactions which are included in each block. All fees paid by a block's transactions are credited to the miners that created said block, after being deducted from the balances of the transaction issuers. Notably, both rewards are only valid within the chain that contains the block that issues the rewards. However, as stale blocks are inevitable, such a stringent policy does not compensate compliant miners that inadvertently create a stale block.

More Inclusive Protocols. A more lenient strategy is to issue **uncle rewards** to miners which have created blocks that recently went stale [SZ15]. These rewards are issued in the canonical chain when it is extended by a block that proves the existence of one or more such recent stale blocks. Uncle rewards are split amongst the miner which extends the canonical chain with a block that references new stale blocks as uncles, and the miners which originally produced those stale blocks.

Another, more general, strategy is to reward more than one miner for the creation of a block.

The rationale behind this is that the reward distribution per block should be more representative of the block creation resource expenditure exhibited by miners throughout the search for a new block, and the rewards should not just be solely distributed to the miner which found the block's nonce in a "winner-takes-all" fashion.

In FruitChains, Pass and Shi quantify reward distribution fairness and use weak mining targets, named Fruits, to reward miners with coins if the fruits are included in a block on time [PS17]. Their work reduces the discrepancy between the variability of rewards of miners with large resources and of those with smaller resources.

Szalachowski et al. propose StrongChain, which uses weak mining targets to increase the weight of a chain and distribute block rewards amongst more miners by permitting the inclusion of headers of "weak" blocks in "strong" blocks [SRHS19]. This approach also reduces the per-block reward variance of miners with smaller amounts of resources.

Bissias and Levine generalize the mining target to be the average of the targets met by several miners in Bobtail, lowering the variance of inter-block arrival times and per-block rewards for miners utilizing resource amounts of all sizes [BL20].

Remark. While effective in many ways, the aforementioned works still distribute rewards in a lottery-like fashion, where only a limited number of miners may win a fixed-size pool of rewards per block. Such approaches only enable a competitive process for miners to earn coins, and does not adjust the amounts of the coin rewards based on the amount of block creation resources invested into the process by the miners.

Alternative Reward Protocols. Alternatively, Dong and Boutaba propose publishing a challenge on a decentralized ledger and subsequently computing a Proof-of-Sequential-Work (PoSeq) on that challenge to mint a new coin within a limited time-span in Elasticoin [Orl20, DB19]. This design does permit coin rewards to increase in proportion to the amount of computational resources used by miners to create PoSeqs, but only to a limited extent. This is because the Elasticoin design assumes that the decentralized ledger has sufficient bandwidth to confirm all PoSeqs within the allotted time before they expire.

In Melmint, Dong and Boutaba utilize Elasticoin minting to peg a cryptocurrency to the value of "one day of sequential computation on an up-to-date processor" and adjust its circulating supply depending on demand for it [DB20]. To stabilize the currency's value, a limited-time auction is operated on a decentralized ledger, which still may not be have sufficient bandwidth to enable full network participation in a timely manner.

On the other hand the Ergon protocol issues per-block coin rewards that are proportional to the block's mining difficulty [Trz]. Interestingly, however, Ergon attempts to peg the cost of production of its currency to the amount of energy miners consume, and consequently implement a mechanism similar to Bitcoin's halving schedule that is designed to make the same amount of rewards more difficult to attain over time [Trz21]. While this reward scheme introduces a unique form of proportionality between mining power and rewards, it still suffers from the drawbacks of winner-takes-all dynamics, and artificial scarcity.

Remark. To date, all state-of-the-art reward mechanisms do not scale up the potentially redeemable rewards in direct proportion to the total amount of resources utilized by miners, and instead encourage miners to compete for an increasingly scarce constant amount of coins.

2.3.2 Fault Penalties

In longest-chain protocols, penalties are deducted from the coin balances of nodes that provably disrupt the consensus process. Implicit forms of penalties are implemented in PoW and PoSpace systems, while PoS protocols implement explicit penalty mechanisms.

A reward that was issued to a miner under any state-of-the-art PoW or PoSpace protocol is contingent on the set of blocks that were created by the miner, in whole or in part, remaining part of the canonical chain [BBPS19, XZLH20, ZL20], or being referenced by the canonical chain as in uncle rewards [SZ15]. If a miner spends its resources to create a block that does not extend the canonical chain, or does not get referenced by it, it receives no compensation for its resource expenditure. The penalty, therefore, for not contributing to the consensus process is implicit in the waste of the miner's resources without compensation. Moreover, despite requiring the same amount of resources to create as canonical blocks, uncle blocks reward their creators with fewer new coins and no transaction fees. This also implicitly penalizes miners that would intentionally mine them.

On the other hand, in PoS protocols, consensus does not directly consume the staked resources, but only encumbers staked coins in the ledger. Consequently, if non-compliant miners use their staked resources to extend two chains simultaneously, the miners would not be prone to the same implicitly unrewarded waste in resources once a canonical chain is decided. This issue creates a challenge for PoS protocols known as the nothing-at-stake problem, which some PoS protocols address by using an explicit penalty system that functions similarly to uncle rewards [XZL+19, BSAB+19]. Explicit PoS penalties rely on blocks in the canonical chain referencing two stale uncles with the same parent block and the same creator. Using this reference as proof that the creator of both blocks attempted to induce a fork in the chain, a penalty is deducted from the staked coins of said creator in the canonical chain.

However, not all forms of misbehavior can similarly be proven as intentional [GHM+17]. Consider, for example, a PoS miner next in line to propose a block, but incurs a crash, or a power outage, before being able to do so. Penalizing such accidental faults as equally as intentional misbehavior would be unreasonable.

Remark. These penalty mechanisms are designed to dissuade a rational miner, which only aims to maximize the number of coins that it receives while participating in the consensus process, from destabilizing the view of the canonical chain. While reasonably effective at doing so, penalties only establish a limited barrier against miners whose non-compliant behavior could be motivated by external incentives, such as bribes [JSZ+19].

2.3.3 Deviation Advantages

Without considering bribery, or other external factors, the incentive and deterrent mechanisms in state-of-the-art consensus protocols can be gamed by non-compliant nodes to derive more rewards than intended. Notably, when the reward scheme in question promotes a competitive process that awards a scarce or finite resource, the rewards gained by a miner **relative** to those gained by other miners can become worth optimizing. Consequently, degrading the rewards of compliant miners [ES14], and even dissuading them from participation [MJP$^+$20], can become a rational objective in some reward schemes.

Zhang et al. [ZP19] propose four metrics as the foundation of a quantitative framework for evaluating how different PoW consensus protocols perform under adversarial mining strategies. The first metric is **chain quality**, which quantifies the share of blocks an adversary could expect to publish in the canonical chain based on how much hashing power the adversary controls. The second metric is **incentive compatibility**, which quantifies the share of rewards received by an adversary based on its hashing power. The third metric is **subversion gain**, which quantifies the additional profit an adversary could earn by performing double-spending attacks. The last metric is **censorship susceptibility**, which quantifies how much reward-loss an adversary could inflict on compliant miners based on its share of the hashing power. These metrics were also individually focused on and evaluated by several studies [ES14, SSZ16, NKMS16, WHF19, ZET20], and are utilized in this thesis to analyze the performance of Reward-All compared to Nakamoto in Chapter 5.

Chen et al. [CPR19] analyse block reward allocation schemes and propose a set of axioms which evaluate how the schemes perform. First, they consider a block reward scheme as **symmetric** if it does not vary rewards based on the identity or ordering of a miner. Second, they consider a scheme to achieve **budget-balance**, if the sum of all expected rewards per miner per block does not exceed 1. Third, a scheme achieves **Sybil-proofness** if splitting hashing power across different identities does not yield more rewards than dedicating that power to one identity. Lastly, a scheme achieves **collusion-proofness** if forming coalitions of miners does not yield more rewards for the coalition than the sum of rewards of the independent miners. Weaker and stronger variations of the aforementioned axioms are also presented in [CPR19], but they are omitted from this thesis for brevity. They demonstrate that Bitcoin's proportional allocation scheme satisfies these criteria. However, this analysis is based on long-term behavior of Bitcoin rewards, i.e. everything is well defined and balanced only on-the-long-run.

Kwon et al. [KLK+19] quantify the decentralization of PoW, PoS and DPoS consensus protocols. In their work, they quantify the decentralization of a network using the difference in combined resources between the most powerful miners, and the remaining miners which represent a certain percentile of a known number of miners. With this measure in mind, they introduce a set of constraints for reward schemes to be able to, with high probability, reach a state of full decentralization, such that the aforementioned difference in mining resources is negligible. They argue that systems without Sybil costs fail to promote decentralization, regardless of whether their consensus protocol is based on computational work or on stake. Their results imply that current reward schemes are not inherently designed to encourage decentralization, since the cost of mining using fragmented mining power across multiple identities is not less than that of mining using a single identity. However, their analysis is based on a competitive coin-creation process, where miners compete for the next set of newly issued coins, and the miners which reinvest more of their earnings into attaining more mining power are destined to control a majority of network resources. Whether such a dilemma exists in a less competitive coin creation process, such as those of the decoupling and responsive protocols from Section 2.3.1, is unclear.

Remark. Thus far, mining reward scheme analyses have provided valuable insights into many aspects of existing reward mechanisms. However, a gap in knowledge exists on how **coin value** is truly distributed between miners under different reward schemes.

2.4 Summary

In this chapter we examined the current state-of-the-art in blockchain ledger data schemes, permissionless consensus protocols, and reward mechanisms. Our examination introduced the basic goals and principles of these designs, and the most pertinent issues they face.

In Section 2.1 we first reviewed the basic data schemes used in state-of-the-art distributed permissionless ledgers, and highlighted the susceptibility of single-use addresses to race conditions, and lack of parallel transaction execution in multi-use addresses. Then, we examined the methods for interpreting transactions and executing them to modify ledger contents in a finite amount of time, pointing out the lack of access current systems provide for transactions to read transaction fee data. Lastly, we reviewed how blockchain ledgers accept updates in the form of sequential blocks, and emphasized how such block structures are not optimized for efficient tamper-evidence.

Later in Section 2.2, we began with a review of how nodes in state-of-the-art permissionless distributed ledger systems are networked and communicate to share the contents of the ledgers they maintain, identifying the gap in their defenses against attackers with a strong network presence. Subsequently, we reviewed three different classes of longest-chain protocols, overviewing how they aim to establish consensus on the canonical chain between miners, and the concessions they make in energy usage, resource exclusivity, and permissionless participation. Lastly, we examined the implications of stale blocks on longest-chain protocols, overviewing both the analyses performed on the attacks that can be carried out using stale blocks, and the defenses that have been proposed to prevent blocks from going stale. In both cases, we comment on the effectiveness of longest-chain despite the feasibility of attacks, and on our preference for a different class of defense than what the state-of-the-art offers.

Finally, we commenced Section 2.3 with a review of incentive mechanisms in permissionless distributed ledgers which compensate miners for their resource expenditures, calling attention to the fact that protocols which aim to be more inclusive of smaller miners still rely on lottery-like mechanisms for reward issuance, while alternative reward schemes suffer from performance bottlenecks and enforce unfair artificial scarcity. Then we examined state-of-the-art mechanisms for punishing non-compliant nodes for disruptions to the consensus process, highlighting that such mechanisms can only deter non-compliant behavior to a limited extent as they do not account for bribes external to their systems. We then concluded with an overview of analyses on the profitability of participating in and gaming these reward and punishment mechanisms to maximize relative rewards, identifying a gap in knowledge on how these mechanisms truly distribute the value behind coins amongst participants.

Chapter 3

Equitable Proof-of-Work Mining Rewards

Nakamoto utilizes its lottery approach to not only pick a leader for consensus, but also to allocate the next set of reward coins without consideration for the amount of computational resources the entire network of miners have used to operate the lottery. More specifically miners with relatively small mining powers, and miners which join the network at a relatively late stage where more miners compete for a fixed amount of coins, pay more than others to create the same amount of coins. This is due to mining difficulty increasing as more miners participate, while the number of reward coins being halved every $210,000$ blocks as per Nakamoto's Bitcoin implementation.

The original goal of these restrictions on rewards were intended to only compensate miners which contribute to the difficulty of creating blocks, and to ensure that the supply of the coins issued does not grow out of proportion [N+08]. However, no clear arguments were given for why Nakamoto's restricted approach to reward issuance was an optimal solution to these challenges, leaving this important research question unanswered, and providing no grounded basis for understanding the side effects of restricted coin issuance on Nakamoto miners.

In this chapter we propose a computationally-grounded approach to analyzing and issuing Nakamoto mining rewards. Our main focus is to establish a set of building blocks which we can use to model and quantify the costs of attaining reward coins in terms of the number of Proof-of-Work attempts made by miners. Using this model, we aim both to identify the bottlenecks of Nakamoto reward issuance which unfairly disadvantage some miners to the benefit of others, and to propose a set of design constraints under which all miners are treated fairly. The main obstacle we aim to overcome in this chapter is to issue equitable rewards in a way that provides a similar level of stability in coin supply growth as done in Nakamoto.

The remainder of this chapter is organized as follows. First, in Section 3.1 we introduce our computationally-grounded framework for establishing a valuation metric for coins. Then, in Section 3.2 we present the current sources of inequity in state-of-the-art PoW mining reward schemes, and quantify their effects on mining in Bitcoin. We then introduce in Section 3.3 a set of constraints for achieving equitable rewards, and argue for their efficacy and reasonable coin supply growth dynamics. Lastly, we summarize this chapter's contents in Section 3.4. In Chapter 4, we present the details of our Reward-All protocol, which rewards miners according to our requirements.

3.1 Computational Coinage Framework

In this section we present our computationally-grounded framework for establishing a valuation of a PoW cryptocurrency's coin supply based solely on the amount of hashing power dedicated to mining it. The first goal of this framework is to establish the criteria under which we measure coin production costs. The second goal is to introduce metrics for quantifying the value of the new coins awarded in exchange for mining expenditures, without consideration for auxiliary sources of compensation, such as transaction fees. To achieve our two goals, we introduce metrics of our computationally-grounded framework according to the following criteria.

Expenditures. The primary expenditure we consider in this computationally-grounded framework is the hash-function calculation used to find valid PoWs, which abstracts away the finer details of real-world resource requirements of this search, and accounts only for the number of hash calculations that the usage of resources results in, and the time required to finish the calculations. This purely hash-based approach was taken to establish a computationally-grounded framework for analysis, in the sense that no external variables which affect the real-world resources required to compute a hash are considered. To elaborate, we do not account for market variables such as electricity prices, hardware cost, maintenance fees, or any similar expenses.

Compensation. We consider only newly minted coins as the primary form of compensation in exchange for hashing expenditures. Similar to the abstraction of real-world costs using hash-function calculations, coins also abstract away any valuation metrics external to a blockchain, such as the coin's exchange rate or purchasing power. Expressing the amount of compensation which a reward scheme issues is our second requirement for establishing a computationally-grounded coin value.

Value. By combining the expenditure and compensation metrics, we establish a purely hash-based valuation unit for coins suitable for our computationally-grounded framework. This approach aims to base the measurement of the value of coins purely on the number of hashing computations dedicated to creating them, enabling the quantification of the hash-based value that can be earned through mining some number of coins.

Notably, our framework is applicable regardless of how miners are rewarded, what hashing function is used, or how block creation conditions are set.

3.1.1 Miner Metrics

In this section, beginning from a single miner's local perspective, we first define what a local miner represents in our computationally-grounded framework, and then present the relevant metrics for a miner.

Definition 3.1 (Miner m_μ) *A miner, denoted by m_μ, where μ is some unique identifier, is characterized at time t by (i) its absolute mining power, denoted by $power(m_\mu, t)$, in hashes* *per second, and (ii) its reward issuance difficulty, denoted by difficulty*(m_μ, t)*, in expected number of hashes.*

We consider a single miner in our computationally-grounded framework at time t as a component with a certain hashing throughput, and an expected number of hashes to perform before being rewarded, as per Definition 3.1.

For example, let m_{4050} be a miner with $\mathit{power}(m_{4050}, t) = 2^{40}$ hashes per second that meets the reward issuance difficulty once every $\mathit{difficulty}(m_{4050}, t) = 2^{50}$ hash computations on average for all t. This simple definition of a miner m_μ is the basis on which we build the remainder of the metrics in this section.

Given a miner m_μ, the expected length of the period of time between each reward issuance to the miner, denoted by $\mathit{period}(m_\mu, t)$, can be derived using Equation 3.1.

$$\mathit{period}(m_\mu, t) = \frac{\mathit{difficulty}(m_\mu, t)}{\mathit{power}(m_\mu, t)} \tag{3.1}$$

For example, m_{4050} is expected to be issued a reward every $2^{50} \div 2^{40} = 1024$ seconds on average. Because the process is memoryless, this expected time does not change with t.

Definition 3.2 (Average Hash-Time-to-Issuance *hti*(m_μ, t)) *The average hash-time-to-issuance metric, denoted by hti*(m_μ, t)*, represents the average amount of time before a miner's hash computation is rewarded.*

In addition to the reward period, we establish in our framework a measurement of the time between completing each hash calculation, and receiving a reward as compensation for it. For every miner m_μ the average hash-time-to-issuance $\mathit{hti}(m_\mu, t)$ quantifies the average amount of waiting time for m_μ that is associated with every hash computation before the reward issuance difficulty is met, as per Definition 3.2. This is slightly different from the average amount of time m_μ has to wait before meeting the issuance difficulty, because the average waiting time in $\mathit{hti}(m_\mu, t)$ is accounted for *per hash* rather than per $\mathit{difficulty}(m_\mu, t)$ hashes.

Assuming that the miner's hashing throughput is uniformly sustained, which will always be assumed to be the case in the remainder of this thesis, $\mathit{hti}(m_\mu, t)$ is calculated using Equation 3.2.

$$\mathit{hti}(m_\mu, t) = \frac{\mathit{difficulty}(m_\mu, t) - 1}{2 \times \mathit{power}(m_\mu, t)} \tag{3.2}$$

As an example, consider the miner m_{13}, where $\mathit{power}(m_{13}, t) = 1$ hash per second, and $\mathit{difficulty}(m_{13}, t) = 3$ hashes on average. Using Equation 3.2, $\mathit{hti}(m_{13}, t) = 1$. This can be derived by examining the waiting time expected to be incurred after performing each hash computation as follows:

1. After the first hash computation, m_{13} has to spend two more seconds computing two more hashes on average before meeting the issuance difficulty, and so the waiting time incurred after computing the first hash is 2 seconds.

2. After the second computation, m_{13} spends 1 more second computing one more hash on average.

3. Lastly, m_{13} is expected to have met its issuance difficulty right after the third computation without any additional waiting time.

The total waiting times divided by the number of hashes is equal to $\frac{2+1+0}{3} = 1 = \mathit{hti}(m_{13}, t)$, meaning that the average time the miner waited between computing each hash and receiving a reward is 1 second, which is different from the expected waiting time of 3 seconds for each reward issuance. However, for much larger numbers, the metric can be approximated as $\mathit{hti}(m_\mu, t) \approx \frac{\mathit{period}(m_\mu, t)}{2}$ with negligible error.

As previously stated, the reasoning behind introducing $\mathit{hti}(m_\mu, t)$ is to quantify the average delay between miners' hash expenditures, and the reception of rewards. This metric will prove useful later on when we describe the opportunity cost aspect associated with the costs of creating coins.

Definition 3.3 (Reward *reward*(m_μ, t)) *The reward, denoted by reward(m_μ, t), is the expected number of coins received by a miner m_μ that finds a PoW that meets its reward issuance difficulty at time t.*

For a single miner m_μ, we denote in our framework the reward received by m_μ as *reward*(m_μ, t) to express a miner's compensation in coins, as per Definition 3.3. For example, for all Bitcoin miners, *reward*$(m_\mu, t) = 625,000,000$ coins[1] (satoshis).

Definition 3.4 (Difficulty-to-Reward Ratio *drr*(m_μ, t)) *The Difficulty-to-Reward Ratio, denoted by drr(m_μ, t), is the average number of hashes computed per reward coin for a miner.*

Furthermore, for a miner m_μ, we establish the Difficulty-to-Reward Ratio as a valuation metric for the average cost of a coin in hashes, as per Definition 3.4. Equation 3.3 defines the formula for *drr*(m_μ, t).

$$drr(m_\mu, t) = \frac{difficulty(m_\mu, t)}{reward(m_\mu, t)} \tag{3.3}$$

As an example, consider miner m_{4050} from before, with *reward*$(m_{4050}, t) = 2^{30}$ coins. Using Equation 3.3, the hash-based cost per coin for m_{4050} is equal to *drr*$(m_{4050}, t) = 2^{50} \div 2^{30} = 1,048,576$ hashes per coin on average.

3.1.2 Blockchain Metrics

In this section we introduce metrics for a chain of PoW-based blocks, rather than a single miner as in the previous section. Similarly, we first present the definition of a blockchain, followed by its associated metrics.

Definition 3.5 (Blockchain B_β) *A blockchain, denoted by B_β, where β is some unique identifier, is the product of a network of miners participating in a PoW protocol.*

[1]This reward is scheduled to be halved to 312,500,000 after the current batch of 210,000 blocks is mined.

Our computationally-grounded framework's concept of a blockchain is devoid of implementation details, and is only constructed to enable the expression of a select few metrics of interest, as per Definition 3.5.

Definition 3.6 (Chain Hashcap *hashcap*(B_β, t)) *The hashcap of a chain of blocks, denoted by hashcap(B_β, t), is an estimate of the expected total number of hash function calculations performed by miners to create the chain.*

From a blockchain viewpoint, the aggregated number of hash function calculations performed by miners to create a blockchain is referred to as the chain's *hashcap*[2], as per Definition 3.6. As the hashcap value provides an indication of the miners' expenditures towards creating a blockchain, we use it as the computationally-grounded valuation metric of the hash-based cost of production of a blockchain.

The hashcap metric value can be estimated for existing state-of-the-art PoW blockchains, such as Bitcoin, by summing the difficulty parameter for each block in a chain. However, while some hash functions can be implemented very efficiently in hardware as ASICs, some are designed to operate efficiently only on general purpose hardware such as commercially available CPUs. Consequently, the hashcaps of two blockchains which use two different hashing functions are not directly comparable.

Definition 3.7 (Coinage *coinage*(B_β, t)) *The coinage of a chain of blocks, denoted by coinage(B_β, t), is the total number of coins rewarded to miners in the chain as of time t.*

We refer to the total supply of coins issued as rewards in a blockchain B_β in this framework as the chain's coinage, as per Definition 3.7. This term is also known as the circulating coin supply, or coin supply for short, in cryptocurrency markets.

[2]The naming of the term *hashcap* is derived from the word Market Capitalization, or Marketcap for short, which is used in cryptocurrency markets to represent the total theoretical value of an entire supply of coins based on the coin's market price in another currency.

Definition 3.8 (Hashcap-to-Coinage Ratio $\boldsymbol{hcr}(B_\beta, t)$) *The Hashcap-to-Coinage Ratio for a blockchain, denoted by $hcr(B_\beta, t)$, is the average number of hashes computed per coin issued in the chain as of time t.*

The Hashcap-to-Coinage Ratio is a metric that is similar to the Difficulty-to-Reward Ratio, but defined for a chain of blocks rather than for a miner, as per Definition 3.8. Using the previously defined terms for hashcap and coinage, we formulate $hcr(B_\beta, t)$ in Equation 3.4.

$$hcr(B_\beta, t) = \frac{hashcap(B_\beta, t)}{coinage(B_\beta, t)} \tag{3.4}$$

This ratio is a key metric, as it enables the estimation of the average hash-based cost of production of a cryptocurrency coins, as we will present in Section 3.1.3.

3.1.3 Coin Metrics

Insofar, in the previous two sections, we have presented basic metrics of interest related to mining throughput, difficulty, and rewards. In this section, we will utilize these metrics to construct valuation metrics which can be used to quantify the hash-based value gained, or lost, from mining coins.

Definition 3.9 (Fungibility Dilution Factor $\boldsymbol{fdf}(m_\mu, B_\beta, t)$) *The Fungibility Dilution Factor for a miner in a blockchain, denoted by $fdf(m_\mu, B_\beta, t)$, is the amount by which the hash-based cost of a miner's reward will be amplified in the blockchain at time t.*

We quantify how much gain, or loss, in hash-based value is made by a miner due to fungibility using the Fungibility Dilution Factor, presented in Definition 3.9. Using our previously defined metrics, dividing the miner's difficulty-to-reward ratio by the blockchain's hashcap-to-coinage ratio results in the fungibility dilution factor, as per Equation 3.5.

$$fdf(m_\mu, B_\beta, t) = \frac{drr(m_\mu, t)}{hcr(B_\beta, t)} \tag{3.5}$$

Coin fungibility dictates that all coins in a chain's coinage are perfectly interchangeable with one another. Because of this, once a new set of coins are created by a miner m_μ in a blockchain B_β, we treat them in our computationally-grounded framework as having a hash-based value equal to the chain's hashcap-to-coinage ratio, even if the difficulty-to-reward ratio that was exhibited by the miner in creating these coins was different (i.e. $hcr(B_\beta, t) \neq drr(m_\mu, t)$).

For a given miner and blockchain, when $fdf(m_\mu, B_\beta, t) > 1$, then the miner will receive a set of coins which represent a larger number of hashes than the miner performed to receive them (i.e. $hcr(B_\beta, t) > drr(m_\mu, t)$). On the other hand, when $fdf(m_\mu, B_\beta, t) < 1$, then the miner will receive a set of coins which represent a smaller number of hashes than performed (i.e. $hcr(B_\beta, t) < drr(m_\mu, t)$). When $fdf(m_\mu, B_\beta, t) = 1$, m_μ is issued in B_β a set of coins worth as many hash computations as were performed to receive them.

Consequently, we consider miners to have made a hash-based gain in our computationally-grounded framework if $hcr(B_\beta, t) > drr(m_\mu, t)$. In other words, if a miner is rewarded with a set of coins at a hash-based cost that is less than the chain's hashcap-to-coinage ratio, it was received a set of coins at a discount. This, of course, does not necessarily mean that the miner can make a profit selling its coins in a real-world market, where prices may not necessarily be dictated by hash-based valuation metrics. Similarly, a loss is said to have been incurred in our computationally-grounded framework, if $hcr(B_\beta, t) < drr(m_\mu, t)$.

Definition 3.10 (Hash-Restitution Time $\boldsymbol{hrt}(m_\mu, B_\beta, t)$) *The Hash-Restitution Time of a miner in a blockchain, denoted by $hrt(m_\mu, B_\beta, t)$, is the amount of time (or number of blocks) between m_μ first receiving a $reward(m_\mu, t)$, and the first time t' (or block) during which $hcr(B_\beta, t') \geq drr(m_\mu, t)$ holds true (i.e. $hrt(m_\mu, B_\beta, t) = t' - t$).*

For the case when $fdf(m_\mu, B_\beta, t) < 1$, we introduce the Hash-Restitution Time in Definition 3.10, which is a means to analyse the amount of time a miner had to wait for the hashcap-to-coinage ratio to first meet the difficulty-to-reward ratio at which it created coins, i.e. the amount of time m_μ has to wait to be paid back in hash-based value.

Unlike previous metrics, $hrt(m_\mu, B_\beta, t)$ may not be immediately measurable for a miner that receives a reward while $fdf(m_\mu, B_\beta, t) < 1$. This is because it may not be easily determinable when exactly in the future $hcr(B_\beta, t)$ is expected to rise, due to instability in mining power or reward-scheme intrinsic reasons. In fact, for some PoW schemes, miners may never even be fully paid back if $hcr(B_\beta, t)$ never follows a sufficiently long upwards trend.

However, once this value is known, it serves as an indicator in our framework for m_μ to quantify the opportunity cost it had to pay before its rewarded coins attained a hash-based value that is equal to the hash-based cost m_μ paid for them. In some cases, m_μ may never wait long enough, and transfer its coins.

Again, this metric is agnostic about both the specifics of the PoW blockchain it is used on, and the coin-spending behavior of miners. Its sole purpose in our computationally-grounded framework is to provide a measurement point that can be used to gain insight about how a reward-scheme delays fully compensating miners in hash-based value.

3.2 Inequity in Nakamoto

In this section, we apply our computationally-grounded framework introduced in Section 3.1 to Bitcoin, the most prominent realization of Nakamoto, practically analysing and quantifying the state of hash-based coin valuation in its network. Our analysis focuses on identifying and highlighting sources of inequity in Nakamoto, and demonstrating how these sources have affected Bitcoin's miners and coinage in practice.

Bitcoin Adaptation. To model Bitcoin using our computationally-grounded framework, we integrate the Bitcoin-specific policies on difficulty and reward using the three following rules:

1. Only a single blockchain B_β is assumed to be mined on without forks.

2. For any miner m_μ, $\mathit{difficulty}(m_\mu, t)$ is equal to the current block mining difficulty for the Bitcoin blockchain B_β.
3. For any miner m_μ, $\mathit{reward}(m_\mu, t)$ is equal to the current Bitcoin block reward per its reward halving schedule.

The first rule applied to our framework is a simplifying assumption to focus the analysis on the best case scenario where consensus is working as intended. While this prevents the examination of the state of affairs during forks, it will highlight how even ideal conditions fail to establish equity amongst miners.

The second rule enables our framework's definition of difficulty to follow that of Nakamoto, whereby mining a block is the only directly rewardable action in the protocol. We define $\mathit{difficulty}(m_\mu, t)$ to abstract away the out of scope details of Bitcoin's difficulty adjustment algorithm, since they hold no repercussions for our analysis.

Similarly, the third rule applies Bitcoin's reward schedule, which begins with $5,000,000,000$ coins (satoshis) and halves the reward every $210,000$ blocks.

3.2.1 Inequitable Hash-Time-to-Issuance

The fact that a miner has to wait for the chance to be a block's creator in Nakamoto creates different hash-time-to-issuance costs for miners of relatively different sizes. This difference is presented in Figure 3.1, which shows the linear relationship between a miner's m_μ relative size and its expected $\mathit{hti}(m_\mu, t)$ value in Nakamoto. While other literature [SRHS19] analyzes this difference using the coefficient of variation of rewards received per block for a miner, our $\mathit{hti}(m_\mu, t)$ metric gives a more concrete sense of the differences in time-related *opportunity costs* for miners of different relative sizes. These results extend directly to Bitcoin.

The cost implications of the aforementioned relationship in Nakamoto can be further elaborated by comparing two data points from the above plot. For example, consider two miners, one with 1% of the total network hashing power, and one with 0.01%. The first miner would

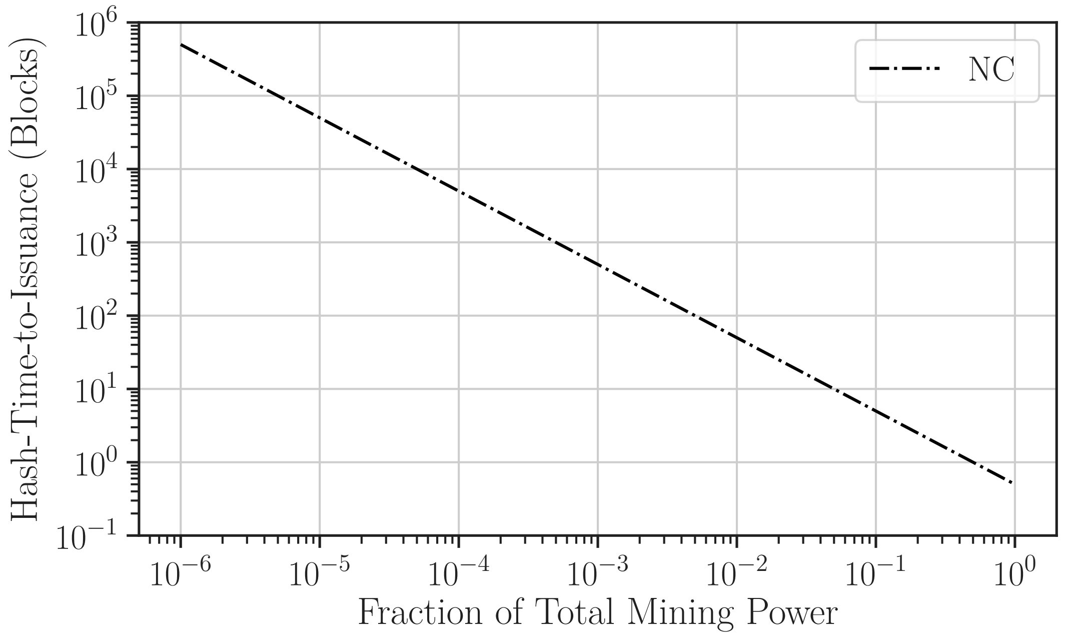
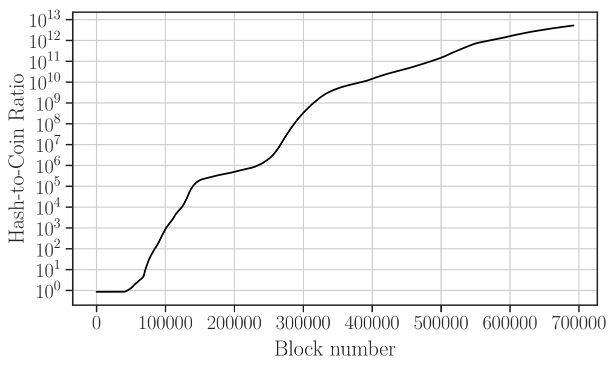


Figure 3.1: Hash-Time-to-Issuance versus relative mining power in Bitcoin.

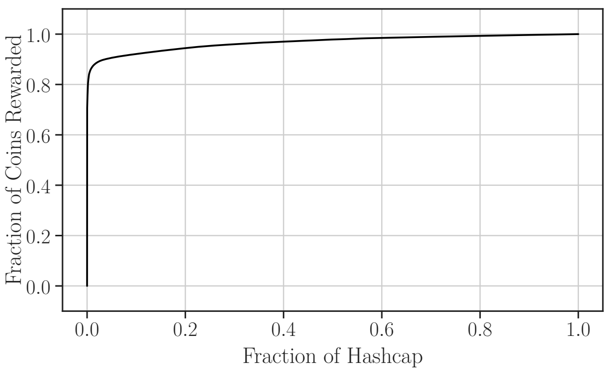
have to experience an average delay of 200 blocks before receiving compensation for each hash it computes, while the second miner's delay is $20,000$ blocks per hash. Under Bitcoin's 10-minute average block interval, the relatively larger miner can spend its mining rewards approximately every 1.39 days on average, while the smaller miner can only do so every 4.64 months approximately. This severely disproportional difference in reward delay between miners is not directly apparent from analyses which utilize the coefficient of variation.

3.2.2 An Increasing Hashcap-to-Coinage Ratio

Because Bitcoin's reward scheme does not adjust the number of coins issued in response to the difficulty of mining, and mining difficulty has increased significantly since Bitcoin's deployment, the Hashcap-to-Coinage Ratio has increased exponentially several times. Even with a stable mining difficulty, the reward-halving schedule causes the Hashcap-to-Coinage Ratio to increase. In Figure 3.2, both the Hashcap-to-Coinage Ratio (Fig. 3.2a), and the distribution of the hashcap (Fig. 3.2b) over coins are presented. The data presented in Figure 3.2 was estimated using Bitcoin's block difficulty parameter as the hash-based cost of production of reward coins.



(a) Hashcap-to-Coin Ratio versus block number in Bitcoin.



(b) Cumulative distribution of Hashcap across coin supply in Bitcoin.

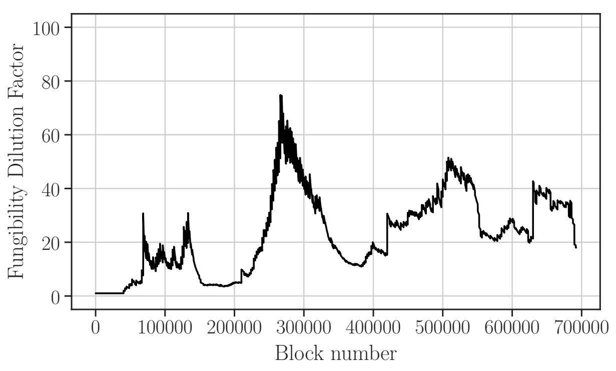
Figure 3.2: Hashcap-to-Coinage Ratio plots.

From a hash-based cost perspective, these increases have caused the running average hash-based cost of production for a single coin to increase dramatically over time in Bitcoin. The insight Figure 3.2a offers is that almost all of Bitcoin's hash-based value was computed after $400,000$ blocks were created.

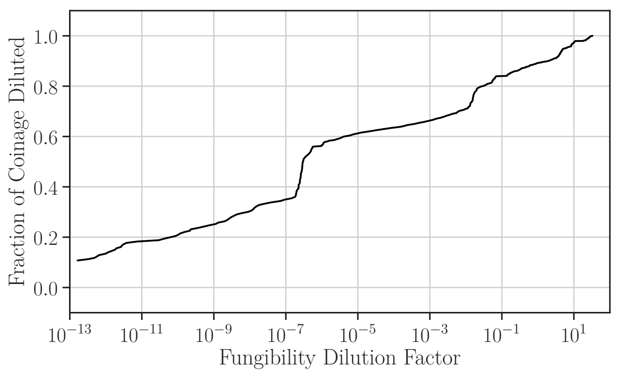
Because of Bitcoin's reward scheme, which is adapted to this computationally-grounded framework using rules #2 and #3, the difficulty-to-reward ratio of all miners has only worsened over time as mining power has increased. To further understand the advantage earlier miners had, one can see how much hashing power was exchanged for different portions of the coin supply in Figure 3.2b. Remarkably, the coin supply of Bitcoin has an incredibly skewed distribution, where more than 80% of the total coin supply was rewarded in exchange for less than 1% of the total number of hashes calculated to maintain the system.

3.2.3 Subsidy through the Fungibility Dilution Factor

The inequity between miners which participate at different times is not just restricted to coin creation costs, but also extends to the hash-based value of received rewards. Because coins are perfectly interchangeable in the ledger, they are are valued equally, even if some have higher hash-based costs than others. To quantify the extent to which this affects the hash-based valuation of mining rewards in Bitcoin, we present the Fungibility Dilution Factor of coins at the time of issuance in Figure 3.3a, and across the coin supply in Figure 3.3b.



(a) Fungibility Dilution Factor versus block number in Bitcoin.



(b) Cumulative Fungibility Dilution Factor Distribution over Coinage.

Figure 3.3: Fungibility-Dilution Factor plots.

If the FDF lies around a value of $1 \pm \epsilon$, where ϵ is some negligible value, then the hash-based value of the mining rewards can be considered to correspond to the number of hash computations performed to attain them. Instead, in Figure 3.3a, it can be seen that Bitcoin's FDF, after running at an almost constant value of 1 for a few thousand blocks, oscillates dramatically over time, leading the issued mining rewards to have a hash-based value[3] that is tens of times less than their hash-based cost.

However, due to the fluctuations of the FDF, the fungibility dilution factor of coins changes after their issuance. Namely, Figure 3.3b presents the cumulative distribution of the FDF over the coin supply so far. Remarkably, while the dilution factor has remained below 100 so far for Bitcoin, it has caused over 70% of the coinage to have its hashcap amplified by over 100-fold in value, with nearly 50% even growing by at least 1-million fold.

Figure 3.3a was constructed by dividing the Difficulty-to-Reward ratio for each block by the Hashcap-to-Coinage ratio as of that block. Figure 3.3b was constructed using the Difficulty-to-Reward ratio for each block and the Hashcap-to-Coinage as of the latest block in the dataset.

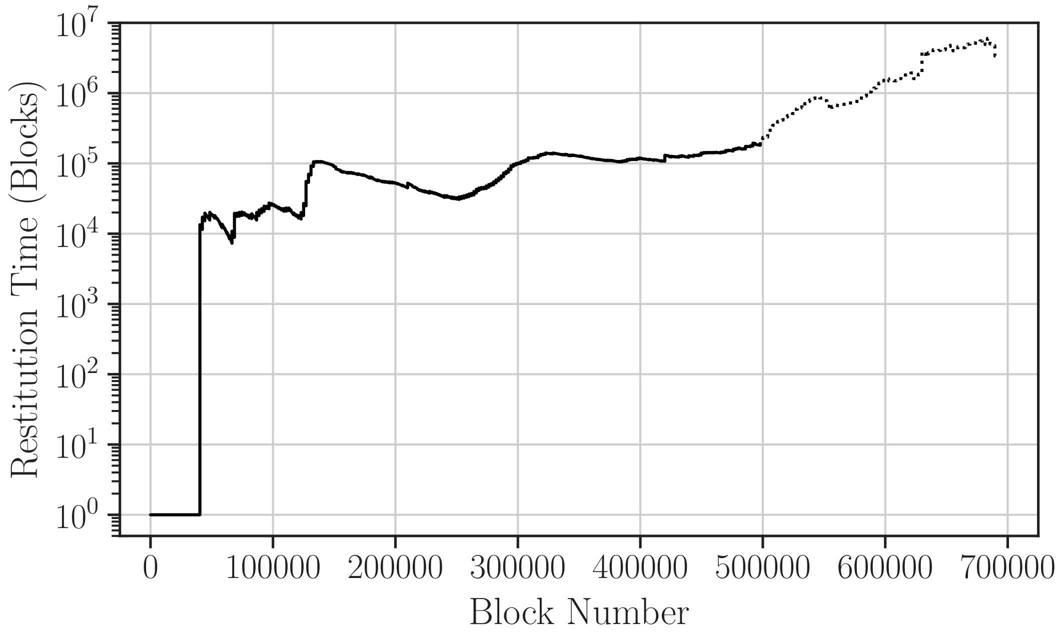


Figure 3.4: Hashcap-Restitution time, in blocks, for each block reward in Bitcoin.

3.2.4 An Increasing Hash-Restitution Time

Given that Bitcoin's fungibility dilution factor has remained well above 1 for most of its lifetime, the Hash-Restitution Time can be used to quantify how long an issued mining reward will take to reach a hash-based value that is at least equal to its hash-based cost.

In Figure 3.4 we estimate the Hash-Restitution Time in blocks for the rewards of the first 500, 000 Bitcoin blocks. As for the remaining blocks, since they have not yet reached an equitable hash-based value, their values are projected based on the assumption that the mining power remains the same as that of the last block used in the analysis.

Relatively shortly after the genesis block, Figure 3.4 shows that miners had to wait for a number on the order of 10, 000 blocks, or approximately 2 months, for the hash-based value of their rewards to reach their hash based costs. Starting from block 300, 000, the Hash-Restitution Time goes up to the order of 100, 000 blocks, or approximately 2 years.

Whether this is done by design to achieve a kind of lock-in effect, or simply an unintended consequence, the Hash-Restitution Time in Bitcoin seems to be uncontrolled, and trending

[3]at the time of reward issuance

towards impracticality for miners which participate relatively late in the protocol's lifetime.

Beyond Bitcoin. While the analysis presented in this section only pertains to Bitcoin, the same methods and reasoning are applicable to other designs with minor adjustments. For example, to apply this approach to Ethereum, one would have to account for *uncle blocks*, factoring in their reward and mining difficulty into the chain's coinage and hashcap respectively.

3.3 Equitable Reward Constraints

Having defined our computationally-grounded framework, which we use to express equity, and demonstrated the current state of inequity in Bitcoin, we establish in this section a set of constraints for achieving equitable rewards in Proof-of-Work Mining. The main focus of the constraints in this section is to prevent miners with relatively small hashing powers, and miners which participate at relatively late, or early, stages of a blockchain's lifetime, from being forced to create coins at a relatively high cost of production.

Basic Approach. The main requirements that we use to establish the design constraints of this section are **proportionality**, and **timeliness**, of rewards, such that:

- The reward scheme issues coin rewards that have a hash-based value that is proportional to their miner's hash-based cost of attaining them.
- The reward scheme issues coin rewards to all miners in an amount of time bounded by the network's block creation interval.

The reasoning behind these requirements is to prevent any underpayment or devaluation in terms of hash-based value, allowing miners to receive coins with a hash-based value equal to their contribution to the system, without unjustifiable delays in compensation.

In Sections 3.3.1, and 3.3.2, we motivate and present the design constraints in terms of the computationally-grounded framework from Section 3.1, such that each constraint is expressed as a set of restrictions that the protocol must place on the relevant framework metrics.

However, to achieve the notions of equity that we aim for, we propose a unique coin issuance approach. Namely, it can no longer be the case that a constant number of coins are issued per block in a winner-takes-all fashion, while having block mining difficulty be a variable. Instead, in our approach, we propose that mining rewards become uncapped. In Section 3.3.3, we examine the effects of adopting our unrestricted reward scheme on coin-supply growth, and outline the conditions for relative coin supply stability.

3.3.1 Undiluted Reward Constraints

The first constraint, expressed in Equation 3.6, aims to maintain a fixed correspondence between the number of hashes performed by miners, and the number of coins they receive in return.

$$hcr(B_\beta, t) \approx drr(m_\mu, t) \tag{3.6}$$

Such a constraint means that, unlike Bitcoin, for any miner to receive a coin, it must perform approximately $hcr(B_\beta, t)$ hash calculations. Essentially, this correspondence aims to stabilize the Difficulty-to-Reward Ratio of all miners to be the Hashcap-to-Coinage Ratio of the entire blockchain, such miner rewards are not diluted.

Stabilizing this correspondence is an endeavor that is in stark contrast to existing blockchain reward schemes, which employ mechanisms that directly destabilize this relationship. For example, Bitcoin's so-called *halving* schedule fundamentally causes spikes in Bitcoin's HCR. These spikes are further exacerbated by changes in Bitcoin's block mining difficulty, which does not affect how many coins are rewarded per block. Whether Bitcoin's HCR instability is by design, or beneficial, our approach is to keep the Hashcap-to-Coinage ratio relatively stable, and avoid directly granting any advantages to miners which participate during periods of lower block mining difficulty.

Remarkably, when the HCR is unstable, the fungibility of coins distorts the hash-based cost of production for the entire coin supply. In Bitcoin, this leads to a situation where a majority of its coin supply is produced at a marginal cost compared to the remainder, yet the entire coinage is treated as having equal face-value. This is inherently unfair to miners which participate while the HCR is increasing, i.e. miners with a DRR higher than the HCR, as their computational resources are being spent towards increasing the HCR, rather than towards receiving more coins. Under our approach on the other hand, the goal is to not dilute miner rewards, and stabilize the Fungibility Dilution Factor for all miners of the chain as expressed in Equation 3.7.

$$\mathit{fdf}(m_\mu, B_\beta, t) \approx 1 \tag{3.7}$$

Of course, these constraints do not cover cases where miners willingly forfeit rewards, or receive a penalty for misbehavior in consensus. In such circumstances, the hashes computed by such miners will go unrewarded, theoretically increasing the HCR of the blockchain.

3.3.2 Prompt Restitution Constraints

Our second reward constraint is the prompt, and direct, distribution of coins to miners in exchange for their hash calculations. This is in contrast to distributing rewards indirectly on the long run using lotteries, or any other mechanisms similar to the round based winner-takes-all Bitcoin dynamic.

More concretely, each miner is rewarded on average at least once per block for each of its search attempts for a PoW, instead of having to wait for some expected number of blocks to be created before being credited, following the constraint described by Equation 3.8.

$$\mathit{hti}(m_\mu, t) \approx \mathit{blocktime}(B_\beta, t) \tag{3.8}$$

For example, a Bitcoin miner with one third of the total network mining power would have to wait two blocks on average before receiving its block reward. On the other hand, an equitably treated miner with the same hashing power would expect to receive a third of the total coins rewarded to the network every block.

This constraint, in conjunction with that of Section 3.3.1, lead towards establishing a short

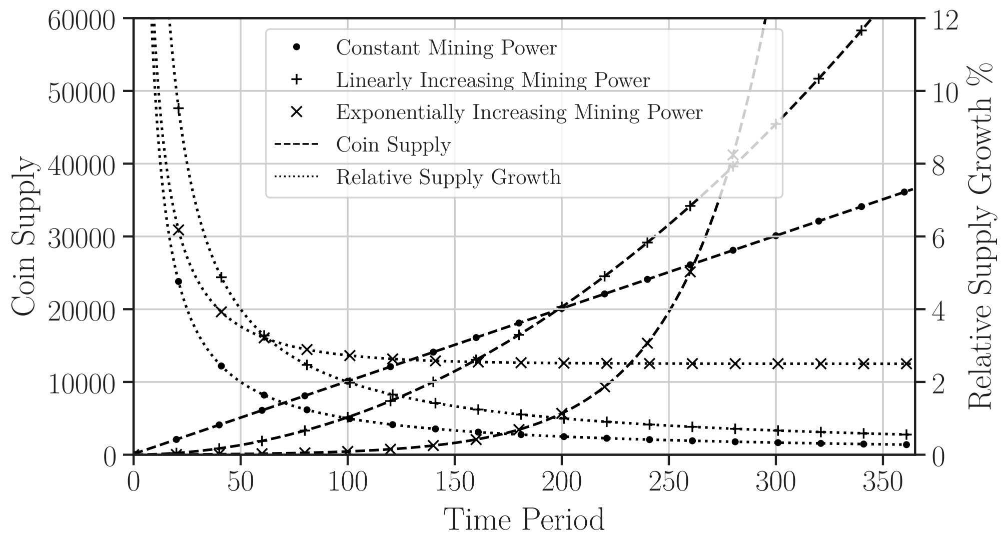


Figure 3.5: Plot of Time versus Coin Supply and Relative Supply Growth (in Percentage) under three different mining power growth scenarios in Reward-All. Points on dashed lines represent Coin Supply and fall on the leftmost y-axis. Points on dotted lines represent Relative Supply Growth and fall on the rightmost y-axis. Under constant mining power over all time periods (●) the relative supply growth per time period of coins goes to zero over time. Similarly, under linear growth of mining power (+), where $1x$ more power is added per time period, the relative supply growth per time period also goes to zero over time. However, under exponential mining power growth (×), where 2.5% more power is added per time period, the relative supply growth converges to the mining power growth over time.

and stable Hash-Restitution Time for all miners, as presented in Equation 3.9.

$$hrt(m_\mu, B_\beta, t) \approx 1 \tag{3.9}$$

In stark contrast, as we demonstrated in Section 3.2, Bitcoin miners have no guarantees on how many blocks they would have to wait before the hash-based value of their reward coins makes up for the hashing power they expended to create them.

3.3.3 Equitable Coin Supply Growth

Achieving equitable rewards in this fashion means that miners accumulate rewards that are proportional to their mining power, leaving the number of new coins that can be created at any given moment virtually uncapped. Practically, however, the total amount of computational power invested by miners in the network restricts the growth of the supply of coins, preventing the relative growth rate of the coin supply from spiraling out of control. More precisely, over time, the relative growth rate of the coin supply tends towards the relative growth rate of mining power over time. Consequently, as long as the amount of computational power invested in mining remains stable, the relative growth rate of coin supply slowly converges to zero. Under such constant mining power, the coin supply grows only by a constant number of coins per block similar to Bitcoin's, converging towards a relative growth rate of 0. Linearly growing mining power leads to the same convergence, but at a slower rate.

In Figure 3.5 we illustrate coin supply growth under different mining power growth rates. To plot this figure, we simulated a simple proportional reward issuance scenario under three different mining difficulty settings. In the first setting, the amount of mining power stays constant, while in the second, the mining power grows by one unit every time period. In the third setting, the mining power is multiplied by 1.025 each time period, denoting a 2.5% increase per period. As we start from a coin supply of zero, the initial relative growth in all scenarios is substantial. However, as time passes, the relative rates of increase in coin supply and mining power converge.

3.4 Summary

In this chapter we introduced a computationally-grounded framework for quantifying and tuning the relationship between miner expenditures and compensation to assess and establish fairness. In this framework, we used hashing power as an objective cost basis for coin creation, and used newly minted coins as an objective measure of compensation, while intentionally avoiding external variables, such as hardware costs or currency exchange rates. We used our framework to demonstrate mining reward inequity in Nakamoto, and to introduce constraints for achieving equitable rewards.

In Section 3.1 we used hashing power as an objective cost basis for coin production, and used newly minted coins as an objective measure of compensation. We then introduce objective coin valuation metrics from both a local miner's and a global blockchain's perspectives. In all metrics we intentionally avoided external variables, such as hardware costs or currency exchange rates.

In Section 3.2 we demonstrated the inequity in state-of-the-art PoW mining reward schemes using Bitcoin as a primary example. We quantified the effects of winner-takes-all lottery dynamics on Bitcoin, highlighting the significant discrepancy in miner rewards for relatively small and late miners. We showed that small miners and miners who participate during periods of relatively more expensive coin creation costs face several disadvantages.

Lastly, in Section 3.3 we presented our constraints for achieving equitable rewards. We argued that the stability of the hashcap-to-coinage ratio is akin to equity of rewards issued to miners who participate at different times. Similarly, we argued that the stability of the hash-time-to-issuance metric, which requires a more continuous reward issuance schedule for miners of all sizes, enforces equity in compensation delays. Furthermore, we showed that when adhering to such constraints, coin supply growth dynamics exhibit notably different behaviors from constant rewards, such that relative coin supply growth becomes proportional over time to relative mining power growth.

In the next chapter we present our Reward-All protocol, which overcomes the reward inequities of Nakamoto.

Chapter 4

System Design

The main challenge arising from Chapter 3 is to satisfy our equitable reward constraints while providing the same security and performance guarantees of Nakamoto. To accomplish this, we designed Reward-All Nakamoto Consensus (Reward-All) to provide an incentive model with equitable rewards while retaining Nakamoto's Proof-of-Work block creation. We leverage the analyses performed by prior work on security of block production in Nakamoto, and focus our efforts on specifying and analyzing reward issuance in Reward-All. However, rewarding an unrestricted number of miners in a permissionless setting is not straightforward.

We tackle this issue by allowing each miner to independently maintain an authenticated record of its own mining attempts, and only present a proof of the contents of that record when spending rewards as a blockchain transaction. However, such a record cannot be arbitrarily created, and must be a trustworthy account of the computational resources utilized by a miner towards extending the canonical chain. Furthermore, this record must be provided in a compact manner when issuing rewards in order for the system to remain efficient.

In this chapter, we detail our Reward-All design. Initially, we overview the high-level composition Reward-All in Section 4.1. Subsequently, we detail how mining is performed in Reward-All, how mining work in Reward-All is proven, and how Reward-All awards coins to miners, in Sections 4.2, 4.3, and 4.4 respectively. We then analyze the soundness and completeness of our approach in Section 4.5. Lastly, we summarize this chapter in Section 4.6.

4.1 Architectural Overview

We implement a variant of the Nakamoto mining process in Reward-All, while introducing two new sequential steps that must be performed after mining for a miner to spend its coin rewards in the ledger. The first post-mining step is called **Smelting**[1], which is the process by which the miner constructs a proof that it had performed some amount of mining work. The second post-mining step for a reward to be confirmed by the network is called **Minting**, whereby the miner broadcasts the smelted proof as a transaction in the network. The consensus process essentially follows the same procedure as in Bitcoin's Nakamoto, and is explained alongside our mining process. However, we concentrate on shifting the energy utilization of Proof-of-Work towards coinage, such that the majority of energy usage contributes towards currency minting, while a relatively smaller amount goes exclusively towards establishing consensus on the ledger contents. This means that more coins are created as more computational resources are spent by miners.

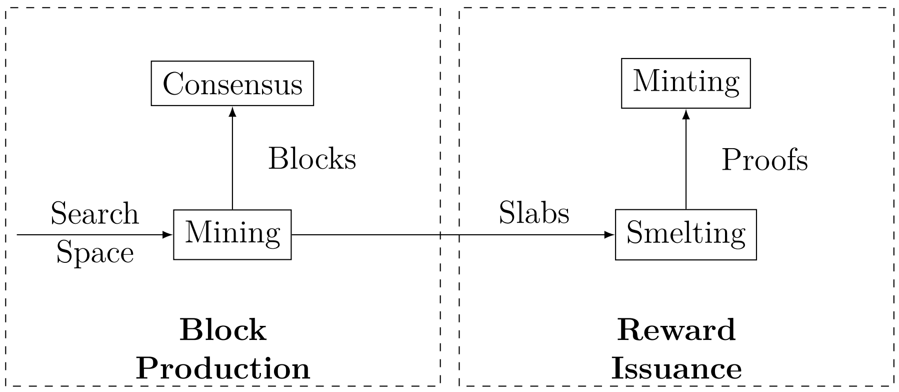


Figure 4.1: Component diagram of Reward-All Nakamoto Consensus.

We introduce Reward-All in a top-down approach, whereby the data and processes that comprise Reward-All, illustrated in Figure 4.1, are gradually broken down and more finely explained. We begin by presenting overviews of Reward-All's two main modules, and of the analysis methods applied to understand their benefits and limitations. We start with Reward-All's block production in Section 4.1.1, briefly touching upon Reward-All's specific mining search space and block structure. Subsequently, we transition to the crux of the Reward-All's

[1]The name smelting was chosen as a metallurgic metaphor, as it follows mining.

reward mechanism in Section 4.1.2, introducing slabs, and how they are used to drive the minting process.

4.1.1 Block Production

Mining for a block in Reward-All is almost identical to mining in Nakamoto, where miners explore a search space of nonces. The miner's search objective is to solve a cryptographic Proof-of-Work hashing problem in order for a proposed block to qualify as the next valid block. However, while the end result of Reward-All's consensus process is the same as that of Nakamoto, many additions exist in Reward-All's supporting components that enable an equitable reward issuance process. Primarily, block production in Reward-All additionally enables tracking each miner's hash calculation expenditures. We provide a high-level comparison between block production in Nakamoto and Reward-All in Table 4.1.

	Nakamoto	**Reward-All**
Consensus	Nakamoto longest-chain protocol	
Blocks	Single chain with periodic difficulty adjustment and restricted block size	
Mining Output	Used to create blocks	Block creation and local mining attempt tracking
Search Space	Nonce	Nonce + Sequence Block Number + Reference Block Number

Table 4.1: Block production comparison between Nakamoto and Reward-All. Consensus and block regulation are identical, while the mining outputs and search spaces differ, whereby additional metadata is used to locally track mining attempts.

Consensus. Essentially, Reward-All implements the longest-chain protocol characterized by Bitcoin's backbone protocol to reach consensus amongst miners, where miners are expected to consider the longest, or heaviest in the sense of most difficult to mine, chain of blocks as the canonical chain.

Furthermore, as in Bitcoin's model, miners are assumed to be interconnected using a peer-to-peer communication protocol where messages, including blocks and transactions, are propagated in a gossip-like fashion. Additionally, no assumptions are made on the smart-contract bearing capabilities of blocks, or what functionality transactions support, for consensus to successfully take place.

Consequently, all existing literature which studies Nakamoto's consensus mechanism in isolation from its reward mechanism is applicable to Reward-All, such as [GKL15, ZP19, DKT$^+$20]. This similarity is leveraged in the analysis of Reward-All carried out in this thesis.

Blocks. Reward-All blocks which can be published to the network to drive the consensus process are akin to Nakamoto blocks. Similarly, for a block to qualify for publication, the result of hashing its plaintext, i.e. its hash, must fall below a target integer value, referred to as the **block target**.

Just as in Nakamoto, the block target in Reward-All must be adjusted to keep the block production rate stable. While we only utilize Nakamoto's periodic block difficulty adjustment procedure in Reward-All, advanced difficulty adjustment methods proposed for Nakamoto, such as those in [IWSK21], are similarly applicable to Reward-All.

Furthermore, blocks are as limited in size in Reward-All as they are in Nakamoto. As a consequence, the entirety of their contents, including headers and transactions, must have a byte-size that is small enough to be propagated between the entire miner network in an amount of time that is relatively small compared to the average block production time.

Mining Output. Mining in Reward-All not only results in blocks, but also outputs **Slabs**, which are the main tool for keeping track of miner expenditures in Reward-All. Aside from this secondary output, Nakamoto and Reward-All mining are almost equivalent.

Primarily, to create slabs, a second target value, called the **Minting Target**, is used in Reward-All's mining process in addition to the block target. When the hash of a block falls below the minting target, the block header data is stored by its miner as a slab. Subsequently, slabs are then used during the reward issuance process to redeem a number of coins that is proportional to how much work the miner has performed, maintaining a stable Hashcap-to-Coinage ratio.

The minting target is a locally set value by each miner. In essence, the main practical restriction is that Reward-All miners are expected to set minting targets which they can find a slab for at least once on average every time a block is found by the network, or face penalties on their reward coins.

Search Space. While mining for a nonce which results in a block, miners in Reward-All have to keep track of additional metadata that enables the slabs which result from these attempts to be uniquely identified and ordered. This ordering supports the process of redeeming a collection of unique slabs for coins.

Consequently, in order to avoid potential losses in mining rewards, a Reward-All miner has to ensure that its mining resources do not search for a Proof-of-Work nonce using the same auxiliary metadata for which it previously found a slab. While managing this metadata during mining adds a layer of complexity compared to the Nakamoto search process, it supports the enforcement of the equitable reward constraints from Section 3.3.

Example. We strengthen the explanation of block production in Reward-All via the following example:

> Alice measures her computer's hashing throughput at 2^{34} hashes per second. Using the blockchain's block interval of 64 seconds, she sets her minting difficulty at $32 \times 2^{34} = 2^{39}$ hashes, so that she expects to create 2 slabs per block on average. Alice starts mining above the latest block. Alice mines for 17 days, at the end of which the network had generated 22950 blocks, and Alice had accumulated 45900 slabs. This example is continued in Section 4.1.2, where Alice inspects the set of metadata of her slabs.

4.1.2 Reward Issuance

Reward-All is an alternative reward mechanism for Proof-of-Work consensus miners designed to stabilize the hashcap-to-coinage ratio (HCR) of the blockchain, and adhere to the equitable reward constraints. The culmination of Reward-All's reward issuance module is the confirmation of an individual miner's rewarded coins in the blockchain, where they become spendable, as overviewed in Table 4.2.

Slabs	Secondary output of mining process
Smelting	Performed to aggregate slabs into compact proofs
Proofs	Used to prove the production and storage of slabs by a miner
Minting	Invoked using proofs to issue rewards

Table 4.2: Reward-All miner reward issuance overview. Slabs are used in the smelting process to create proofs that can be included in blockchain transactions to mint new coins.

Slabs. Slabs are the primary method in Reward-All by which miners locally track their own hash calculation expenditures during block production, as miners can only claim new coins in the ledger by proving the number of unique slabs they have found while mining.

Notably, slabs have to be kept in a miner's storage at least until they are redeemed for coins. However, using a significantly low minting target can create a storage burden, while using significantly high minting target can lead to penalties during minting, as will be explained in Section 4.1.2.

Furthermore, as the minting target determines how easily these slabs can be found, it additionally determines how many coins each slab should be redeemed for. To maintain the stability of the HCR at 1 for example, each slab with a minting difficulty of 2^{32} hashes on average should equate to 2^{32} coins.

Smelting. **Smelting** in Reward-All is the process by which a miner aggregates its slabs into a single proof of the total amount of hash calculations it is expected to have performed. This is the first post-mining step a Reward-All miner performs once it has acquired slabs.

With this proof, a prover is able to convince a verifier, with overwhelming probability, that it has found some number of unique slabs which all have the same minting target.

When mining for physical minerals in the real world, metal ore is extracted, which is comprised of a metal, such as gold or silver, and many other unwanted elements. Using heat, ore is smelted into a pure base metal form, free from unwanted elements. Similarly, smelting in Reward-All involves distilling one of the results of mining in Reward-All, namely slabs, into a basic element that will be later on used in Minting coins. Following the same spirit as metallurgic smelting Reward-All's smelting process does not result in absolute purity. Instead, its outcome is a cryptographic proof which can only be used to ascertain, with overwhelming probability, that a fraction of a miner's work log was calculated. Luckily, this fraction can reach values as high as 99%, but at the expense of more computations and larger proof size.

In each invocation of this proving system, the prover is called by each miner looking to claim a number of new coins as a reward, and the verifier is executed by each node replicating the distributed ledger when validating a block that contains such proofs. To attest to the security of this advantageous approach, its soundness and completeness are quantified using the well-studied characteristics of operating curves with zero rejection tolerance in Section 4.5.

Proofs. Slab aggregation proofs in Reward-All are the main output of the smelting process. Reward-All's slab proving system is based on minimal cryptographic assumptions and built with simplicity, resulting in every proof only being comprised of a constant number of random samples of the slabs it aggregates.

In the implementation of Reward-All presented in this thesis, the sizes of proofs depend on the number of unique slabs claimed to have successfully met the minting target, and on the number of samples of such attempts revealed by the prover. These sizes grow logarithmically in proportion to the number of slabs, and linearly in proportion to number of samples.

However, because of the sampling-based approach employed, proof sizes constitute a transaction cost overhead in Reward-All. These costs, along with their effects on miners, are quantified in Section 6.2.1, and are the primary motivation for creating an efficient proving system.

Minting. Minting is the process whereby miners redeem their slab aggregation proofs for spendable coins in the blockchain. This process is carried out through publishing slab aggregation proofs in blockchain transactions.

Notably, these proofs can be practically incrementally provided and rewarded in Reward-All across multiple transactions. Depending on the number of samples provided so far which correctly meet the minting target, an appropriate fraction of the coins claimed for reward becomes incrementally spendable by the prover.

However, unlike in Nakamoto, adhering to mining on the latest block in Reward-All is motivated using a direct penalty, which is deducted during minting once the first proof sample is published.

Example (Continued). Continuing our example from Section 4.1.1:

> Alice, wanting to start spending her mining reward coins, performs the smelting procedure using all of her 45900 slabs. The procedure outputs a proof containing 1730 sampled slabs, which is sufficient to allow her to spend 95% of her coins. Because the blockchain prices its coins in Kilohashes, such that HCR $= 2^{10}$, Alice expects to receive[2] 2^{29} coins per slab. Alice inputs her proof into the minting procedure, and opts to publish all proof samples simultaneously at a moderate transaction fee. The minting procedure outputs a large transaction, which Alice then immediately broadcasts into the network. In block 515706, after 7 new blocks had been created, Alice's transaction is confirmed, and the breakdown of the total new coins in her account is as follows: Alice was to be credited with 95% of 45900×2^{29} coins, equal to 23,410,256,117,760 coins (a large number!). Because her proof only used reference block number 515699 as the most recent block, her minting transaction was penalized with $6 \times 2 \times 2^{29} =$ 6,442,450,944 coins. Alice's minting transaction resulted in a total of 23,403,813,666,816 coins being awarded to her after the penalty is deducted from the proven amount. Alice rejoices ☺.

[2] recall that her minting difficulty is 2^{39} hashes per slab.

Hypothetically, if Bob were to acquire and operate the same mining equipment as Alice is using, and mirror her decisions in choosing when to mine, smelt, and mint, Bob would receive the exact same number of reward coins, as the issued rewards are not artificially restricted as in Bitcoin.

4.2 Mining Blocks

In this section we dive into the details of the mining process, embodied by the procedures in Algorithm 1, where we present three main functions.

Algorithm 1: Reward-All mining.

```
function mine(B)
    /* retrieve next seq and ref block numbers */
    $(N_{\text{seq}}, N_{\text{ref}}) \leftarrow nextSeqRefPair()$;
    /* iterate through nonce until target met */
    for $nonce \in \{0, 1, 2, ...\}$ do
        $header \leftarrow createHeader(N_{\text{seq}}, N_{ref}, nonce, B)$;
        $h \leftarrow H(header)$;
        if $h < T_b$ then
            /* store header for minting */
            $storeMintingTargetHeader(header)$;
            /* return header which qualifies block for publication */
            return $header$;
        else if $h < T_m$ then
            $storeMintingTargetHeader(header)$;
            /* continue mining */
            return $mine(B)$;
function nextSeqRefPair()
    /* return first unused (seq,ref) pair */
    for $N_{seq} \in \{0, 1, 2, ...\}$ do
        $N_{\text{ref}} \leftarrow highestRefNumAtSeq(N_{\text{seq}})$;
        if $N_{ref} < mainChainLength$ then
            return $(N_{seq}, N_{ref})$;
function assertBlockIsValid($B_N$)
    assert$(H(B_N.header) < T_b)$;
    assert$(B_N.header.ancestorHash == H(B_{N-K}))$;
    assert$(B_N.header.parentHash == H(B_{N-1}))$;
    assert$(validTransactions(B_N))$;
```

The main procedure, *mine*, takes as input a block and attempts to find a PoW that would either append it to the main blockchain, or reward the miner for its attempt. The *nextSeqRefPair* procedure returns the appropriate metadata that can be used to search for blocks and slabs. The *assertBlockIsValid* procedure verifies that a block meets the requirements for extending its parent chain.

Unlike Nakamoto, mining in Reward-All produces two kinds of output, each of which is designed to satisfy its own objective. The first objective is block proposal, which is enabled by a classic Nakamoto Consensus process using PoW. Section 4.2.1 details the block proposal process, which depends on the output of blocks from mining. The second objective is work logging, whereby miners individually log the number of hashing computations each has performed for itself while attempting to propose blocks. Section 4.2.2 presents work logging, which is enabled by slab mining outputs.

4.2.1 Block Proposal

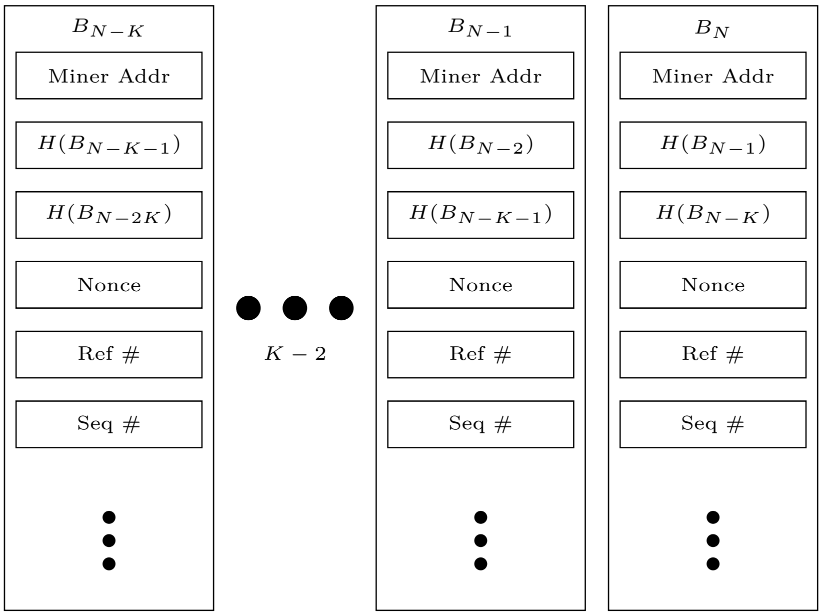


Figure 4.2: Illustrated Reward-All block headers. For $K = 1$, the chain commitments would collapse to those of a regular blockchain, where each block only commits to its immediate parent, as $H(B_{N-K}) = H(B_{N-1})$ for all N. For $K > 1$, the ancestor block hash would refer to a block B_{N-K} that precedes the immediate parent block in the chain.

Similar to mining in Nakamoto, miners in Reward-All collect transactions into blocks, and proceed to solve PoW puzzles derived from the headers of these blocks to win the right to append their blocks onto the blockchain. However, Reward-All blocks are structurally different from those of Nakamoto, but the overall consensus process remains mostly the same.

1. First, while a Nakamoto block only commits to its immediate parent, a Reward-All block also includes a commitment to its K^{th} ancestor, where K is a configurable Reward-All parameter. As we illustrate in Figure 4.2, for a block B_N, the commitment to the parent block is denoted by $H(B_{N-1})$, and the commitment to the ancestor is denoted by $H(B_{N-K})$. We leverage this subtle, yet significant, difference to handle rewards issued for mining stale blocks more leniently than in Nakamoto, as we discuss in Section 4.3.

2. Second, to solve said PoW puzzles, miners repeatedly compute the hashing function H over block headers, each time with a different nonce, until they reach a hashing output which meets the *block target* T_b. However, miners do not only cycle through nonces, but also cycle through two additional integers, according to the criteria we explain in Section 4.2.2. Nonetheless, to propose blocks, miners search for a *header*, that when hashed, results in a value $H(header) < T_b$.

As in Nakamoto, the expected number of search attempts required to find a header that meets the block target is defined as $D_b = T_{MAX}/T_b$, where T_{MAX} is maximum (easiest) target value. Once such a header is found, it is published by the miner, along with its corresponding block, and appended to the blockchain by the network.

Furthermore, while not shown in Algorithm 1, when a miner is presented with two conflicting blockchains, the miner is prescribed to follow the chain with the most accumulated PoW. This longest-chain protocol specification serves as the backbone of the security analysis performed in Chapter 5 against block withholding attacks on Reward-All.

However, unlike in Nakamoto, the miner who wins the right to propose the next block does not receive a special block reward for doing so. Only all the transaction fees paid in that block are credited to the block's miner. We retain this fee mechanism to effectively motivate non-empty block publication and further motivate mining on the latest block.

4.2.2 Work Logging

To receive compensation for mining, a Reward-All miner must choose its own *minting target* T_m value prior to commencing with mining. As per Algorithm 1, this value determines the threshold past which the miner retains a slab as a record of its PoW searching attempts. While not shown in Figure 4.2, the minting target is part of the Reward-All block header.

Consequently, upon finding a *header* that meets the minting target, such that $H(header) < T_m$, the miner stores it as a slab. Similar to blocks, the expected number of search attempts required to find a slab that meets the minting target is defined as $D_m = T_{MAX}/T_m$. This means that D_m is not an exact measure of how many hashing attempts were made to find a single slab, but is rather an increasingly more accurate estimate as more slabs are considered.

More precisely, for a collection of n slabs with minting difficulty D_m, the actual attempts required to produce all n slabs are represented by n random samples $\{X_1, ..., X_n\}$ independently drawn from a geometric distribution X with a mean of D_m, such that:

$$X \sim \text{Geom}(D_m) \tag{4.1}$$

$$\bar{X_n} \equiv \frac{X_1 + ... + X_n}{n} \tag{4.2}$$

As the number of samples n approaches infinity, the mean of the samples, denoted by $\bar{X_n}$ approaches the mean D_m of the geometric distribution from which the samples are drawn, while $\bar{X_n}$ follows a normal distribution for sufficiently large n per the central limit theorem.

Furthermore, as we briefly mentioned in Section 4.2.1, in Reward-All, miners search for slabs by modifying two fields in addition to the nonce: the **reference block number**, and the **sequence block number**. Miners must use these two fields in order to organize slabs in a way that allows them to be efficiently aggregated into a proof, and distributes their rewards equally across each block in the chain.

First, miners set the reference block number such that slabs that meet the minting target are uniformly distributed over a series of referenced blocks. Second, miners utilize the sequence number field, such that:

1. No two slabs which meet the minting target and use the same referenced block number share sequence numbers.
2. The smallest non-negative available sequence number is always used first for new slabs.

Under these constraints, miners accumulate slabs that can be used to later prove the amount of work that has been performed, as we describe in Section 4.3.

Lastly, each miner should set a minting target that it can expect to meet meet at least once per block arrival. This is because a miner should meet the minting target on average at least once per block in order to keep up with the blockchain growth rate. Doing so allows the miner to avoid the minting penalty that will be described in Section 4.4. Once the miner finds it necessary to spend its mining rewards, it uses its accumulated slabs to follow the steps we describe in Section 4.3.

4.3 Smelting Proofs

Algorithm 2: Reward-All smelting.

```
function smelt(Ref, Seq, P)
    $headers \leftarrow []$;
    for $b \in \{Ref.start, Ref.start + 1, ..., Ref.end\}$ do
        for $s \in \{Seq.start, Seq.start + 1, ..., Seq.end\}$ do
            $header \leftarrow fetchStoredMintingHeader(b, s)$;
            $headers.append(header)$;
    $tree \leftarrow createMerkleCommitmentTree(headers)$;
    $proofs \leftarrow []$;
    for $i \in \{0, 1, ..., P\}$ do
        $samplePosition \leftarrow H(i||tree.root) \mod |headers|$;
        $merkleProof \leftarrow tree.inclusionProof(samplePosition)$;
        $proofs.append(merkleProof)$;
    return $(tree.root, proofs)$;
```

A miner, in possession of a collection of slabs as a result of mining, can create a proof, using the *smelt* procedure from Algorithm 2, that convinces a verifier with overwhelming probability that the miner has performed some number of hashing calculations. The proof is comprised of a random subset of slabs sampled from the set of slabs that the miner is proving to have created.

For a miner to successfully smelt by following Algorithm 2, the collection of slabs it aims to utilize must have a specific structure, which we describe in Section 4.3.1. Subsequently, given an acceptable collection of slabs, we explain the second part of Algorithm 2, whereby miners create the proof of their work logs properties, in Section 4.3.2.

4.3.1 Proving Conditions

Before smelting, a miner must first gather a *uniformly distributed set* of slabs that meet the minting target. This uniform distribution of slabs must be made with respect to the continuous series of the M reference block numbers selected by the miner for smelting. The miner achieves this uniformity by selecting a continuous series of N sequence numbers, such that it knows M slabs for each sequence number, and each of these M slabs must correspond to exactly one reference block number in the selected reference block number series. This results in $N \times M$ unique slabs being selected to smelt, as illustrated in Figure 4.3. No sequence number previously included in a published proof may be selected as to avoid reward duplication.

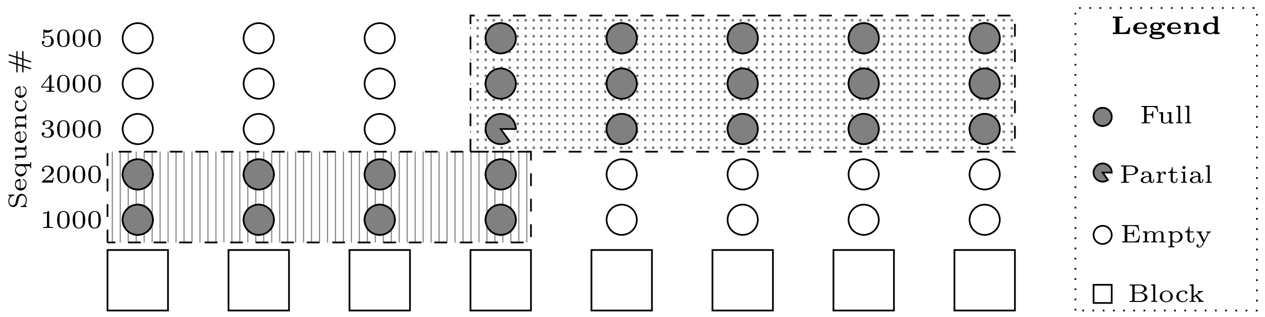


Figure 4.3: Example smelting scenario. The row of a circle determines the sequence number range of its set of headers. The column, or block, determines the reference block number used by all represented headers. The lower-left collection of headers, shaded in vertical lines, can be proven up to any fraction of validity. The upper-right collection of headers, shaded in dots, is 1% invalid, and may fail to be proven valid, as only 85% of the headers with sequence numbers between 2000 and 3000 which refer to the earliest reference block in the proof are valid. This is illustrated by partially shaded circle in the upper-right collection being only 85% filled.

Furthermore, the ancestors referenced in all selected slabs for smelting must still be part of the main blockchain. This entails that if a block re-organization replaces a suffix that is longer than K blocks, then all stored slabs which point to ancestors affected by the re-organization must not be selected. Under such rules, all Reward-All miners are penalized when blocks older than the K-long suffix of the blockchain are replaced, while the proposers of the K-long suffix blocks lose their transaction fee earnings.

Moreover, miners should opt to select a series of reference block numbers to smelt that ends with a block number that is as recent as possible. This is motivated by the fact that proofs which use reference block number series that end with blocks older than the most recent block prior to proof confirmation are penalized, as briefly mentioned in Chapter 4.1. This penalty grows with respect to how old the most recent reference block in the series is, and with respect to how many sequence numbers are used in smelting. Specifically, if a miner's proof is included in block i while the proof contains block j as the last used reference block, where $j < i$, then a penalty of $\rho \times (i - j - 1) \times N$ coins is applied to the amount rewarded from the proof's publication, where ρ is a preset constant in the system. Consequently, this motivates miners to continue mining, repeat smelting, and update the transaction that publishes their proof, until their proof publication transaction is confirmed. In Figure 4.4 we illustrate an example of this penalty when enforced on a miner that published a minting transaction with a two-block reference-confirmation difference (i.e. $i - j - 1 = 2$) and $\rho = 1$.

4.3.2 Proof Creation

Finally, the miner first creates a Merkle-tree commitment of the $N \times M$ slabs selected for smelting. Then, using the root of this tree, the miner derives a deterministic pseudo-random sequence of samples from the selected slabs. Subsequently, for each sampled slab, the miner creates a Merkle-tree inclusion proof for that slab with respect to the created Merkle-tree commitment. The intervals denoting the series of reference block numbers and sequence numbers, the minting target T_m, the Merkle-tree commitment root, along with the inclusion proofs and the slabs they lead to, are then published as a transaction.

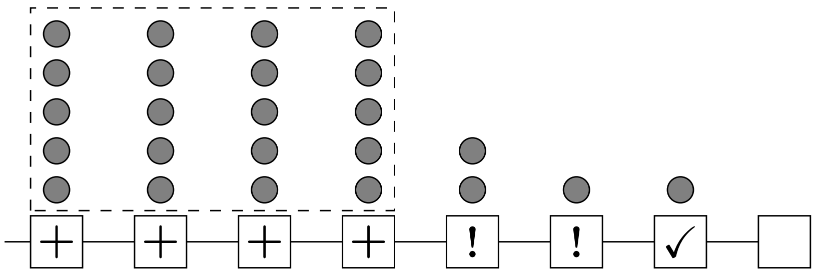


Figure 4.4: Example reward claim scenario from the perspective of a miner. **Squares** represent mined blocks. **Circles** represent weak headers whose reference blocks are represented by the squares below them and whose parent blocks exist in the chain. In this scenario, a miner successfully publishes a minting transaction in the block with the check-mark (✓). The proof in this transaction only confirms the work done using blocks with a positive sign (+) as reference blocks. No work is proven using blocks with an exclamation mark (!) as reference blocks. Consequently, $N \times M = 5 \times 4 = 20$ unique weak headers are used to generate the published proof. As the minting transaction was confirmed three blocks after the last used reference block, a deduction of $\rho \times 2 \times 5$ headers worth of coins is applied to the minted amount as a penalty. With $\rho = 1$, only 10 headers worth of coins are minted.

Smelting in Reward-All creates a computationally-sound statistical sampling proof (argument) with minimal cryptographic assumptions. Aside from the simple implementation advantages this approach offers, the cost of smelting using this approach in practice is negligible in comparison to the cost of mining. This prevents smelting from disrupting mining power. A downside to this approach is that proof sizes can be quite significant. For example, the number of samples required to receive 99% of the coins owed in exchange for the slabs presented is 8828. However, proofs do not have to be revealed in one shot, but can be incrementally revealed across multiple transactions.

In the next section we explain how these proofs are verified as blockchain transactions, and how these transactions are leveraged in Reward-All for reward issuance.

4.4 Minting Coins

In this section we detail Reward-All's minting process, whereby new coins are issued into the decentralized ledger in favor of miners, using Algorithm 3, in exchange for the smelted slab proofs from Section 4.3.

First, in Section 4.4.1 we explain the verification process of smelted proofs, which takes place

Algorithm 3: Reward-All minting.

```
function $mint(B_N, miner, T_m, Ref, Seq, s, treeRoot, proofs, \lambda)$
    $M \leftarrow Ref.end - Ref.start + 1$;
    $N \leftarrow Seq.end - Seq.start + 1$;
    for $i \in \{s, s+1, ..., s+|proofs|\}$ do
        $h_i \leftarrow proofs[i].h_i$;
        assert$(h_i.miner == miner)$;
        assert$(h_i.T_m == T_m)$;
        assert$(blockInChain(h_i.ancestor))$;
        $position \leftarrow H(i||treeRoot) \mod N \times M$;
        $SeqNum \leftarrow position \mod N$;
        $RefNum \leftarrow \lfloor \frac{position}{N} \rfloor$;
        assert$(h_i.ref == RefNum$ and $h_i.seq == SeqNum)$;
        assert$(merkleRoot(position, proofs) == treeRoot)$;
        assert$(H(h_i) \leq T_m)$;
        assert$(h_i.ancestor.num + K \geq RefNum)$;
    $t \leftarrow s + |proofs|$;
    $\theta \leftarrow \sqrt[t]{2^{-\lambda}}$;
    return $\theta \times N \times M \times D_m$;
```

in every copy of the distributed ledger. Subsequently, we describe in Section 4.4.2 the coin reward calculation process, which determines how many coins are issued in exchange for provided proofs, and prescribe an additional penalty calculation procedure, not shown in Algorithm 2, for incentivizing mining on the latest block.

Much akin to minting metallic coins, minting in Reward-All involves transforming a smelted element into transferable coins of known quantities. However, because a Reward-All miner always knows in advance the number of coins it will be granted from the successful publication of its proofs, the Reward-All minting process can be considered as a reward spending, or registration, mechanism, rather than a reward issuance mechanism.

4.4.1 Proof Publication

Once a miner has smelted a proof, it must first publish the information that describes and commits to its collection of slabs. Specifically, the miner must publish a transaction which contains the minting target used, the reference block number range, the sequence block number range, and the Merkle-tree commitment root. Once this transaction is published, any penalty to be deducted from the coins redeemable is calculated, and the maximum number of coins that can be rewarded is derived.

Once a transaction with only the above four pieces of information is published, no reward is yet issued, but the risk of being further penalized for the publication of data related to this proof is removed. However, for this storage step to complete successfully, and for the transaction to be accepted, the sequence number series information is compared with that of the last previously saved entry if it exists. If the last used sequence number, if any, is not less than the first sequence number used in the submitted proof, the proof is immediately rejected. Past this point, the blockchain can begin accepting sample slabs, along with their Merkle-tree inclusion proofs, for verification and incremental coin issuance.

The pseudo-random sequence of slabs to be sampled is deterministically recomputed as done by the prover using the submitted commitment tree root. The samples must be published, and verified, in that exact sequence, across one or more transactions.

Notably, to verify a minting transaction, a verifier deterministically computes the expected sequence number and reference block number for each sample. This is done only using the sample's position in the proof, and the stored sequence number series and reference block number series information for the proof. This entails that all slabs are expected to be unique, such that reuse of duplicate slabs in smelting would lead to proof rejection. Consequently, we omit these two pieces of information from the Merkle-inclusion proofs.

Furthermore, for each sampled slab, the verifier asserts that the identity of the miner of the slab is the same as the prover. This is necessary to prevent different miners from laying claim to the same set of slabs.

Lastly, for each sampled slab, the ancestor hash is verified to refer to a block B_N, such that $N + K$ is greater than or equal to the reference-block number of the slab, and B_N is part of the blockchain the proof transaction was included in. This is done to enforce the penalty associated with block reorganizations deeper than K blocks mentioned in Section 4.3.

4.4.2 Reward Calculation

As the verifier validates more sampled slabs according to the generated sequence, it grants more coins to the balance of the prover. We denote by θ the fraction of the $N \times M$ slabs that the verifier is confident are correct with overwhelming probability.

Specifically, let Λ be a predefined security parameter in the system, let s be the number of sampled slabs successfully validated so far, and let $W = N \times M \times D_m$ be the expected number of mining attempts claimed by the prover. We express in Equation 4.3 the probability $P(s, \theta)$ of drawing s random valid sampled slabs from a collection with a fraction of validity θ.

$$P(s, \theta) = \theta^s \tag{4.3}$$

Given that the verifier has so far validated the first s random slabs from the pseudo-random sequence of samples, the verifier calculates the maximum value of θ such that Inequality 4.4 holds true.

$$P(s, \theta) \leq 2^{-\Lambda} \tag{4.4}$$

Finally, the verifier allows up to $\theta \times W$ coins to be spent by the miner, and as the miner reveals more samples, this allowance is increased appropriately. Since miners with the ability to perform more work, increasing either $N \times M$ or D_m, can reach higher values of W, they can attain more rewards, allowing us to provide direct proportionality between mining power and coin rewards in Reward-All.

In Figure 4.5 we plot the minimum number of samples s required for each value of θ. Notably, s and θ are independent of W. However, when smelting proofs that claim values of $\theta \times W >> 2^\Lambda$, the expected amount of computations required for successful smelting using invalid data is on the order of 2^Λ. Consequently, only values of Λ, such as $\Lambda = 128$, which result in a number of computations 2^Λ that is infeasible to perform in practice are considered in this dissertation.

Alternatively, and without smelting, the full slab collection can be submitted for full valida-

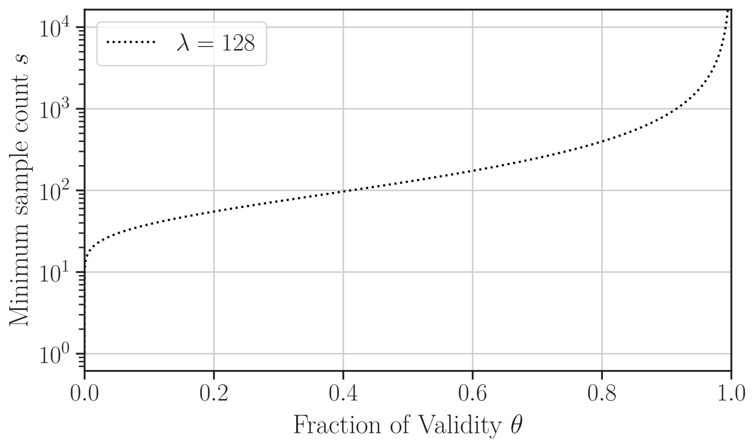


Figure 4.5: Semi-log plot of $s = \frac{\log 2^{-\lambda}}{\log \theta}$, such that $\theta^s = 2^{-\lambda}$, for $\lambda = 128$ and $0 < \theta < 1$. To reach $\theta = 0.5$, at least $s = 128$ samples are required, and for $\theta = 0.8$, $s = 398$ samples must be verified.

tion without any statistical sampling. In this case, the verifier runs all of the validations in Algorithm 3 on the full set of $N \times M$ slabs submitted. We discuss the feasibility and appropriateness of doing so in Chapter 6.

Notably, our mining, smelting, and minting processes enables an unrestricted number of miners to receive and spend their rewards when necessary, albeit after paying the necessary transaction fees. While the last minting stage may seem like a bottleneck, miners which do not broadcast their minting transactions, but continue to accumulate slabs, do not suffer any losses in their rewards. Consequently, our novel construct enables miners to partake in a fee-based priority queue for access to their owed rewards.

4.5 Proving System

In principle, the statistical sampling argument we introduced for reward validation in this chapter is based on constructing a sampling plan with a strict operating characteristic curve [Wal45] where no defective samples are tolerated. In this section, we outline the soundness and completeness guarantees of this approach.

Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation, and let $L_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ be the language defined by R. It is said $(x,y) \in R$, and $y \in L_R(x)$ iff:

1. $x = (B, \mathit{miner}, T_m, \mathit{Seq}, \mathit{Ref}, \mathit{merkleRoot}, \theta)$ represents a tuple of a blockchain B, a miner identity *miner*, a minting target $T_m \in \mathbb{N}$, a pair of sequence numbers $\mathit{Seq} \in \mathbb{N} \times \mathbb{N}$, a pair of block reference numbers $\mathit{Ref} \in \mathbb{N} \times \mathbb{N}$, a Merkle-tree root *merkleRoot*, and $\theta \in [0, 1)$.

2. $y = (\overrightarrow{\mathit{blockHeaders}}, \overrightarrow{\mathit{merkleProofs}})$ represents a pair of lists, where $\forall s \in \{\mathit{Seq.start}, .., \mathit{Seq.end}\}$, $\forall r \in \{\mathit{Ref.start}, .., \mathit{Ref.end}\}$, with probability $\beta \geq \theta$, $\exists b \in \overrightarrow{\mathit{headers}}$, and $\exists p \in \overrightarrow{\mathit{merkleProofs}}$, such that b is a valid block header, and p is an acceptable Merkle-tree proof, as described in Algorithm 3.

Essentially, the procedures we presented in Sections 4.3 and 4.4 describe how a computationally-bounded prover P, the miner, can convince a verifier V, the blockchain, that for a given x the prover knows a $y \in L_R(x)$. This relation essentially captures the set of valid proofs that can be created following the procedures from this chapter.

To connect this relation to Reward-All's proving system, we first describe a simpler interactive version of the proving system used in prior sections. We refer to this simple system as π:

1. $P \rightarrow V : x$

2. $P \leftarrow V : c \in \{1, ..., |\overrightarrow{\mathit{blockHeaders}}|\}$

3. $P \rightarrow V : b \in \overrightarrow{\mathit{blockHeaders}}, p \in \overrightarrow{\mathit{merkleProofs}}$

4. $P \leftarrow V$: *accept* iff the response corresponds to the challenge and passes the validations in Algorithm 3.

Using π, we incrementally build up and analyze the non-interactive argument protocol from Sections 4.3 and 4.4 in three steps. First, let π^t be a protocol which accepts iff t sequential iterations with independent randomness of π all accept, noting that $\pi^1 = \pi$. Then, using the security parameter Λ, we construct the protocol π^t_Λ, which accepts iff π^t accepts and $\theta^t < 2^{-\Lambda}$. Finally, let Π^t_Λ be the result of applying the Fiat–Shamir heuristic to π^t_Λ. Note that Algorithm 3 releases a fraction θ of coins to a miner iff running Π^t_Λ on its input and θ leads to *accept*.

The probability that the randomly selected challenge c by the verifier corresponds to a block header that the prover had correctly computed is equal to the fraction β of correctly computed headers by the prover. It then follows that the probability that a verifier accepts in π^t is always equal to exactly β^t. Similarly, the probability that the verifier does not accept in π^t is equal to $1 - \beta^t$.

Soundness Error. A lapse in the soundness of the π^t verifier's conviction occurs if it accepts when $\beta < \theta$, i.e. when the verifier is convinced that the prover knows at least a fraction θ of the headers, while it only knows a smaller fraction β. The probability that this occurs in π^t, i.e. the soundness error of π^t, is equal to β^t. In the case of π^t_Λ, and Π^t_Λ, there is an upper-bound of $2^{-\Lambda}$ on the soundness error. This is because the verifier does not accept in any case where $\beta^t \leq \theta^t < 2^{-\Lambda}$.

Completeness Error. A gap in the completeness of π^t occurs when the verifier does not accept despite the honest prover having $\beta \geq \theta$. A prover will fail to convince a π^t verifier that it knows $y \in L_R(x)$ with probability $1 - \beta^t$, which is the probability the verifier sends a challenge that the prover cannot answer. This result extends directly to Π^t_Λ.

4.6 Summary

In this chapter we presented Reward-All's novel system design, beginning with an overview of its architecture, followed by a detailed description of each process and its data elements. Our design of Reward-All minimally modifies Nakamoto to provide a novel incentive mechanism whereby an unrestricted number of miners can receive rewards that are directly proportional to their individual mining powers.

In Section 4.1, we first briefly introduced how block production in Reward-All is akin to that in Nakamoto, enabling the same consensus process, but with differences that allow miners to track their hashing calculations. Subsequently, we described from a high level how mining rewards are issued in Reward-All, where miners receive coins in exchange for publishing proofs of their hash calculations.

In Section 4.2 we presented Reward-All's mining process and described its main mining algorithm. After detailing Reward-All's modified block structure, we described how Reward-All's work logging takes place, enabling each miner to retain slab data that tracks the number of hash calculations performed.

Subsequently, we described Reward-All's smelting process in Section 4.3. Using its slab data, a miner follows our smelting algorithm to derive a proof of the total amount of hashing it has performed while mining. Our smelted proofs guarantee with overwhelming probability that a target fraction of the claimed amount of hashing computations were performed by the miner.

Moreover, we detailed Reward-All's minting process in Section 4.4, whereby miners incrementally spend coins in the ledger in exchange for publishing their smelted proofs using one or more blockchain transactions. Using a penalty-based approach, we diminish the value of spendable coins from Reward-All miners that don't prove they have performed hashing calculations to mine on the most recent block.

Lastly, in Section 4.5 we demonstrated that Reward-All's coin issuance mechanism offers computational soundness and completeness under reasonable assumptions and parameters. We presented an analysis based on operating curves which quantified the security properties of coin minting in Reward-All.

In the next chapter, we analyse the security of our protocol against block-withholding attacks, the strongest threat to Proof-of-Work blockchains.

Chapter 5

Block Withholding Attack Analysis

Block withholding attacks are mining strategies whereby an adversary deviates from the prescribed protocol by strategically withholding its newly found blocks from publication until a point in time when the adversary can gain an advantage, as discussed in Section 2.3.3. In summary, this class of attacks demonstrates that miners can negatively affect the chain even if their relative share of the total mining power falls below the 51% threshold.

Dembo et al. prove that suffering such an attack is the worst case scenario for consensus in longest-chain protocols, such as Nakamoto, Reward-All and even Proof-of-Stake protocols [DKT$^+$20]. Besides disrupting agreement on the canonical chain, gaming the consensus process through block withholding can be used by the adversary to derive more rewards from mining, in total through double-spending, or relative to other miners [ES14, ZP19].

In this chapter we utilize the framework proposed by Zhang et al. to quantify the resistance of Reward-All against optimal block withholding attacks [ZP19]. We create four different Markov-Decision-Process models for mining in Reward-All, such that solving each model yields an optimal policy for satisfying a certain adversarial objective in the system. In each case, we compare the performance of Reward-All with that of Nakamoto[1]. Our results demonstrate that Reward-All can reach near-ideal resilience against censorship and profitability

[1]Zhang et al. assess the performance of other blockchains in their work [ZP19]. However, their results cannot be used to infer a like-for-like comparison with the Reward-All results presented here as the cutoffs of the block races they model cannot yield significant results for Reward-All.

attacks, equal resilience to Nakamoto in chain quality attacks, and worse resilience than in Nakamoto against double-spending attacks depending on transaction value and mining power.

We first overview in Section 5.1 the parameters we use to capture the Reward-All mining process. Subsequently, we detail in Section 5.2 each MDP state variable, and their intended purpose. Then, in Section 5.3, we describe the actions agents can take in our MDPs, and the conditions under which each action can be taken. Then we present in Section 5.4 the tool we implemented to solve our MDPs and yield the optimal adversarial mining strategies. Afterwards, we present in Section 5.5 the findings of our analysis for each of the four MDPs, and discuss their ramifications for Reward-All. Lastly, we summarize this chapter in Section 5.6.

5.1 Model Parameters

In our analysis, we assume that a single adversarial miner with a fraction $\alpha \in [0, \frac{1}{2})$ of the computational resources available in the network attempts to construct a single withheld fork of the compliant miner's public chain. Our model does not explore adversaries with at least half of the mining power, as such powerful attackers can, given enough time, create chains that are longer and heavier than any chain created by compliant miners.

At the end of every time step in the MDP, a block is mined with probability α by the adversarial miner, and $1 - \alpha$ by the compliant miners. This discretization accurately models block arrivals, since they occur completely independently of one another, with inter-arrival times following an exponential distribution. Once a block is mined, even if by a compliant miner, the adversarial miner immediately learns of its existence, leaving no notion of propagation delay between the adversarial miner and compliant miners. However, a fraction $\gamma \in [0, 1]$ of the compliant miners is set to favor mining on the adversarial miner's chain when presented with two chains of equal lengths. We refer to this fraction of compliant miners as the **rushed miners**, and the remaining fraction $(1 - \gamma)$ as **distant miners**. This allows our model to account for cases where the adversary can propagate its own blocks to a fraction γ of the compliant miners faster than the remainder of the compliant miners. We illustrate the basic transitions that utilize these probabilities in Figure 5.1.

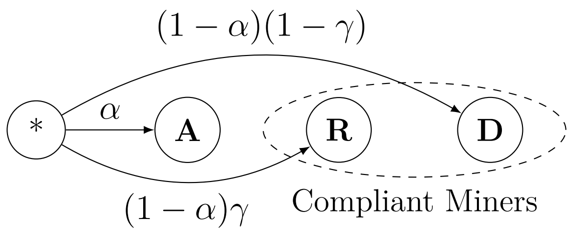


Figure 5.1: Visualisation of base state transition principles in our MDPs. Given a starting state (*), a block is found by the **A**dversary with probability α, by the **D**istant miners with probability $(1-\alpha)(1-\gamma)$, or by the **R**ushed miners with probability $(1-\alpha)\gamma$. Together, rushed miners and distant miners represent the entire set of compliant miners that find blocks with probability $1-\alpha$, where rushed miners represent the γ fraction that receive adversary blocks before compliant miners' blocks, while the distant miners receive compliant blocks first.

Remarkably, as in similar analyses of Nakamoto [ZP19], the expected time between block arrivals, λ, plays no role in our analysis of Reward-All. Despite the fact that the amount of time a Reward-All miner spends mining a block of a certain height determines how many slabs it would find, the expected such number of headers over the long run is equal to $n = \frac{T_m}{T_b}$ per block arrival. Consequently, the fraction of the total coins earned by the network which is earned by the miner on average is equal to $\frac{n \times D_m}{D_b}$, which is the miner's share of the hashing power α.

Furthermore, our analysis only considers **valid unminted coins** as rewards, and is carried out without consideration of the effects of the work proving system employed, or its soundness and completeness as analyzed in Section 4.5. Despite this simplification, our analysis accurately represents the block creation process, as it does not conflict with slab accumulation.

5.2 State Space

At any given time step, the state of the mining process is represented by the following variables:

1. $a \in \mathbb{N}$: The length of the withheld chain mined by the adversarial miner.

2. $h \in \mathbb{N}$: The length of the public compliant miners' chain.

3. $f \in \{$***a**ctive, **r**elevant, **i**rrelevant*$\}$: The potential for a tie-breaking fork race to occur in the public chain.

   - *active*: tie-breaking rules are actively being used by compliant miners to reach consensus and decide which chain fork to follow.
   - *relevant*: the latest compliant miner block is not yet propagated to all compliant miners, and tie-breaking rules are relevant to the adversary's next decision.
   - *irrelevant*: compliant miners agree on the latest compliant miner block, and tie-breaking rules are not relevant to the adversary's next decision.

4. $t_{\mathrm{a}} \in \mathbb{N}$: The number of time steps the adversary has spent mining on its withheld chain while $a \geq K$.

5. $t_{\mathrm{h}} \in \mathbb{N}$: The number of time steps the compliant miner network has spent mining on the public chain while $h \geq K$.

6. $t_{\mathrm{m}} \in \mathbb{N}$: The number of time steps during an active tie the γ-fraction of the compliant miner network has spent mining on the adversary's h-long prefix while $h \geq K$.

The first three variables are used in MDP designs of Nakamoto to model the mining race between the adversary and the compliant miner network. We use these three variables for the same purposes. In Figure 5.2, we illustrate how these variables can model an example scenario.

In addition to the first three variables, we introduce three additional variables for reward calculation in our Reward-All model which are not present in previous MDP models of Nakamoto.

Notably, to precisely calculate the rewards of the adversarial miner, we found that a computationally infeasible to solve state representation is necessary. In that state space, a bit vector is used to keep track of the mining race, such that a 1 is appended to a bit vector each time an adversarial miner creates a block, while a 0 is appended for compliant miners' blocks.

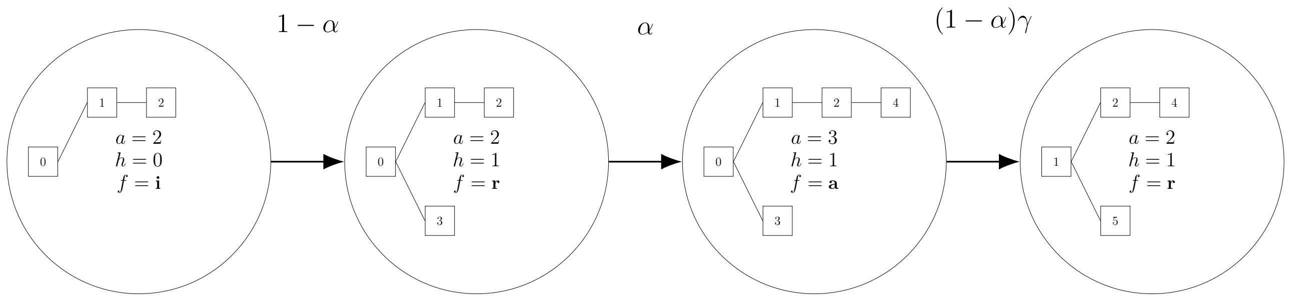


Figure 5.2: A small sub-sequence of the states in the MDP representation of the mining race between the adversarial miner and the compliant minters. Arrows denote transitions between states, where the probability of each transition is denoted in terms of α and γ above each arrow. Blocks are denoted by squares that contain their order of arrival. The leftmost block is accepted by both the adversary and the compliant miners. The upper sequence of blocks represents the adversary's withheld chain of blocks, while the bottom sequence denotes the compliant miners' published chain. In the initial leftmost state, the adversarial miner has 2 blocks in its chain ($a = 3$), while the compliant miners have mined no block ontop of the mutually accepted block ($h = 0$). As $f = \mathbf{i}$, the adversary cannot initiate a tie-breaking fork race. In the subsequent state, the next block to be found belongs to the compliant miners, creating an opportunity for the adversary to initiate a tie-breaking race ($f = \mathbf{r}$) between blocks number 1 and 3. The third state represents one possible next outcome after the adversary initiates the race, whereby the next block is found by the aversary, and the tie-breaking race continues ($f = \mathbf{a}$). In the next state, the rushed compliant miners find a block ontop of the adversary's partially revealed chain, and the adversary is able to carry out another attack.

This bit vector would permit the calculation of how many time steps the adversary has spent mining on each withheld block of a height greater than K. Using this information, the expected rewards at each step in the mining process can be modelled. However, this approach would have only allowed the examination of very short block races that would not yield significant results[2].

Instead, we optimistically allocate all of the mining time the adversary spends mining while $a \geq K$ to be towards producing one block. In one instance, that block is chosen to be block number $K + 1$, and this time is represented using the variable t_a. We present an example of this in Figure 5.3. In another instance, that block is chosen to be block number K, and the need for the variable t_a is forgone. These optimizations allow us to forgo the exponentially growing state space representation in exchange for polynomially-sized state spaces. Still, this comes at the cost of over-rewarding the adversarial miner, and only allows the calculation of

[2]More precisely, it would allow us to only model settings where the value of 2^{C-K} is relatively small, where C is the upper bound on a and h.

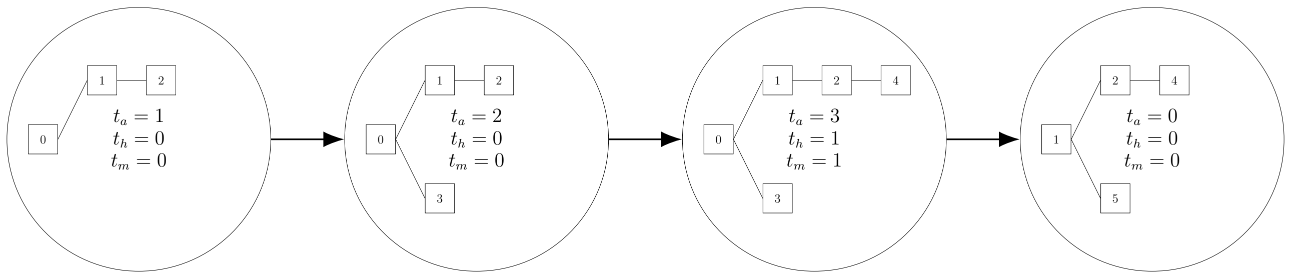


Figure 5.3: Extension of the example presented in Figure 5.2 where each state is annotated with the values of the variables t_a, t_h, and t_m for $K = 1$. In this example, we choose all mining time spent creating a chain longer than K to be attributed to creating the block of height $K+1$. In the first state, the adversary had spent one time unit mining block 2. In the second state, $a = 2 \geq K = 1$ for one additional time unit, where the compliant miners find block 3. Subsequently, as the adversary had started a tie-breaking fork race while $h \geq K$, both t_h and t_m increase by one. Lastly, as the tie was broken in favor of the adversary, and block 1 was confirmed by the compliant miners, the three variables are reset.

upper-bounds on how much of an advantage an optimal adversary could gain. Nonetheless, as K increases, the effect of this overpayment decreases, and the positive differences between the calculated upper-bounds and the true maximum values converge to zero.

In all cases, the variables t_h and t_m are sufficient to calculate the rewards attained by the compliant miners without the need for a bit vector representation. This is because, as will be presented in Section 5.3, there are no state transitions in Reward-All and Nakamoto models which cause a prefix shorter than h of the compliant miner's chain to be confirmed, or removed, and only the entire public chain of length h may be accepted or replaced by the adversarial miner.

5.3 Action Space

After learning of a block's discovery, at the beginning of the next time step, the adversarial miner can instantaneously take one of the Nakamoto actions below:

1. *Adopt:* The adversary discards its current withheld chain, and starts building a new withheld chain based on the latest block found by compliant miners.

2. *Override:* The adversary publishes the first $h + 1$ blocks from its withheld chain, which

leads all the compliant miners to abandon the h blocks they have mined so far in favor of the $(h + 1)$-long prefix of blocks released by the adversary.

3. *Wait:* The adversary keeps mining and waits for another block to arrive.

4. *Match:* The adversary publishes the first h blocks from its withheld chain, leading the γ-fraction of the compliant miners to mine above them, while the remaining miners continue mining on the h compliant miner blocks constructed so far. If at the end of the current time step a block is discovered by the γ-fraction, the remaining miners abandon their h blocks and mine along the γ-fraction. This action is only possible when the last discovered block was mined by the compliant miner network.

These same four actions are the same as those found in models of Nakamoto. This is because Reward-All follows the same block proposal process as Nakamoto, and only utilizes a different reward calculation mechanism.

Similar to MDP designs for Nakamoto, our Reward-All MDPs also enforce a **cutoff** C which restricts the maximum lengths of the chains that can be examined. Because of this, the adversary's ability to *wait* or *match* when either $a = C$ or $h = C$ is revoked, such that the adversary may only *adopt* or, if possible, *override*. While this cutoff reduces the Reward-All MDP state space from infinite to finite, it is instantiated with a sizeable value such that the results it produces are within acceptable bounds of the true values.

Despite the extra dimensions the Reward-All MDP has, relative to the MDPs designed for Nakamoto, we explore races that are longer than those explored in work related to Nakamoto. This is because we implement and analyze the Reward-All MDPs using the C++ API of the Storm Probabilistic Model Checker [3], which utilizes many optimizations [DJKV17].

In Table 5.1, we present the combined state-action transition and reward matrices for three different reward functions, one for each Reward-All MDP that is different from the Nakamoto mining MDP [SSZ16].

[3]`https://www.stormchecker.org/`

Initial State × Action	**Next State**	**Probability**	R_I	R_S	R_C
$(a, h, f, t_a, t_h, t_m) \times$ *adopt*	$(1, 0, \mathbf{i}, 0, 0, 0)$ $(0, 1, \mathbf{i}, 0, 0, 0)$	α $1 - \alpha$	α	α	0
$(a, h, f, t_a, t_h, t_m) \times$ *override*	$(a - h, 0, \mathbf{i}, \omega_K(0, a - h - 1), 0, 0)$ $(a - h - 1, 1, \mathbf{r}, \omega_K(0, a - h - 1), 0, 0)$	α $1 - \alpha$	α	$\alpha \times \psi_K(t_a, a - h - 1) + V_{\text{ds}}(h, \sigma)$	$(1 - \alpha) \times (t_h + (1 - \gamma) \times t_m)$
$(a, h, \mathbf{i}, t_a, t_h, t_m) \times$ *wait* $(a, h, \mathbf{r}, t_a, t_h, t_m) \times$ *wait*	$(a + 1, h, \mathbf{i}, \omega_K(t_a, a), \omega_K(t_h, a), t_m)$ $(a, h + 1, \mathbf{r}, \omega_K(t_a, a), \omega_K(t_h, a), t_m)$	α $1 - \alpha$	α	$\alpha \times \psi_K(0, a)$	0
$(a, h, \mathbf{a}, t_a, t_h, t_m) \times$ *wait* $(a, h, \mathbf{r}, t_a, t_h, t_m) \times$ *match*	$(a + 1, h, \mathbf{a}, \omega_K(t_a, a), t_h, \omega_K(t_m, h))$ $(a, h + 1, \mathbf{r}, \omega_K(t_a, a), t_h, \omega_K(t_m, h))$ $(a - h, 1, \mathbf{r}, \omega_K(0, a - h), t_h, \omega_K(t_m, h))$	α $(1 - \alpha) \times (1 - \gamma)$ $(1 - \alpha) \times \gamma$	α	$\alpha \times \psi_K(0, a)$ $\alpha \times \psi_K(t_a, a - h) + V_{\text{ds}}(h, \sigma)$	0 $(1 - \alpha) \times (t_h + (1 - \gamma) \times \omega_K(t_m, h))$

Table 5.1: MDP transitions and rewards. Each row in the first column contains a set of state-action pairs. The second column contains the set of possible outcomes of all of the state-action pairs of the same row. The third column contains the probability of each outcome being reached. The last three columns contain the rewards of each outcome as a consequence of taking some action in an initial state. For brevity, the functions $\omega_K(u, v) = (\textit{if}\ v \geq K\ \textit{then}\ u + 1\ \textit{else}\ u)$, and $\psi_K(u, v) = (\textit{if}\ v < K\ \textit{then}\ u + 1\ \textit{else}\ u)$ are defined. R_I rewards are *Incentive Compatibility* rewards, and stand for the number of coins the adversary can mint in exchange for mining. R_S rewards are *Subversion Gain* rewards, and stand for the total number of adversary coins confirmed by the honest network, including mining rewards and double-spending profit. σ denotes how many confirmations a recipient waits before considering a transaction as final. $V_{\text{ds}}(h, \sigma)$ denotes how many coins the adversary receives after successfully replacing h honest blocks, each containing transactions to be double-spent. R_C rewards are *Censorship Susceptibility* rewards, and stand for share of mining rewards the adversary can cause the honest network to lose.

5.4 Implementation

We modeled and solved the aforementioned MDPs using the Storm Probabilistic Model Checker C++ API [DJKV17]. We implemented a command-line user interface tool that takes as input the MDP type and its parameters, and outputs the result of solving the generated model and experimentally validating its output using a simulation in which the adversarial miner follows the optimal strategy output by our model checker.

5.4.1 Overview

Given a series of valid parameters, our tool constructs an explicit model of the MDP corresponding to the provided input. For every possible MDP state, our model contains all actions that the adversary may take, along with their expected outcomes and rewards.

Each state is represented by a Storm Expression comprised of Storm Variables which correspond to the aforementioned MDP state variables. The states are enumerated and explored using a breadth-first search, whereby the initial state represents the start of the attack, whereby the adversarial miner and compliant miners agree on the canonical chain and no blocks are withheld. The exploration proceeds to visit each possible state only once, enumerating all possible transitions to and from each state, and storing them using the Storm Sparse Matrix data structure. Our tool then generates a Storm Sparse MDP Model object, which the model checker engine can use to generate a strategy that optimizes or satisfies a certain Storm Property based on the user's input.

Given the optimal adversarial mining strategy by the model checker, our tool then proceeds to run a mining simulation to validate that the results of solving the MDP are reasonably applicable in practice. Reproducing the model checker outcome using simulation is essential to validating that the generated model accurately depicts the real-world dynamics.

5.4.2 Tool Parameters

The tool's command-line interface takes several parameters which determine how to generate, solve, and validate, a single MDP.

Reward Scheme. This command-line parameter determines the mining reward scheme, where a parameter value of `nakamoto` employs winner-takes-all constant mining reward dynamics in the model, while a parameter value of `reward-all` utilizes our Reward-All dynamics.

Target Metric. This command-line parameter determines the metric being optimized. Under both Nakamoto and Reward-All rewards, four metrics can be examined. Depending on which of the below values are used as the parameter value, the tool generates a different model:

- `quality` indicates that a Chain Quality model should be generated, and solved to find the optimal strategy for the Storm Property: `R{"blocks"}max=? [ F "exit" ]`. This property leads the agent to maximize the `"blocks"` rewards it receives before reaching the state labeled with `"exit"`.

- `incentive` indicates that an Incentive Compatibility model should be generated, and solved to optimize the Storm Property: `R{"revenue"}max=? [ F "exit" ]`. This property leads the agent to maximize the `"revenue"` rewards it receives before reaching the state labeled with `"exit"`.

- `subversion` indicates that a Subversion Gain model should be generated, and optimized for the Storm Property: `R{"revenue"}max=? [ LRA ]`. This property instructs the agent to maximize the long-run average of the `"revenue"` rewards it receives.

- `censorship` indicates that a Censorship Resilience model should be generated, and the following Storm Property optimized: `R{"loss"}max=? [ LRA ]`. This property instructs the agent to maximize the long-run average of the `"loss"` rewards it receives.

A fifth model option, `reachability`, is available when using the Reward-All model. This option generates a simple Discrete Time Markov Chain that allows the enumeration of all reachable states under the given parameters.

Stale Forgiveness. This command-line parameter specifies the K parameter used in our model, which allows our model to reward both the adversarial miner and the compliant miners for chains of stale blocks are up to K blocks in length. This parameter may only take non-negative integer values, and is always set to a value of 0 when modeling Nakamoto rewards.

Race Length. This command-line parameter, denoted by the variable T, determines the maximum length of the adversary's chain in the block-withholding attack represented by our model. This parameter is necessary for generating finite models, and restricting the MDP to only the set of states that are reachable during an attack with non-negligible probability. Consequently, only positive integer values may be provided for this parameter.

Adversary Strength. This parameter, denoted by α, determines the adversarial miner's share of the total mining power. The tool accepts integers in the range $[0, 100]$ as input, representing the percentage of the adversary's share.

Adversary Advantage. This parameter, γ, determines the fraction of compliant miners which receive adversary blocks before they receive blocks from other compliant miners, i.e. the percentage of miners which favor the adversarial miner's chain during block races in our model.

Expected Horizon. This parameter is relevant when generating chain quality or incentive compatibility models. Denoted by H, this parameter determines the expected number of block arrivals that are to take place before the attack is terminated by our model.

Transaction Confirmations. For subversion gain models, this parameter determines how many confirmation blocks are required by recipients before the adversary is considered to be able to double-spend a transaction in the generated model. The valid inputs for this parameter are integers in the range $[0, T]$.

Transaction Value. In subversion gain model inputs, this parameter denotes the reward value of each transaction successfully double-spent by the adversary in our model. The value is interpreted as a percentage of the block reward received by the adversarial miner, and may take on any non-negative integer value.

Overpay Adversary. This parameter only applies for Reward-All models when specifying inputs for incentive compatibility and subversion gain MDPs. The parameter represents a boolean flag that denotes whether the adversary is optimistically rewarded in our model for mining more than K blocks in its attack chain. Given a value of `y`, this parameter instructs the creation of a compact model with an advantage for the adversary, while a value of `n` instructs the tool to create an model that accurately rewards the adversary, yet at the cost of exponential growth in model size that is in terms of the value of $T - K$.

Forgive Adversary. This parameter also only applies for Reward-All incentive compatibility and subversion gain models, and specifies whether the adversarial miner is rewarded for all of its mining effort, regardless of whether it successfully overrides compliant miner blocks or not. A value of `y` instructs the tool to generate a model that forgives the adversary for any stale blocks, while a value of `n` instructs the generation of a more accurate model. Similar to the optimistic over payment flag, this flag allows highly optimized models to be generated at the cost of representing an adversarial miner with a more rewarding action space than possible.

Matching Estimate. For Reward-All Incentive Compatibility model inputs, this parameter specifies whether rewards for compliant miners during block races between chains of length K are estimated with every new block arrival instead of at the end of the block race resolution. For γ values of, 0, 0.5 and 1.0, setting this parameter to `y` enables the tool to generate a more optimized model without sacrificing accuracy. For other values of γ, this parameter should be set to `n`, as to accurately model the compliant miners' rewards.

Generate Scheduler. This parameter determines whether the tool will retain the resulting optimal adversary actions in all states and use them to run a simulation which validates the model checker's results. A value of `y` instructs the retention of the adversary, while a value of `n` skips this retention and does not run a simulation.

Choice Labels. This parameter determines whether the tool will explicitly label all choices in the generated model with their action name. A parameter value of `y` creates choice labels in our model, while a value of `n` skips their creation.

5.4.3 Output

On each run, the tool outputs the parameters used to generate and solve our model, a concise description of the generated MDP, the results returned by the model checker, and the validating simulation results if applicable. Furthermore, for the generation, solving, and validation steps, the tool outputs the amount of time taken at each step. The following is an example output transcript:

```
EXTENDED
Incentive Compatibility Analysis
K=140 T=256 A=0.25 G=0.5 H=1000000 O=1 F=1 E=1 S=1 C=0
Formula: R{"revenue"}max=? [ F "exit" ] R[exp]{"revenue"}max=? [F "exit"]
Explicit Model Construction:
Threatening states: 5142686
Seconds: 72
VM: 4525212 RS: 4471156
--------------------------------------------------------------
Model type:  MDP (sparse)
States:  7921761
Transitions:  62745924
Choices:  17471240
Reward Models:  revenue
State Labels:  2 labels
   * exit -> 3 item(s)
   * init -> 1 item(s)
Choice Labels:  none
--------------------------------------------------------------

Model Checking:
Seconds: 973
I(a) = 0.25
1 - I(a) = 0.75
(1 - a) - I(a) = 1.657e-06
Simulation: (Fractional, Timed)
0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25
```

```
0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25
0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25
0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25 0.25,0.25
Seconds: 36
Average Result: 0.25,0.25
```

The first three lines describe the model parameters that will be used to generate the MDP. The subsequent line contains the property that will be optimized for our model, followed by its corresponding equivalent Storm Formula, where the two are almost identical.

The *Explicit Model Construction* line is the heading that denotes the start of our model generation steps. The *Threatening states* count describes the number of states in our model where the adversary can override the compliant miners' chain or attempt to start a block race. The *Seconds* line denotes the number of seconds spent generating our model. The *VM* and *RS* counts describe the memory used by the tool.

The next set of lines describe the generated model. The model type, number of states, number of transitions, and number of choices correspond to the generated model attributes. The reward models line specifies the named MDP transition rewards defined for the model. The state labels line specifies how many special labels are used to mark special states, followed by each label along with the number of states that have that label. In this output, two labels, "exit" and "init", are used to label four states collectively. The choice labels line describes the number of labels used to mark special choices, if any.

The *Model Checking* header marks the start of the checking stage, whereby the Storm Model Checker processes the provided model and property. The number of seconds taken to solve our model is provided, along with the requested results. For our analysis, the adversarial miner's relative share of all mining rewards is printed, followed by the compliant miners' share, and then the advantage gained by the adversarial miner under the optimal strategy compared to what it would have attained by following the compliant strategy.

Lastly, the *Simulation* header marks the start of the simulation phase, and outputs the result of each run. The results are separated by spaces, such that each result is a pair of numbers denoting the adversary's relative reward share, once using the adversarial miner's relative mining power to directly calculate rewards as done in the MDP, and once using the amount of time elapsed mining on each block to calculate rewards. As the two quantities match, this indicates that the MDP accurately models reward allotment. The last two lines denote the number of seconds spent running all simulations, and the average result of all runs combined, respectively.

5.5 Results

Using our tool, we evaluated Nakamoto and Reward-All across the four different metrics proposed by Zhang et al. [ZP19]. In this section we present our results, which demonstrate that Reward-All chain quality matches that of Nakamoto, and that Reward-All can reach nearly ideal incentive compatibility and censorship susceptibility under sufficiently long fork forgiveness depths, but at the cost of a loss in subversion gain resilience that depends on block difficulty and transaction values.

5.5.1 Chain Quality

Since block proposal and tie-breaking in Reward-All follow the same rules of Nakamoto, Reward-All exhibits the same chain quality bounds under adversarial mining as Nakamoto. Nonetheless, we reproduce the methods of Zur et al. to construct an MDP to quantify the chain quality bounds of Nakamoto as in [ZET20]. Performing this analysis was a necessary step towards our analysis of incentive compatibility in Reward-All, which we present next in in Section 5.5.2.

We examine chains of maximum length $C = 256$ at an expected process termination horizon of $H = 10^6$ confirmed blocks. The adversarial miner's objective is to maximize its accepted blocks before H blocks on average are accepted by both the adversarial miner and the compliant miners. Our tool can support larger values of C and H for this analysis, however, they do not yield significantly better results.

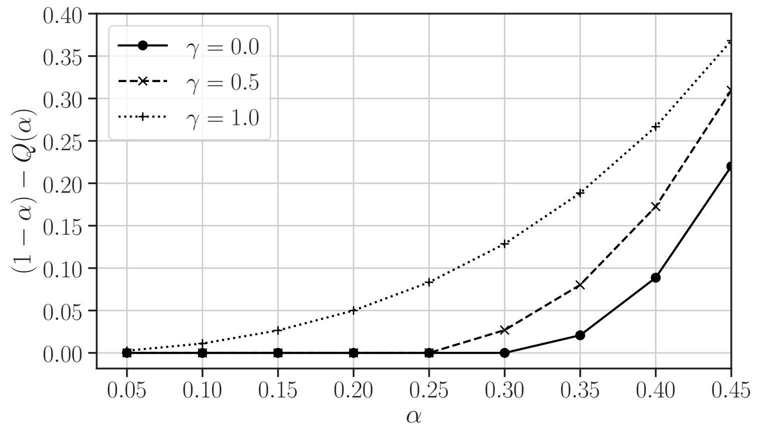


Figure 5.4: Plot of loss in chain quality $(1-\alpha)-Q(\alpha)$ in Nakamoto and Reward-All versus adversarial mining power α, where $Q(\alpha)$ is the chain quality under α, for three values of γ. Both protocols exhibit the same chain quality under the same parameters.

In Figure 5.4, we present the results of our analysis of the loss in chain quality versus the relative mining power of an adversary that maximizes its relative share of blocks, yielding results that are within ± 0.01 of those of [ZP19, ZET20, SSZ16].

Our results for chain quality can be reproduced, for $\alpha \in [0, 0.45]$ and $\gamma \in \{0.0, 0.5, 1.0\}$, using the following command-line parameters in our tool:

```
nakamoto quality 0 256 α γ 1000000 y n
reward-all quality 0 256 α γ 1000000 y n
```

5.5.2 Incentive Compatibility

Despite the adversary's ability to replace compliant miner blocks in Reward-All, the fact that miners of forks of depth at most K are rewarded in Reward-All reduces the *relative* profitability of selfish-mining. To quantify the incentive compatibility of Reward-All, we adapt the approach of Zur et al. in [ZET20] to construct an MDP which models the mining process while accounting for the time spent mining as described in Section 5.2. The adversarial miner's goal in this model is to maximize its R_I rewards, specified in Table 5.1, before H coins on average are rewarded.

We permit forks of length up to $K = 140$ to be rewarded, while modeling chains of maxi-

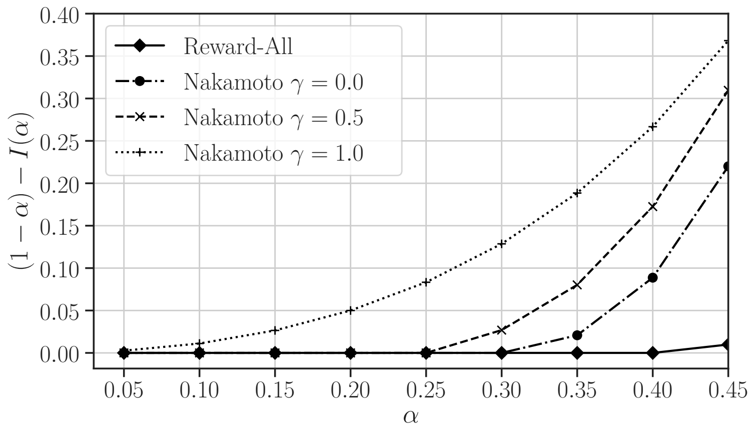


Figure 5.5: Plot of loss in incentive compatibility $(1 - \alpha) - I(\alpha)$ in NC and Reward-All versus adversarial mining power α, where $I(\alpha)$ is the incentive compatibility at α. Reward-All results for all γ are within ± 0.001 of each other for $\alpha \leq 0.45$.

mum length $C = 256$ at an expected process termination horizon of $H = 10^6$ confirmed reward coins. We reach this value of K by performing a binary search for the least value of K for which the optimal policy for maximizing relative rewards cannot attain more than a 0.011 improvement, even when $\gamma = 1$, for adversarial hashing powers of $\alpha \leq 0.45$. Our choices for C and H permit us to create MDPs that lead to solutions that cannot be significantly improved with higher H or C values [ZET20].

In Figure 5.5, we plot the results of our analysis of the loss in incentive compatibility for Nakamoto and Reward-All with respect to the relative mining power of an optimal adversary which maximizes its relative share of mining rewards. Our yielded numerical rewards for Nakamoto are identical to those obtained in the analysis of its chain quality presented earlier. This is because a miner's relative share of the mining rewards in Nakamoto correspond directly in our model to the miner's relative share of the blocks accepted by both the adversarial miner and the compliant miner network. On the other hand, our analysis of Reward-All yields drastically improved results.

This improvement is mainly attributed to the fact that Reward-All rewards compliant miners for the work they have performed towards mining blocks of height less than K within the race. Consequently, despite the adversary's ability to replace compliant miner blocks, the compliant miner rewards are mostly retained. The adversarial miner could only increase its relative share of rewards when it manages to override compliant miner blocks that form a blockchain suffix longer than K blocks. As K increases, it becomes increasingly more difficult for the adversarial miner to mine more blocks than the compliant miners, since $\alpha < (1 - \alpha)$ (i.e. adversary blocks arrive less often than compliant miner blocks do on average).

Our results for incentive compatibility can be reproduced, for $\alpha \in [0, 0.45]$ and $\gamma \in \{0.0, 0.5, 1.0\}$, using the following command-line parameters:

```
nakamoto incentive 0 256 α γ 1000000 y n
reward-all incentive 140 256 α γ 1000000 y y y y n
```

5.5.3 Censorship Susceptibility

Because of the right to claim mining rewards for the mining effort spent towards compliant miner blocks that were later replaced by the adversarial miner, Reward-All exhibits stronger resilience against censorship than Nakamoto. To quantify the censorship susceptibility of Reward-All, we follow the approach of Zhang et al. [ZP19], and construct an MDP that focuses on the time spent mining by the compliant miners. In this MDP, the adversary maximizes the long-run average of the R_C rewards specified in Table 5.1.

We permit forks of length up to $K = 206$ to be rewarded, while modeling chains of maximum length $C = 256$. We reach this value of K by performing a binary search for the least value of K for which the optimal policy for maximizing R_C rewards cannot cause more than a 0.011 loss of profits to the compliant miner network, even when $\gamma = 1$, for adversarial hashing powers of $\alpha \leq 0.45$. Higher values of C do not significantly improve model results, and create larger MDPs which are slower to solve.

This is a significant improvement over censorship susceptibility in Nakamoto, whereby an equally powerful adversarial miner can cause the compliant miner network to lose a portion equal to 0.8182 of its rewards. However, because of the disassociation between block discovery and rewards in Reward-All, these results can be interpreted differently, such that compliant miners in both Reward-All and Nakamoto can lose the same portion of their mined

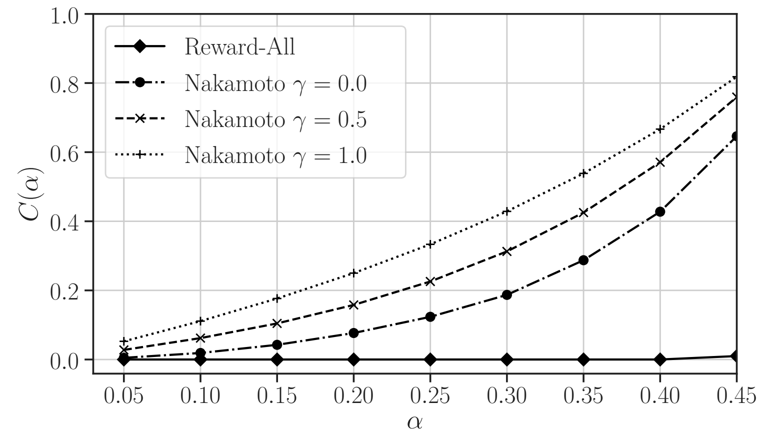


Figure 5.6: Plot of the censorship susceptibility $C(\alpha)$ in NC and Reward-All versus adversarial mining power α. In Reward-All, results for all values of γ are within ± 0.001 of each other for $\alpha \leq 0.45$.

blocks due to an adversarial miner that follows the optimal censorship policy.

Our results for censorship susceptibility can be reproduced, for $\alpha \in [0, 0.45]$ and $\gamma \in \{0.0, 0.5, 1.0\}$, using the following command-line parameters:

```
nakamoto censorship 0 256 α γ y n
reward-all censorship 206 256 α γ y n
```

5.5.4 Subversion Gain

The forgiveness mechanism in Reward-All for K-long forks creates an incentive to always attempt double-spends in the system, as there is no punishment to failing to double-spend when merchants wait for $\sigma \leq K$ block confirmations on transactions. However, because of the equivalence in chain quality guarantees between Reward-All and Nakamoto, the probability of success of a double-spend attempt remains the same in both systems. The goal of the adversary in this MDP is to maximize the long-run average of its R_S rewards from Table 5.1.

We consider block races with a cutoff of $C = 256$ blocks, and a fork forgiveness depth of $K = 206$ blocks. This value of K allows us to evaluate security against double-spending while ensuring that the adversary does not have an advantage of more than 0.011 in either incentive compatibility or censorship susceptibility scenarios, as discussed in Sections 5.5.2

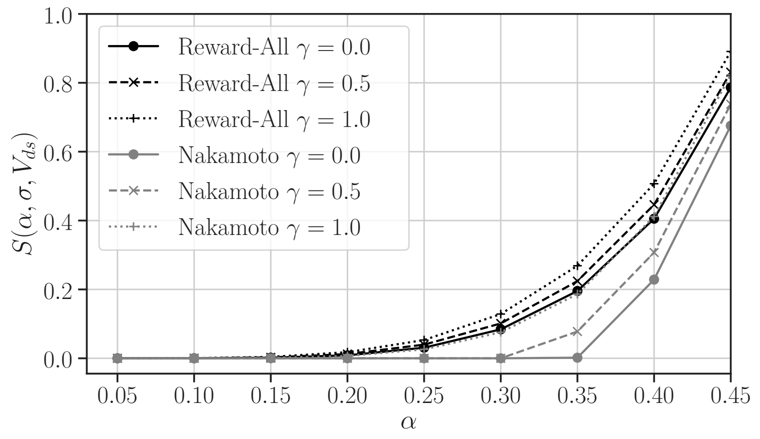


Figure 5.7: Plot of the subversion gain $S(\alpha, \sigma = 6, V_{ds} = 3)$ in NC and Reward-All versus adversarial mining power α.

and 5.5.3. Similarly, higher values of C do not significantly affect our results.

In Figure 5.7, we present the analysis of the profitability of attempted double-spending in Nakamoto and Reward-All. Remarkably, a significant regression in Reward-All compared to Nakamoto in our analysis can be seen. Specifically, for the same value of γ, Nakamoto outperforms Reward-All across all values of α. This is because the adversary incurs no mining reward losses in Reward-All when it fails to produce a withheld chain of length shorter than K and gives up its current forking attempt.

However, the Nakamoto and Reward-All results in Figure 5.7 do not constitute a straightforward comparison. This is because the reward V_{ds} for each successful double-spend is specified in terms of multiples of the **total rewards issued per block**. In Nakamoto, the total rewards issued per block do not increase with how much mining power is invested into the network. In Reward-All, on the other hand, more rewards are granted as the network grows. Consequently, to attain a like-for-like comparison, one must first specify a total hashing power for Reward-All, and subsequently convert the V_{ds} value used in the Reward-All analysis to a comparable value for the Nakamoto analysis.

To elaborate, consider the case where both networks have the same block arrival rate, and the same hashing power of 150×10^{18} hashes per second, which is approximately within the same order of magnitude of the current Bitcoin network hashing rate. Each Nakamoto block would reward 6.25×10^8 units per block, while each Reward-All block would reward approximately 150×10^{18} units on average. For the results presented in Figure 5.7 to be comparable, one would have to assume that 1 Nakamoto coin is equal in value to 24×10^{10} Reward-All coins. If the value of Reward-All coins increases compared to Nakamoto coins, one would have to consider an equally lower V_{ds} value for Reward-All to evaluate and compare double-spending incentives against Nakamoto. Similarly, one would have to consider higher V_{ds} values for Reward-All if its value equally decreases against Nakamoto coins.

Our results for subversion gain can be reproduced, for $\alpha \in [0, 0.45]$ and $\gamma \in \{0.0, 0.5, 1.0\}$, using the following command-line parameters:

```
nakamoto subversion 0 256 α γ 6 300 y n
reward-all subversion 206 256 α γ 6 300 y n y n
```

5.6 Summary

In this chapter we quantified Reward-All's resilience against block withholding attacks, showing its advantages and feasibility under practical parameterizations. Our analysis focused on quantifying chain quality, incentive compatibility, censorship resilience, and subversion gain in Reward-All, as per the definition of Zhang et al. for these metrics [ZP19].

In Section 5.1 we first presented the main parameters for initializing our models. Using two main parameters, we described how we capture block arrival times and adversarial block propagation in our MDPs.

Then, in Section 5.2 we described our MDP state variables, which hold the information required to represent the mining process when modelling each block withholding attack. Primarily, we detailed three new state-variables in addition to those already utilized in analyzing Nakamoto to model attacks in Reward-All.

Subsequently, we introduced the different actions in Section 5.3 that an agent in one of our MDPs can perform, the state transitions these actions create in the MDPs, and the conditions which must be met for each action to be valid.

Next, in Section 5.4, we presented our MDP solving tool, which is based on the Storm Model Checker. We presented how we implemented each MDP model in the tool, and described how to execute each model with different parameters.

Lastly, we demonstrated in Section 5.5 that the reward structure of Reward-All can offer significant advantages in terms of the protocol's incentive compatibility and censorship susceptibility compared to Nakamoto Consensus, without any loss in chain quality. We furthermore quantified the costs of these advantage in terms of the loss in resilience against subversion gain, and elaborated on how the increasing block difficulty in Reward-All's leads to subversion gain resilience improvements.

In the next chapter we evaluate the costs of using our prototype Reward-All implementation.

Chapter 6

Evaluation

In this chapter we evaluate the practicality of Reward-All, comparing its efficacy against that of Nakamoto. We focus on our reward issuance process, quantifying its overheads and advantages in terms of its parameters. To show that Reward-All is feasible, we implemented a full local miner node prototype capable of demonstrating that a permissionless peer-to-peer overlay network of Reward-All miners can run and manage replicas of a cryptocurrency ledger while their rewards are proportional to their computational investment in the Nakamoto consensus process.

First, in Section 6.1 we present our implementation considerations and solutions from a high-level design perspective, leaving low-level details to Appendix A. We introduce our multi-threaded Reward-All implementation architecture, which ensures different threads do not block each other's progress, and present additional components designed to overcome the technical challenges of realizing Reward-All mining, smelting, and minting in practice. Then, in Section 6.2, we present the overheads associated with the sizes, in bytes, of the smelted proofs in Reward-All, and the trade-offs between waiting time and rewards received that miners can exploit under different conditions. Furthermore, we derive the coefficient of variation of mining rewards in Reward-All, and compare it to that of Nakamoto. Interestingly, this coefficient is a constant value of 1 in Reward-All, outperforming all state-of-the-art mining reward proposals in stability. Lastly, we summarize this chapter in Section 6.3.

6.1 Implementation

To implement our prototype, we wrote 6581 lines of Python code across 71 files. Our implementation's data structures share many similarities with the Ethereum cryptocurrency ledger, sharing its design choice of multi-use addresses and Merkle-Patricia Trie based ledger commitments [W+14].

Our main challenge was that Reward-All mining procedures require additional complexity compared to those of Nakamoto, particularly when searching for a nonce to produce a block, when monitoring whether hash attempts are successful, and when validating newly created coins. These concerns stem from the fact that in Reward-All we introduce additional metadata in the mining search space, and a minting target that parallels the block mining target. As we presented in Chapter 4, both of these components are crucial to creating slabs, the crux of our smelting and minting processes.

We prioritize minimizing the overhead slab creation places on the mining process in our implementation, such that miners using our implementation do not suffer a disadvantage in their hashing throughput when producing and storing slabs compared to miners who would only mine blocks and forgo slab production. This efficiency ensures that miners cannot inflate their block production rate at the expense of their slab production. Furthermore, we ensure that slab validation during minting is not computationally demanding.

6.1.1 Multi-threaded Architecture

In our design, we identify three main threads as time-sensitive, and mutually blocking, leading us to prevent their execution from being blocked by each other through multi-threading. As we illustrate in Figure 6.1, these three threads enable the user interface, primary Reward-All functionalities, and peer-to-peer networking communication. Our design is mainly structured around two input/output handling threads for the console interface and networking, in addition to one critical application logic thread for our primary logic. This concurrent division of labor allows us to reduce the impact of auxiliary operations on mining performance,

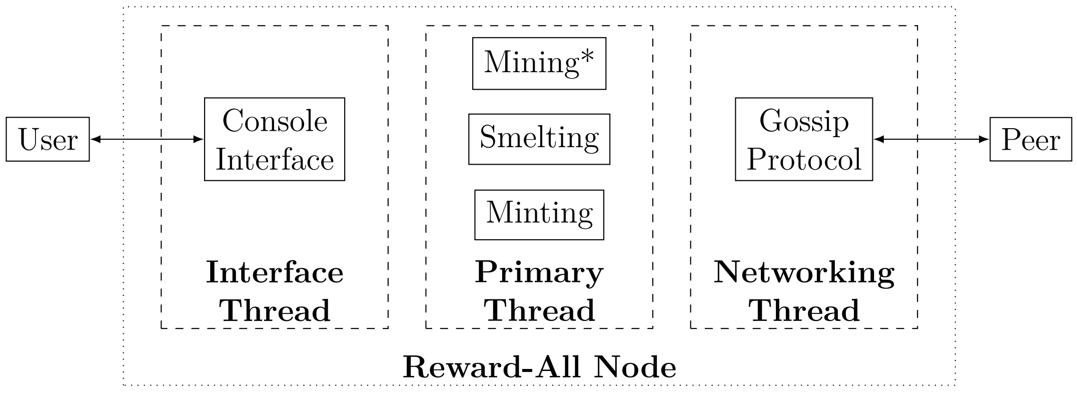


Figure 6.1: Reward-All implementation architecture diagram. The User and Peer are processes external to the system, which communicate with the node via the interface and networking threads respectively. While not pictured, bidirectional inter-process communication channels exist between all three threads. * The Mining process launches child threads as needed.

which is our fundamental concern.

Primary Thread. Our architecture's efficiency relies primarily on the user input of the minting target (T_m), which determines how frequently mining is interrupted on the primary thread to store slabs. If the local miner outputs slabs in between each block arrival so frequently that the average hashing throughput of its mining procedure is disrupted, then we consider T_m as relatively low compared to the average target the local miner can effectively meet once per block arrival on average.

This frequent interruption can be disruptive because the mining procedure launches independent threads for each batch of mining attempts. Each thread terminates after producing a slab or block, and returns its output to the Primary Thread for storage before another thread is started to continue mining with new metadata. Smelting, and minting, tasks, however, operate directly on the Primary Thread without significantly disrupting performance.

Nonetheless, our implementation serves as a prototype, and can benefit from additional features and optimizations for use in a live network. Notably, a means to perform distributed mining across multiple machines is necessary. For a single user operating multiple independent mining hardware units, using different identities for each unit would create a minting cost overhead, as each unit would require its own minting transaction. When using the same identity across machines, however, the mining metadata has to be coordinated between them, and the resulting slabs smelted together for minting. Consequently, our prototype is only most suitable for a user to operate an independent peer that only utilizes one machine in the network.

Networking Thread. We created our implementation to permit a local miner to act as a node in a distributed peer-to-peer network of Reward-All miners. The Networking Thread essentially communicates with a set of peers, which are specified by the user, in order both to decide on the canonical chain that the local miner should base its mining efforts on, and to guide its peers towards mining on that same chain. Consequently, the main role of this thread is to gossip with peers on ledger data, propagating and retrieving transactions and blocks.

To remain efficient, a local miner only solicits full block data from its peers when such data is relevant to extending or changing its local view of the canonical chain. Similarly, a miner only provides full block data when solicited by its peers, and instead advertises a summary of newly acquired blocks when available. For example, when the local miner finds or receives a new block, it only advertises the block's header to its peers, and only provides the block's full contents to peers which solicit it. Similarly, only transaction hashes are advertised, and full details are provided to a peer upon solicitation.

To facilitate this prototype's deployment in practice, however, some improvements can be made to its networking. For example, a peer reputation system, such as that of Bitcoin nodes, which allows a local miner to maintain a rating of its peers based on their behavior is necessary. Such ratings can be then used to drop connections with misbehaving peers, which e.g. provide invalid block data or do not answer solicitations for advertised data. Furthermore, peer discovery mechanisms are also required in order to enable the overlay network to automatically form new connections rather than depend on manual user specification of node addresses.

Interface Thread. This thread mainly reads the local miner's state, and renders it as a textual user-interface in the console. It furthermore reads user-input and translates it into internal instructions which our three main threads can process. While this thread's role is fairly straightforward, the challenges we faced in its design are three-fold.

First, it should not slow down other procedures in order to read the local miner's state. A naive implementation that performs regular polling in order to provide a live view to the user would be too disruptive. Instead we only push state updates when changes occur.

Second, it should be responsive to user commands without disrupting the execution of other procedures. Rather than naively immediately executing operations based on user input, which can leave the ledger in an incoherent state, we pass instructions to their destination threads using queues, and delegate when to best act on instructions to the destination threads.

Third, rendering the interface must not be computationally expensive. Not only is rendering high-resolution images and rich graphical user-interfaces unnecessary to our implementation's utility, but so is repeatedly refreshing the interface, which uses precious CPU time. Consequently, we only render the latest state in the console as a Text User-Interface (TUI) every 500 milliseconds. This not only reduces the workload on the local miner, but also enables the TUI to be efficiently used by clients connected remotely to the local miner's console.

6.1.2 Data Flows

In this section we present how we store, retrieve, and process the data that enables Reward-All coin issuance throughout our implementation. Of particular interest are two of our mechanisms. The first is for minimizing the interruption of mining to store the generated slab data, even when operating under relatively low minting targets that would result in frequent mining interruptions in a naive procedure. The second is for preventing reward duplicity when minting coins, such that reference and sequence numbers that were previously used for coin issuance by the same miner cannot be reused.

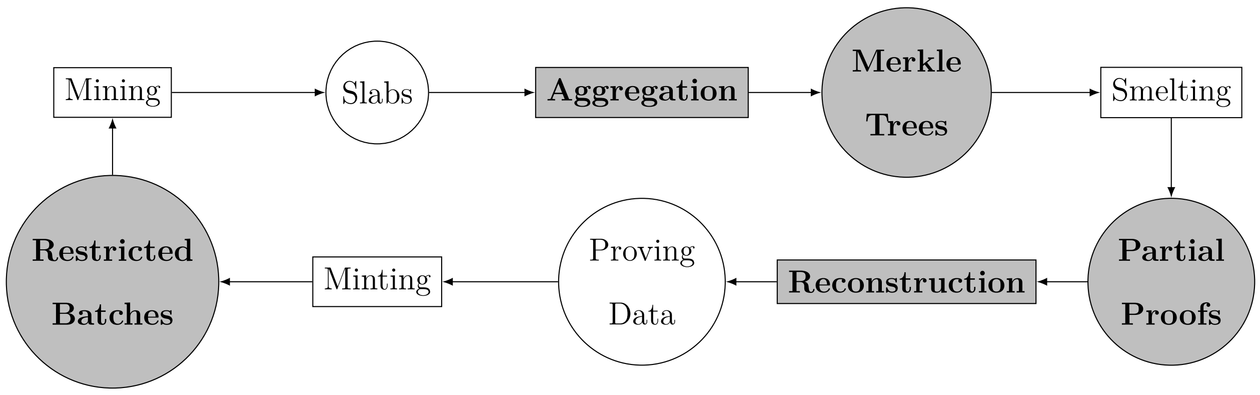


Figure 6.2: Reward-All implementation data flow diagram. In addition to the processes specified in the original system design, we introduce five additional components in this diagram: The **aggregation** process, their resulting **merkle trees** data outputs, **partial proofs** that result from smelting, the **reconstruction** process, and the **restricted batches** resulting from minting.

Aggregation and Reconstruction. In our implementation we introduce two additional intermediate processes between mining, smelting, and minting. The first is Aggregation, whereby the resulting slabs from mining are encoded as Merkle tree commitments before being accessible for smelting. The second is Reconstruction, whereby the partial proofs that result from smelting encoded Merkle trees are amended to include the necessary proving data by our Reward-All protocol. One additional parameter, the **compression commitment height** N, determines the efficiency and performance of these two additional steps both in reducing the number of times a local miner commits slab data to its storage and in maximizing mining throughput.

Aggregation. During the aggregation process, a batch of slabs of some predetermined size 2^N slabs, where N is the *compression commitment height*, is first found in memory by the miner. Then, only the Merkle-tree root commitment to the 2^N slabs is passed to the storage, along with the reference and sequence number range metadata of the slab batch.

Reconstruction. During reconstruction, we recompute each batch selected in a smelted proof in order to amend the proof with the Merkle-tree inclusion data necessary to open the commitment data behind the targeted set of slabs. For each batch, where N is the *compression commitment height*, 2^N slabs are deterministically recomputed using the same reference and sequence number ranges, such that the full tree behind the stored slab batch Merkle-tree roots is recreated.

As N increases, the savings associated with storage space and throughput increase. However, the reconstruction process also becomes more computationally intensive. For significantly large minting targets, such as those which can be met on average only once per block by the local miner, increasing N may lead the reconstruction costs to outweigh the storage savings.

While these two processes reduce the frequency of mining interruptions, we furthermore implement a measure to reduce the duration, or impact, of each mining interruption. Namely, we pre-generate a series of reference and sequence block number pairs to be used for the next series of mining attempts. This amortizes the storage lookup costs necessary for reference and sequence block number generation over a series of mining interruptions. The length of the series is configurable in the implementation.

Restricted Batches. As mentioned in Section 4.4, slab data that was previously used to smelt a proof and mint a set of coins may not be reused to mint a different set of coins in order to prevent reward duplicity. We differentiate between different batches of slabs using only two pieces of information: the miner address, and the sequence block number.

While two or more different batches of slabs may share the above information, but have different reference block numbers or nonces, and consequently require independent amounts of work to create, using only the above fields, we efficiently prevent reward duplication through one rule:

> For each miner, every new batch of slabs must utilize a starting sequence block number larger than all previously used ending sequence block numbers.

Minting transactions that conform to this rule are referred to as **restricted batches**, as illustrated in Figure 6.2. Notably, under this rule, miners should not use starting sequence block numbers that would lead to a loss of the value of previously unused slabs with lower sequence block numbers.

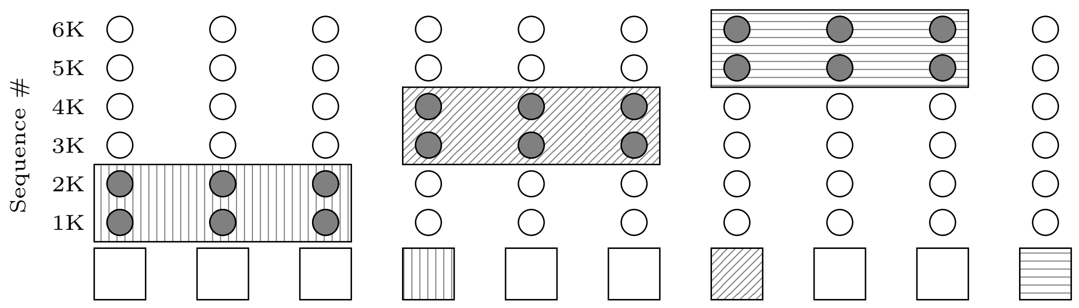


Figure 6.3: Example restricted minting scenario. Empty circles denote non-existent weak headers. Full circles denote valid weak headers. Squares denote reference blocks. The row of a circle determines the sequence number range its headers fall in. The column, or block, determines the reference block number used by all represented headers. Three shaded regions denote three individual batches of weak headers published in minting transactions in their corresponding block shaded in the same style.

To illustrate this rule, we provide an example scenario in Figure 6.3. One miner in this case publishes three minting transactions in the three shaded blocks. The shading of the block corresponds to which batch of slabs it rewards. In each minting transaction, the miner publishes proofs for $6,000$ slabs, and receives no penalty as the transactions are confirmed in the blocks immediately following the last used reference block numbers. If the minting transaction for the second batch of slabs were confirmed before the first, the miner would not be able to publish a minting transaction for the first batch, not considering the penalty.

6.2 Results

To gain insight into Reward-All, we take several different evaluation approaches. Namely, we measure reward issuance costs, simulate different parameterization scenarios to understand how transaction fees can affect minting, and mathematically derive a model of Reward-All coin creation stability.

6.2.1 Proof Sizes

In our sampling-based approach, proof sizes are expected to grow in proportion to the fraction of validity required from the proof. To assess proof sizes in concrete terms, we use the

SHA3 hashing function with an output of length 32-bytes (256 bits) with the security parameter $\lambda = 128$ in the proving system. Consequently, each sampled slab is comprised of 160 bytes for the header, in addition to a the Merkle-tree inclusion proof data. As only full binary Merkle-trees are considered within our implementation, which are augmented with dummy-nodes if they are not full, the number of bytes required for the Merkle-tree inclusion proof of each sample is equal to $\lceil \log_2 |headers| \rceil$, where $|headers|$ is the total number of headers the proof is referring to.

In Figure 6.4, we plot the proof-size evaluation results on a semi-log scale using Equation 6.1. With 2^{50} slabs, proving that 50% of slabs are valid requires 220 kilobytes of data, while proving that 95% of a collection of 2^{65} slabs requires 3.6 megabytes of data. Effectively, this approach renders fractions of validity close to 1 impractical, which means that no miner can be fully compensated for its entire collection of slabs using a sampling based proof.

$$B = 32 \times \frac{\log 2^{-\lambda}}{\log \theta} \times (\lceil \log_2 |headers| \rceil + 5) \tag{6.1}$$

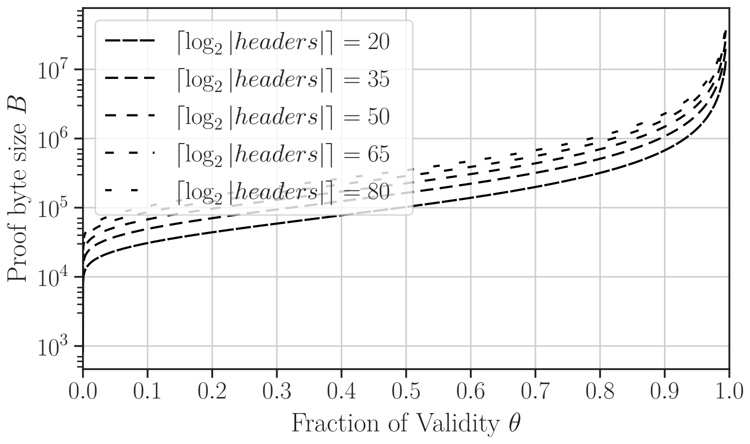


Figure 6.4: Semi-log plot of proof byte sizes versus claimed fractions of validity for different header collection sizes, derived from Equation 6.1 for $\lambda = 128$.

On the other hand, it can be more cost-effective to **fully reveal** an entire collection of slabs rather than samples of it, depending on the number of slabs $|headers|$ in the collection and the target fraction of validity θ. As long as Inequality 6.2 holds, the data required to reveal the entire collection would be no more than that required for the sampling based proof, but would permit full validation of the entire slab collection.

$$160 \times |headers| \leq B \tag{6.2}$$

For example, revealing as little as 5 slabs would only cost 800 bytes. If a miner's share of the mining power is significant (e.g. 0.1%), and its minting target is equal to its power, it will be, under certain transaction fee conditions, more economical to publish full slabs.

6.2.2 Reward Times

To evaluate mining reward times, we evaluate the expected number of blocks an α-strong independent miner in Reward-All would have to wait before it can receive compensation for its work. We refer to this waiting time as the **expected reward delay**. Notably, at all times, Reward-All cannot perform worse than Nakamoto because a miner can use the space in the block it generates for its own minting purposes.

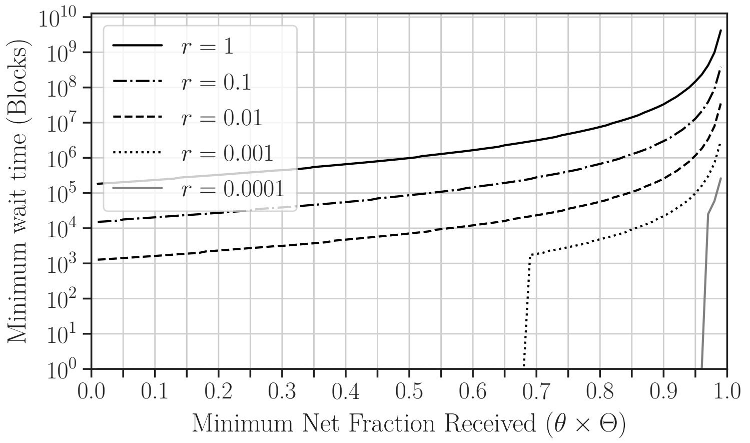


Figure 6.5: Semi-log plot of minimum number of headers required in Reward-All for different net payout fractions. $r = \frac{c_b}{\alpha \times D_b}$ denotes the ratio between c_b, the cost per byte, and $\alpha \times D_b = D_m$, the maximum minting difficulty of a miner.

Consequently, our focus lies on evaluating the fraction of validity and transaction-fee thresholds under which Reward-All has a smaller expected reward delay than Nakamoto for an α-strong miner. In Figure 6.5, we plot the minimum number of slabs a miner would have to accrue before it is able to publish a proof which rewards it with at least a fraction $\theta \times \Theta$ of the coins it is owed, which we refer to as the **net reward fraction**. This excludes penalties or external expenses such as mining costs.

As a reminder, θ denotes the fraction of validity which the proof claims, i.e. the portion of slabs which are proven valid with overwhelming probability. Θ denotes the fraction of proven coins that will be rewarded to the miner after transaction fees are deducted.

We examine the relationship between the net reward fraction and the minimum number of slabs under different values of r, which is the ratio between the cost per byte in transaction fees c_b and the miner's minting difficulty per slab D_m that determines the reward per slab. Since the minting difficulty D_m should not exceed $\alpha \times D_b$ to avoid penalties, as discussed in Chapter 4, we use r is to infer a direct comparison between the expected reward delay in Reward-All and in Nakamoto for different α-strong miners under the same block mining difficulty D_b and different transaction fees.

To compare the expected reward delay between Nakamoto and Reward-All, we first choose the parameters α and c_b. As an example, consider $\alpha = 0.01632\%$ of the mining power, and $c_b = 102$, which is approximately the current cost per byte in Bitcoin for transaction inclusion within the next block. The expected reward delay in Bitcoin for this miner is approximately $\frac{1}{\alpha} \approx 6127$ blocks, which amounts to nearly 42.5 days. Then, we derive the ratio r between the transaction fee per byte and the expected rewards per block for an α-strong miner. Since the reward per block in Bitcoin is currently[1] $R = 6.25 \times 10^8$ coins (Satoshis), therefore $r = \frac{c_b}{\alpha \times R} \approx 0.01$.

Using our calculated value of r for the α-strong miner, we can infer the expected reward delay in Reward-All using Figure 6.5. When $r = 0.01$, the miner can claim 10% of its owed coins after accumulating 1632 slabs, 20% with 2322 slabs, 30% with 3151, or 40% with 4736, which would amount to D_m multiplied by 163, 464, 945, and 1894 coins respectively.

More optimistically, with $r = 0.001$, miners can expect to claim up to 68% of their reward every block with $\Theta = 0.68$, and up to 96% when $r = 0.0001$ by setting $\Theta = 0.96$. In such

[1]As of April 5th 2022

a setting, it becomes economic for miners to publish full slabs that meet their minting target as they are found, opting for full validation ($\theta = 1$) of their reward claims, and paying the fraction $1 - \Theta$ of the coins minted as transaction fees.

In hindsight, an eager independent miner can pay a portion of its mining rewards to reduce its expected reward delay in Reward-All, while such an option is not present in Nakamoto without joining a mining pool. The cost of this perk largely depends on the overhead of the proving system, which is non-negligible in our specification of Reward-All, and may lead the miner to choose to collect transaction fees instead of utilizing its mined block to mint its own coins.

6.2.3 Reward Variance

We derive the coefficient of variation of the distribution of mining rewards per block in Reward-All as follows. Let $\mathbf{RA}(\alpha)$ denote the random variable which represents the **fraction of unminted mining rewards** attained by an individual miner in Reward-All with a share α of the total network's hashing power. As the block discovery process is identical to that of Nakamoto, the time interval between block arrivals follows an **exponential distribution**, such that blocks continuously arrive independently of each other at an average constant rate λ. Such a distribution is denoted by $\mathrm{Exp}(\lambda)$, and is used to define $\mathbf{RA}(\alpha)$ accordingly in Equation 6.3:

$$\mathbf{RA}(\alpha) = \mathrm{Exp}(\lambda) \times \alpha \tag{6.3}$$

It follows that since the expected value $\mathrm{E}[\mathrm{Exp}(\lambda)] = \frac{1}{\lambda}$, therefore the expected value of $\mathbf{RA}(\alpha)$ is as follows in Equation 6.4:

$$\mathrm{E}[\mathbf{RA}(\alpha)] = \mathrm{E}[\mathrm{Exp}(\lambda)] \times \alpha = \frac{\alpha}{\lambda} \tag{6.4}$$

Furthermore, as the variance $\mathrm{Var}[\mathrm{Exp}(\lambda)] = \frac{1}{\lambda^2}$, the variance of $\mathbf{RA}(\alpha)$ is derived in Equation 6.5 as follows:

$$\mathrm{Var}[\mathbf{RA}(\alpha)] = \mathrm{Var}[\mathrm{Exp}(\lambda)] \times \alpha^2 = \frac{\alpha^2}{\lambda^2} \tag{6.5}$$

Lastly, Equation 6.6 presents the formula for the coefficient of variation of $\mathbf{RA}(\alpha)$, which is defined as the ratio of the standard deviation of the variable to its mean:

$$\mathrm{CV}[\mathbf{RA}(\alpha)] = \frac{\sqrt{\mathrm{Var}[\mathbf{RA}(\alpha)]}}{\mathrm{E}[\mathbf{RA}(\alpha)]} = 1 \tag{6.6}$$

The coefficient of variation of per block mining rewards in Nakamoto is [SRHS19]:

$$\mathrm{CV}[\mathbf{NC}(\alpha)] = \frac{\sqrt{\alpha \times (1-\alpha)}}{\alpha} \tag{6.7}$$

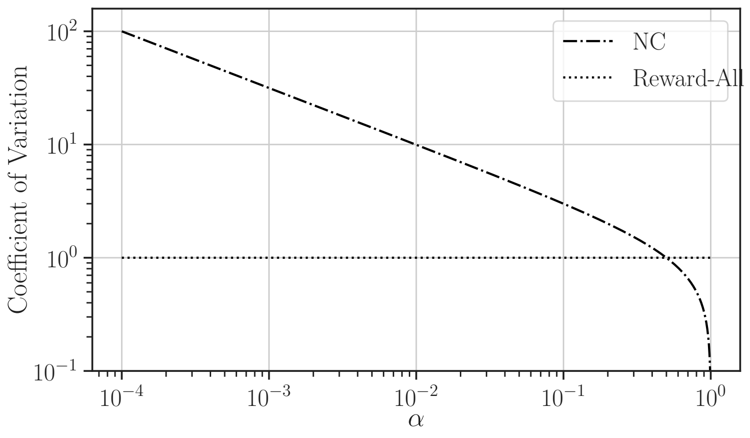


Figure 6.6: Log-scale plot of coefficients of variation in Nakamoto and Reward-All versus mining power α.

In Figure 6.6, we present a plot which demonstrates how the constant coefficient of variation in mining rewards of Reward-All compares to the variable coefficient of Nakamoto at different individual mining powers α. Consequently, it can be seen that Reward-All outperforms Nakamoto in reward variability for $\alpha < 0.5$, and a tie is found at $\alpha = 0.5$. However, Nakamoto miners with $\alpha > 0.5$ of the mining power exhibit less reward variance than Reward-All miners with the same hashing power.

6.3 Summary

In this chapter we presented our implementation of a Reward-All full-node, and our corresponding evaluation of Reward-All. For brevity, we delegated low-level technical discussions around implementation code to Appendix A. We introduced the challenges to implementing Reward-All, and our corresponding solutions which enable miners to minimize the overheads associated with slab production and storage. Furthermore, our evaluation focused on quantifying proof sizes and the variables that affect them, the costs which miners would have to incur before being able to mint their coins, and the long-term stability of the flow of mining rewards for miners of different relative mining powers.

First, in Section 6.1 we introduced our implementation's multi-threaded architecture, highlighting our use of threading to protect mining performance from being hindered by other tasks in the application. We overviewed our three main threads, which executed our procedures for our user-interface, core Reward-All logic, and networking. Moreover, we presented how data flows across different components of our implementation, explaining the main challenges associated with supporting the Reward-All coin issuance model. We first discussed how we handle relatively low minting targets to minimize storage space and mining interruptions. Then we described our measures that efficiently prevent reward duplicity.

Later, in Section 6.2.1, we examined the byte-sizes of proofs in our Reward-All implementation, and highlighted a clear boundary equation using which Reward-All miners can decide whether it is more economical to use our probabilistic proving system, or publish their full slabs onto the chain. Furthermore, we examined the trade-off Reward-All miners can make between spending rewards early, and spending discounted rewards, under different transaction-fee conditions, showing that Reward-All offers a reward time advantage not present in Nakamoto. Lastly, we systematically derived the coefficient of variation of per-block Reward-All mining rewards for miners with different relative mining powers. We showed that Reward-All establishes a constantly stable stream of reward for miners of all sizes over the long term, outperforming the Nakamoto reward stability for miners with less than half of the mining power, yet falling behind the stability offered by Nakamoto for miners with more than half of the total network mining power.

In the next chapter we discuss the challenges and limitations of our Reward-All protocol, and propose solutions to overcoming their drawbacks.

Chapter 7

Discussion

In this chapter we discuss the most notable operational aspects of Reward-All, as well as outline approaches and methods for tackling the challenges they pose. Our main areas of concern are the security and feasibility of operating a Reward-All blockchain in practice, focusing on the qualities of mining, smelting, and minting.

We first examine in Section 7.1 how *negligent miners*, which dedicate their mining resources to creating slabs, rather than contributing to block creation difficulty, can be dissuaded. Then, in Section 7.2 we consider the incentives for miners to form pools in Reward-All, and how pool members can mine blocks independently, but combine only their slabs to cut down costs. Subsequently, we explore how to maximize the potential gain from the smelting process in Section 7.3 while minimizing minting penalties and costs. Lastly, we look at the unique coin supply properties Reward-All issuance establishes in Section 7.4, where we investigate total coin supply transparency, transaction inclusion incentives, and real-world coin valuation.

7.1 Negligent Mining Deterrence

One remarkable difference between Nakamoto and Reward-All is that a miner which utilizes its power only towards creating slabs for minting, without aiming to append a valid block to the main blockchain accepted by the network, may still receive mining rewards.

While this problem is similar to mining empty blocks in Nakamoto, such that the main disincentive for this behavior is the miner missing out on collecting transaction fees, the main difference in Reward-All is that miners who do not contribute towards increasing the block mining difficulty of the main blockchain can still be rewarded.

We refer to such miners as **negligent**, as they ignore strengthening the security of the chain that their rewards are tied to against block replacement. Despite the penalty introduced in Section 4.3 against miners which reference fairly outdated blocks, and the loss in potential transaction fee earnings, negligent mining is not completely eliminated.

Solution. Our proposed solution to this problem would be to introduce a specialized keep-alive mechanism, which is of independent interest. We summarize this mechanism as follows:

1. Miners append additional metadata to their slabs. This additional data is a commitment to the following:
   (a) The miner's network identity, which is a freshly generated public identity.
   (b) The Merkle-tree root of the miner's set of peers. This set must be of a globally fixed sufficiently large size (e.g. 128).
   (c) The miner's keep-alive target. This value is similar to the minting and mining targets, but is used only as the threshold for which keep-alive messages are sent to peers. The miner must be able to meet this target periodically.
2. Whenever the miner meets the keep-alive target while mining, it sends a keep-alive message to its peers, which contains the Proof-of-Work, and the metadata above. Each peer validates its membership in the set of peers in the metadata, and that the PoW is mined on the latest block in the canonical chain.
3. Miners attempt to maximize the cumulative difficulty contribution of their peers, such that when a connection with a larger keep-alive target can be established in exchange for dropping that of a lower keep-alive target, the opportunity is taken.

In essence, we believe this should force negligent miners to contribute to the canonical chain's mining difficulty the minimum amount of work necessary to stay connected to the overlay of compliant miners and synchronized with the latest canonical block. With this mechanism in effect, mining on older blocks can no longer be attributed to neglecting to synchronize the latest block state, and may only be useful for intentionally-adversarial chain forking.

7.2 Pooled Mining Incentives

For individual Nakamoto miners with small mining powers, joining a mining pool offers a solution to the problem of having to wait too long for rewards. Even in Reward-All, miners are incentivized to join mining pools, because the minting costs are shared by the entire pool, which would drastically improve the cost-efficiency of minting for small Reward-All miners.

In Nakamoto, large mining pools are attractive because they offer steadier rewards than their smaller counterparts. However, reward variance in Reward-All is constant (q.v. Section 6.2.3), allowing a mining pool with as little as 0.1% of the network hashing power to enjoy an r-value[1] $\approx 1.632 \times 10^{-4}$ under today's Bitcoin transaction fees, as explained in Section 6.2.2.

Therefore, while redeeming mining rewards in Reward-All does not discourage mining pools, as in [MKKS15], it does reduce the incentive to form significantly large ones. Of course, our discussion ignores the potentially added income from sharing transaction fees in the mining pool, which can form an additional incentive for pools to form.

Solution. Alternatively, we propose the formation of *minting* pools, rather than mining pools, whereby miners conjoin together their batches of slabs during smelting, and submit a unified set of batches for minting. When doing so, the posted proof for minting validates all conjoined batches simultaneously, rather than one by one.

[1]Recall that r is the ratio between the cost per byte in transaction fees and the reward per slab.

In order to make grouped minting practical, our proving system must be altered in order to support slab batches from different miners with different minting targets. However, each individual batch conjoined may still only contain slabs all having the same minting target for simplicity. To accomplish this, we change our sampling strategy as follows. Each conjoined batch is given a weight equal to the batch's minting target multiplied by the number of slabs in the batch. A batch's weight, divided by the sum of all batch weights, decides the probability that the next sample will be drawn from that batch. After selecting a batch, a random slab is uniformly drawn as previously done in our scheme. Furthermore, the verifier must also ensure that no previously smelted slabs are being reused for every conjoined batch. This can be accomplished for each batch as previously described in Section 4.4.

Under this scheme, miners are free to independently collect and validate transactions, avoiding centralization. They mainly coordinate with each other when smelting the master proof for their conjoined batches. Even at that stage, miners are free to change who they partner with for minting, or to mint independently.

7.3 Full Smelting Opportunity

Recalling our earlier assumptions from Section 6.2.3, we modeled the canonical chain as having an average inter-block arrival rate of λ, with each arrival being independent of all others, such that the distribution of inter-block arrivals follows an exponential distribution $\text{Exp}(\lambda)$. Similarly, slabs are found by a miner at a constant rate on average, whereby individual inter-arrival times also follow an exponential distribution. Denoting the slab arrival rate by $\hat{\lambda}$, we denote the exponential distribution of their inter-arrival times as $\text{Exp}(\hat{\lambda})$.

Because of how these distributions behave, the ratio between $\hat{\lambda}$ and λ is equal only to how many slabs a miner expects to find before a new block is found **on average**. Consequently, miners will not always be able to smelt a proof which captures all of the slabs they have found so far, without incurring the penalty described in Section 4.4.2.

More specifically, because miners have to specify a reference block hash that is currently part of the canonical chain, when the miner generates, for a given sequence number, a slab that utilizes the canonical chain tip as a reference block, it must move on to another sequence number. This means that, for each sequence number, a miner may not remain on par with the chain's height, and will slowly start to fall behind.

Notably, the unused portion of slabs does not go to waste, but can instead be used for a separate smelting operation. This leads to a "lock-in" effect for Reward-All miners, which deters them from discontinuing mining, as miners which abandon mining have one of two choices: (i) perform smelting and minting using the slabs accrued so far, incurring a penalty for sequence numbers which fall behind in terms of the latest reference block used, or (ii) discard slabs accrued so far, losing out on their reward value. Such a lock-in effect is not present in Nakamoto, where miners may stop at any time without such an explicit loss in rewards owed.

Solution. We can mitigate this issue using the following reference block hash and sequence number scheduling strategy: given a starting reference block B and starting sequence number N, select the minimum possible sequence block number followed by the oldest possible reference block as the next mined slab parameters.

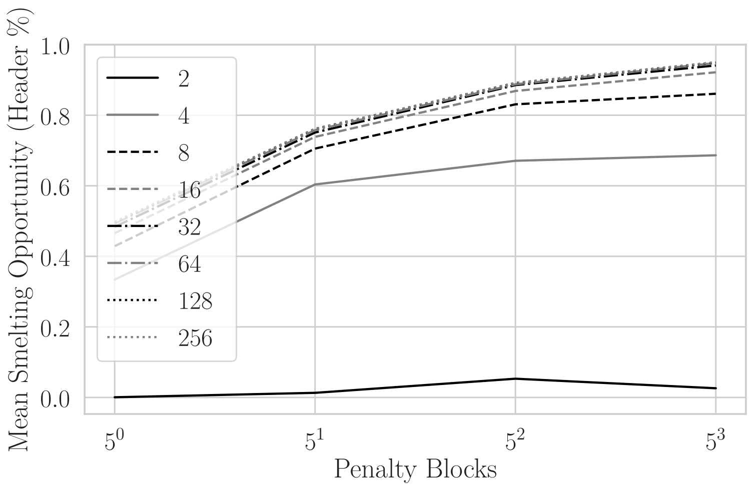


Figure 7.1: Semi-log plot of the number of penalty blocks a miner is willing to accept versus the expected fraction of accumulated headers the miner can utilize for smelting under different header to block ratios. These values are measured for our reference and sequence number scheduling strategy.

In Figure 7.1 we plot the effectiveness of utilizing our strategy, where we measure the expected fraction of accumulated slabs a miner will be able to utilize while smelting under different scenarios.

Our first observation is that as the ratio $\frac{\hat{\lambda}}{\lambda}$ between the miner's slab mining rate and the block mining rate increases, the miner is able to utilize more of its accumulated slabs at a time on average, reaching almost half at a time without penalty when $\frac{\hat{\lambda}}{\lambda} = 256$.

Second, we observe that as the difference in block height between the canonical chain tip and the miner's ending reference block increases, and consequently the minting penalty increases, the miner is able to utilize a larger portion of its accumulated slabs when smelting. This is because the miner is able to utilize a larger range of sequence numbers.

7.4 Coin Supply Dynamics

Transparency. Interestingly, our Reward-All mining mechanism introduces a layer of uncertainty about the true total coin supply. Nothing restricts miners who continue to participate in mining from proving their successful weak mining attempts later rather than sooner. This means that miners may choose to keep their coins secret without penalty, as long as they continue to mine. Nonetheless, assuming that the effect of negligent mining on block difficulty is negligible, the mining difficulty can be used to estimate the total number of hashes computed by miners as we have done for Bitcoin in Chapter 3, establishing an upper-bound on the total number of Reward-All coins all miners may mint.

Fees vs rewards. Because we employ a statistical-sampling based proving system, an overhead is introduced to minting coins. When transaction fees are high, a miner may find it more economic to refrain from proving that it should mint a set of coins as mining rewards, or only mint a certain fraction. Even if the miner in question gets to propose a block, if it chooses to occupy that block's space with its own minting transactions, it would do so at the cost of forgoing transaction fees that would it could have earned from including other parties' transactions. This means that it may be more profitable to collect transaction fees instead of minting mining rewards when transaction fees are expensive, or it may not be cost-effective or economic to mint rewards that are relatively small compared to the current transaction fees. Introducing a more cost-effective proving system to Reward-All, possibly based on more complex cryptographic tools such as succinct zero-knowledge proofs, than our approach would reduce this overhead.

Coin valuation. Communities and advocates of many prominent cryptocurrencies encourage users to hold on to their coins and not sell them based on the belief that the price of their coins will increase later on in time. This belief is mainly based on the fact that the supplies of these coins exhibit a form of scarcity in their issuance that is fueled by a lack of correspondence between the total amount of computational power invested in mining and the number of coins issued as a reward per block. In Reward-All, such a correspondence exists, and therefore the notion that holding onto coins will yield a profit would not be as well arguable based on coin scarcity. On the other hand, it does not make sense for a miner to exchange a unit of currency it has earned for something of less value than its cost of mining and endure a loss. This creates a natural price floor for Reward-All coins. However, future advances in technology will result in cost-efficiency improvements and/or computational power increases. This implies that over time, the cost of producing a single coin in Reward-All would decrease as technology improves, leading to a form of inflation influenced by advances in hardware.

7.5 Summary

In this section we discussed several important challenges in Reward-All coin issuance, and outlined our approach to overcoming them in practice.

First, in Section 7.1 we introduced the problem of negligent mining, whereby Reward-All miners receive rewards despite not contributing to block mining difficulty, a phenomenon not present in Nakamoto. In addition to the penalties imposed on said miners during minting, we outlined a solution that enables compliant miners to filter out negligent peers, disconnecting them from the network unless they contribute to block difficulty.

Second, in Section 7.2 we highlighted how high minting transaction costs can impact Reward-All miner behavior, encouraging the participation in centralized mining pools. As a more decentralized alternative, we proposed minting pools, whereby miners independently mine blocks, yet conjoin their batches of slabs during minting in order to share the validation costs.

Third, in Section 7.3 we presented how slab accumulation speed may fall behind block production, forcing miners to use outdated reference block hashes and endure harsh penalties which lead to a lock-in effect that acts as an incentive against abandoning mining. As a countermeasure, we presented our reference block and sequence block scheduling strategy, and quantified its effectiveness in maximizing the number of accumulated slabs that can be smelted at a time under different penalties and relative minting targets.

Lastly, in Section 7.4, we discussed various aspects about Reward-All's coinage, whereby full transparency into the total amount of coins may not be possible, transaction fees may cause miners to favor earning transaction fees rather than minting, and potential profiteering from holding onto a large set of coins would be practical.

In the next chapter we conclude this thesis with a summary of each chapter, an outline of future research directions, and our final closing remarks.

Chapter 8

Conclusion

In this thesis we designed, analyzed, and implemented Reward-All Nakamoto Consensus, a novel reward mechanism which provides equitable Proof-of-Work (PoW) mining rewards while using Nakamoto Consensus. In this chapter we conclude our work. First, in Section 8.1, we reexamine our methods, and assess our results, proceeding chapter by chapter. Then, in Section 8.2 we outline our directions for future work. Lastly, we come to an end in Section 8.3 with our final commentary.

8.1 Review

Chapter 1. We highlighted the drawbacks of Nakamoto's winner-takes-all lottery, whereby only a single miner is periodically elected to receive a fixed number of reward coins, and a stale-block can reallocate that reward to a different winner with non-negligible probability. Consequently, we established five main objectives for Reward-All to accomplish in order to overcome these drawbacks without losing the core benefits of Nakamoto:

1. Establishing a fixed mining cost per coin.
2. Minimizing value leakage from mining rewards to existing coin holders.
3. Normalizing reward variance across mining powers.
4. Retaining the mutual peer distrust of Nakamoto.
5. Minimizing the impact of stale blocks on miner rewards.

Chapter 2. As we set out to reinvent a core blockchain functionality, we reviewed central notions of blockchain design, spanning ledger data structures, blockchain consensus protocols, and mining reward mechanisms. Central to our objectives were our examinations of related work in compliance payoff mechanisms, i.e. other alternatives to Nakamoto for issuing mining rewards. Notably, state-of-the-art mechanisms which similarly aim to provide fairness for miners fell short in at least one of thee primary ways compared to Reward-All.

1. They only allowed up to N winners to succeed in a recurring lottery similar to that of Nakamoto, which still drives a competitive process with significant reward instability.
2. They did not account for the limited bandwidth of the underlying blockchain, which would cause miners to permanently lose rewards during congestion.
3. They did not issue rewards in proportion to the mining power used by miners, which would lead to unfair coin production costs.

Chapter 3. We established a computationally-grounded approach to measuring and mandating equity in reward issuance, creating a framework for modeling computational expenditures, mining compensation, and coin value, all in terms of the number of PoW attempts made during mining. Under our framework, we proposed a set of reward issuance constraints which ensure both the proportionality of reward amounts to mining expenditures, and the timeliness of delivering mining rewards to miners. In contrast to Nakamoto, where only a fixed number of coins are issued, our approach permits the relative growth rate of the coin supply to converge over time to the relative growth rate of the network's mining power. Notably, only one related work partially satisfies our criteria, issuing rewards in proportion to the block mining difficulty, but following a halving schedule similar to that of Nakamoto, and having no notion of timeliness as in Nakamoto [Trz].

Chapter 4. We overcame the challenge of rewarding a potentially unrestricted number of miners simultaneously by proposing a system design whereby miners individually track their own computational expenditures in a verifiable manner by producing slabs in addition to blocks while solving PoW challenges. Our slabs not only enable miners to accrue rewards as they mine, and spending these rewards when required, but also provide resilience against reward loss due to stale-blocks. However, many other designs achieve similar properties to Reward-All:

1. Receiving mining rewards is not conditional on being elected leader [DB19, DB20].
2. Stale blocks do not rescind all rewards associated with them from their miners [SZ15].

Nonetheless, Reward-All to date is the only design where a miner's rewards are based on its individual PoW solving throughput. Furthermore, the limited bandwidth of the underlying blockchain does not lead to permanent loss of miner rewards, but only requires that miners sustain their mining throughput until they spend their rewards.

Chapter 5. To assess Reward-All's resilience, we extend the state-of-the-art in block withholding attack analysis methods to create four models of different attacks on Reward-All. Our results demonstrate that Reward-All provides miners with near-ideal resilience in two out of four attacks, equal resilience to Nakamoto in one attack, and worse resilience than in Nakamoto against double-spending attacks. However, as block creation difficulty increases in Reward-All, so does its double-spending resilience, permitting it to potentially exceed Nakamoto for sufficiently high-valued transactions. Despite utilizing the state-of-the art in probabilistic model checking tools, our analysis yields only upper-bounds on the advantages an adversary can attain using block-withholding attacks. This is primarily due to the added complexity of modeling Reward-All, which requires an exponentially growing state-space to accurately depict in a model. Nonetheless, this only means that Reward-All actually performs better than the results of our analysis in practice.

Chapter 6. To demonstrate and assess the practicality of operating Reward-All in practice, we created a prototype implementation of a Reward-All miner, where we introduced an architecture which efficiently enables both PoW expenditure logging by miners, and reward calculation in the blockchain. Our efforts were focused on realizing Reward-All without adding overheads to the mining process, preventing any miners which opt to forgo accruing rewards from inflating their ability to publish blocks in the chain. However, we only produced a prototype capable of showcasing what mining in Reward-All is like, and not a fully-fledged mining client capable of running in a live production environment. Our evaluation focused on the costs of spending reward coins in the blockchain under different transaction-fee conditions, and on the stability of attaining rewards. Following our computationally-grounded approach, we quantify the trade-offs miners can make in Reward-All between reward waiting times and relative transaction fees, which enables miners to access rewards earlier in Reward-All than in Nakamoto, but at the cost of higher transaction fees. Remarkably, miners in Reward-All enjoy a constant coefficient of variation in their per-block rewards, regardless of their relative shares of the total network mining power, leading all Reward-All miners to enjoy equal reward stability, unlike their Nakamoto counterparts which retain significantly less stability as their relative share of the network mining power decreases. However, our methods did not include simulations where multiple clients running our implementation maintained a Reward-All blockchain over time, which could have provided useful insight into the stability of our design over time.

Chapter 7. Furthermore, we critically analyze and propose solutions to limitations in our design. Namely, we outline an approach for miners to force their peers to contribute to the difficulty of creating blocks, an aspect that is not directly addressed by our Reward-All design due to the resilience of rewards against stale-blocks. Furthermore, as an incentive for miners to pool together to reduce costs still exists, we highlight the possibility to form more decentralized minting pools, where miners individually mine for blocks, but only coordinate to share minting costs. Moreover, we address the implications of slab production, which does not always enable a Reward-All miner to create a proof that covers its entire work log, and motivates miners to continue mining. To mitigate this, we propose a method for Reward-All miners to regulate their slab creation in order to be always able to prove half of their slabs on average without penalty. In addition, we discuss notable aspects about Reward-All coins. Namely, they do not guarantee complete knowledge of their total supply as those of

Nakamoto, are more economical for miners to collect as fees rather than as rewards, and do not promote hoarding since they can become cheaper to produce rather than purchase.

8.2 Future Directions

We summarize our primary avenues for future work as follows:

- **Design.** In our design, we directly employed PoW Nakamoto to drive the consensus process. However, Nakamoto has been widely criticized due to its intensive ongoing energy consumption requirements. A revised design could amend Reward-All to utilize storage space, which is a much less energy dependent resource that is favored by many new projects, such as Chia [CP19]. However, many new challenges arise when considering such an alternative consensus design for Reward-All. For example, the notion of slabs in Reward-All needs redefinition to work with storage space, as the idea of grinding a random sampling process is no longer the driving factor behind puzzle solutions.
- **Analysis.** Due to the limitations we faced in modelling mining races under our approach as Markov Decision Processes, we employed several optimizations in the state-space which explicitly over-rewarded the adversary, producing worse results than are actually attainable in practice. Following the methods of Hou et al. [HZJ+19], deep reinforcement learning techniques can be applied to analyze the security of mining in Reward-All against block withholding attacks without our over-rewarding optimizations. Such an analysis should yield more accurate results, and potentially lower values of K when configuring Reward-All for near-ideal resilience against α-strong adversaries.
- **Implementation.** The entire set of tradeoffs we uncovered in our evaluation revolved around the significant costs associated with minting, whereby a large set of proof data is published by miners in order to spend reward coins. Consequently, our use of a probabilistic sampling approach for slab validation is what leads to these expenses, and using a more optimized approach would improve Reward-All's cost-efficiency. An alternative is to investigate the applicability of succinct non-interactive zero-knowledge

arguments to minting in Reward-All. The main challenges will be to retain the flexibility of configurable slab creation targets, the efficiency of the smelting process, and the transparency of minting (i.e. a public-coin argument is necessary).

8.3 Closing Remarks

In retrospect, the design space of incentive mechanisms for mining in blockchains remains widely unexplored, and full of potential prospects. As one moves away from the rigid approach of Nakamoto's restricted reward mechanism, where miners are incentivized to compete for coins rather than to cooperate in securing the blockchain, much more powerful constructs become possible to create. This is partly due to the fact that, in principle, there are no natural grounds for issuing a fixed number of coins per block, or for halving the block reward every $210,000$ blocks. The main repercussion of these two arbitrary design choices is not a gold-like scarcity in coin supply, but rather a starved ecosystem deprived of the necessary incentives to sustain its functionality. Over the long run, we believe that the natural choice for cryptocurrency systems that aim to serve as mediums of exchange or stores of value would be to issue newly minted coins coins in proportion to the amount of demand for their creation, with permissionless mining used as the signal for that demand.

Bibliography

[ABFG14] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *International Conference on Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[ABLZ18] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 541–560. Springer, 2018.

[AZV17] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE symposium on security and privacy (SP)*, pages 375–392. IEEE, 2017.

[B$^+$02] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[BBPS19] Marianna Belotti, Nikola Božić, Guy Pujolle, and Stefano Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[BJO09] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 43–54, 2009.

[BKLZ20] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 928–946. IEEE, 2020.

[BL20] George Bissias and Brian N Levine. Bobtail: Improved blockchain security with low-variance mining. In *ISOC Network and Distributed System Security Symposium*, 2020.

[Blo] Blockchair. Bitcoin explorer – blockchair. `https://blockchair.com/bitcoin`. Accessed: 2022-04-27.

[BMC+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE symposium on security and privacy*, pages 104–121. IEEE, 2015.

[BSAB+19] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

[CCM+20] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[CDG+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

[CP19] Bram Cohen and Krzysztof Pietrzak. The chia network blockchain, 2019.

[CPR19] Xi Chen, Christos Papadimitriou, and Tim Roughgarden. An axiomatic approach to block rewards. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 124–131, 2019.

[DB19] Yuhao Dong and Raouf Boutaba. Elasticoin: Low-volatility cryptocurrency with proofs of sequential work. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 205–209. IEEE, 2019.

[DB20] Yuhao Dong and Raouf Boutaba. Melmint: trustless stable cryptocurrency. *Cryptoeconomic Systems*, 2020.

[DGH$^+$87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

[DKT$^+$20] Amir Dembo, Sreeram Kannan, Ertem Nusret Tas, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Everything is a race and nakamoto always wins. *arXiv preprint arXiv:2005.10484*, 2020.

[DV18] Alex De Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018.

[EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, pages 45–59, 2016.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[Eya15] Ittay Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE, 2015.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

[GKS20] Ulrich Gallersdörfer, Lena Klaaßen, and Christian Stoll. Energy consumption of cryptocurrencies beyond bitcoin. *Joule*, 4(9):1843–1846, 2020.

[Hei14] Ethan Heilman. One weird trick to stop selfish miners: Fresh bitcoins, a solution for the honest miner. In *International Conference on Financial Cryptography and Data Security*, pages 161–162. Springer, 2014.

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015.

[HM19] Shihab S Hazari and Qusay H Mahmoud. Comparative evaluation of consensus mechanisms in cryptocurrencies. *Internet Technology Letters*, 2(3):e100, 2019.

[HZJ+19] Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. Squirrl: Automating attack analysis on blockchain incentive mechanisms with deep reinforcement learning. *arXiv preprint arXiv:1912.01798*, 2019.

[IWSK21] Dragos I Ilie, Sam M Werner, Iain D Stewart, and William J Knottenbelt. Unstable throughput: When the difficulty algorithm breaks. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5. IEEE, 2021.

[JSZ+19] Aljosha Judmayer, Nicholas Stifter, Alexei Zamyatin, Itay Tsabary, Ittay Eyal, Peter Gazi, Sarah Meiklejohn, and Edgar Weippl. Pay to win: cheap, crowd-fundable, cross-chain algorithmic incentive manipulation attacks on pow cryptocurrencies. *Cryptology ePrint Archive*, 2019.

[KAC12] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 906–917, 2012.

[KD21] Rami Khalil and Naranker Dulay. Adaptive layer-two dispute cutoffs in smart-contract blockchains. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 129–136. IEEE, 2021.

[KK21] Dimitris Karakostas and Aggelos Kiayias. Securing proof-of-work ledgers via checkpointing. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5. IEEE, 2021.

[KKKS20] Hyunjun Kim, Kyungho Kim, Hyeokdong Kwon, and Hwajeong Seo. Asic-resistant proof of work based on power analysis of low-end microcontrollers. *Mathematics*, 8(8):1343, 2020.

[KKS+17] Yujin Kwon, Dohyun Kim, Yunmok Son, Eugene Vasserman, and Yongdae Kim. Be selfish and avoid dilemmas: Fork after withholding (faw) attacks on bitcoin. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2017.

[KLK+19] Yujin Kwon, Jian Liu, Minjeong Kim, Dawn Song, and Yongdae Kim. Impossibility of full decentralization in permissionless blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 110–123, 2019.

[KLS16] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 61–78. Springer, 2016.

[KMZ20] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. In *International Conference on Financial Cryptography and Data Security*, pages 505–522. Springer, 2020.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[KSL+21] Lucianna Kiffer, Asad Salman, Dave Levin, Alan Mislove, and Cristina Nita-Rotaru. Under the hood of the ethereum gossip protocol. In *Proceedings of the 2021 International Conference on Financial Cryptography and Data Security (FC'21)*, 2021.

[LABK17] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315. Springer, 2017.

[LSZ15] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[MJP+20] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. Bdos: Blockchain denial-of-service. In *Proceedings of the 2020 ACM SIGSAC conference on Computer and Communications Security*, pages 601–619, 2020.

[MKKS15] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of* *the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 680–691, 2015.

[MO16] Tal Moran and Ilan Orlov. Proofs of space-time and rational proofs of storage. *IACR Cryptol. ePrint Arch.*, 2016:35, 2016.

[N$^+$08] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system.(2008), 2008.

[NH19] Till Neudecker and Hannes Hartenstein. Short paper: An empirical analysis of blockchain forks in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 84–92. Springer, 2019.

[NKMS16] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320. IEEE, 2016.

[OM14] Karl J O'Dwyer and David Malone. Bitcoin mining and its energy footprint, 2014.

[Orl20] José I Orlicki. Sequential proof-of-work for fair staking and distributed randomness beacons. *arXiv preprint arXiv:2008.10189*, 2020.

[PKF$^+$18] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gaži, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *International Conference on Financial Cryptography and Data Security*, pages 480–499. Springer, 2018.

[PL22] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, February 27 - March 2022, 2022*, 2022.

[PPA$^+$15] Sunoo Park, Krzysztof Pietrzak, Joël Alwen, Georg Fuchsbauer, and Peter Gazi. Spacecoin: A cryptocurrency based on proofs of space. *IACR Cryptology ePrint Archive*, 2015:528, 2015.

[PS17] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324, 2017.

[RD17] Ling Ren and Srinivas Devadas. Bandwidth hard functions for asic resistance. In *Theory of Cryptography Conference*, pages 466–492. Springer, 2017.

[Ros11] Meni Rosenfeld. Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980*, 2011.

[SRHS19] Pawel Szalachowski, Daniël Reijsbergen, Ivan Homoliak, and Siwei Sun. Strongchain: Transparent and collaborative proof-of-work consensus. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 819–836, 2019.

[SSZ16] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International conference on the theory and application of cryptology and information security*, pages 90–107. Springer, 2008.

[SZ15] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[Trz] Karol Trzeszczkowski. Ergon - stable peer to peer electronic cash system.

[Trz21] Karol Trzeszczkowski. Proportional block reward as a price stabilization mechanism for peer-to-peer electronic cash system, 2021.

[VG17] Valentin Vallois and Fouad Amine Guenane. Bitcoin transaction: From the creation to validation, a protocol overview. In *2017 1st Cyber Security in Networking Conference (CSNet)*, pages 1–7. IEEE, 2017.

[W$^+$14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[Wal45] Abraham Wald. Sequential tests of statistical hypotheses. *The annals of mathematical statistics*, 16(2):117–186, 1945.

[WHF19] Fredrik Winzer, Benjamin Herd, and Sebastian Faust. Temporary censorship attacks in the presence of rational miners. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 357–366. IEEE, 2019.

[WWK19] Matthew Walck, Ke Wang, and Hyong S Kim. Tendrilstaller: Block delay attack in bitcoin. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 1–9. IEEE, 2019.

[XZL$^+$19] Yang Xiao, Ning Zhang, Jin Li, Wenjing Lou, and Y Thomas Hou. Distributed consensus protocols and algorithms. *Blockchain for Distributed Systems Security*, 25, 2019.

[XZLH20] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465, 2020.

[ZET20] Roi Bar Zur, Ittay Eyal, and Aviv Tamar. Efficient mdp analysis for selfish-mining in blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 113–131, 2020.

[ZL20] Shijie Zhang and Jong-Hyouk Lee. Analysis of the main consensus protocols of blockchain. *ICT Express*, 6(2):93–97, 2020.

[ZP17] Ren Zhang and Bart Preneel. Publish or perish: A backward-compatible defense against selfish mining in bitcoin. In *Cryptographers' Track at the RSA Conference*, pages 277–292. Springer, 2017.

[ZP19] Ren Zhang and Bart Preneel. Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 175–192. IEEE, 2019.

Appendix A

Implementation Addendum

This appendix is designed to give the reader a solid understanding of Reward-All's codebase, providing a technical documentation detailed enough to enable the reader to augment or modify the implementation. To accomplish this, the documentation is presented at a high degree of detail across different sections.

A.1 Overview

At a glance, our Reward-All implementation provides the following key functionalities:

Data Authentication. Our implementation enables a basic blockchain datastructure to encapsulate all executed transactions. Just as in popular blockchains, such as Bitcoin [N$^+$08] and Ethereum [W$^+$14], our chain structure enables a Reward-All full-node to maintain an append-only ledger with strong cryptographic guarantees on data immutability and tamper evidence.

Private Key Management. Our Reward-All full node allows its users to create and manage locally stored private keys, along with their corresponding public account addresses. We utilize a hash-based signature scheme which relies on minimal cryptographic assumptions, and provides post-quantum security guarantees.

Account-based Ledger. We designed an account-based system for our implementation that utilizes multi-use addresses for sending and receiving coin transactions. Similar to Ethereum [W$^+$14], we utilize the hexadecimal representation of the hash of an account's public key to be the public account address.

Reward-All Issuance. We implemented the full suite of procedures required for mining, smelting, and minting. Our full-node client efficiently stores and retrieves the necessary slab data for smelting, as well as enables the creation and broadcast of minting transactions.

Peer-to-Peer Networking. Another key feature of our Reward-All implementation is the support for a full-node to exchange blockchain data with its peers to reach consensus on the canonical chain tip. This involves procedures for listening for connections from, and establishing new connections to, peers running the same full-node implementation.

Textual Interaction. To provide a cross-platform compatible visual interface, we implemented a text-based graphical user interface that can be used in Linux, Windows, and Mac OS machines.

A.2 Data Structures

This section describes each datastructure employed in our Reward-All prototype. In Appendix A, Table A.1 lists the specifications of all structures, providing a compact reference table upon which the remainder of this section will elaborate.

The presentation is split across 6 parts, titled *Trees*, *Authentication*, *Ledger*, *Rewards*, *Networking*, and *UI Instructions*, respectively. The first two parts introduce the commitment and signature scheme primitives used throughout the implementation. Subsequently, the latter two parts introduce the structures used for coin and work accounting. Finally, the last two parts introduce the structures used in peer-to-peer communication, and those used in local intraprocess communication.

Table A.1: Datastructure specification reference table. 21 data classes, followed by 4 field enumerators, are presented in the table below in a compact form.

Structure	
Field	**Format**
Account	
address	bytes32
outgoing transaction count	uint6-++4
balance	uint128
highest sequence number	uint64
Accounted Batch	
creator	bytes32
commitment	bytes32
starting reference block number	uint64
ending reference block number	uint64
starting sequence number	uint64
ending sequence number	uint64
target	uint256
proofs	uint16
Block Header	
miner	bytes32
ancestor block hash	bytes32
parent hash	bytes32
subheader hash	bytes32
nonce	uint128
sequence number	uint64
reference block number	uint64
Block Subheader	
height	uint64
parent block hash	bytes32
parent tree hash	bytes32
state trie root	bytes32
transaction tree root	bytes32
timestamp	uint64
work	uint256
Block Body	
transaction ref list	List[bytes32]
Block	
header	BlockHeader
subheader	BlockSubheader
body	BlockBody
Hash Tree	
root	bytes32
root height	uint8
leaf height	uint8
leaves	Optional[List[bytes32]]
Header Batch	
commitment	bytes32
miner	bytes32
starting reference block number	uint64
ending reference block number	uint64
starting sequence number	uint64
ending sequence number	uint64
target	uint256
Header Batch Proof	
batch commitment	bytes32
challenge	uint16
block header	BlockHeader
membership proof	Bytes32List
External XMSS Key	
public seed	bytes32
public tree root	bytes32
Internal XMSS Key	
public key	ExternalXMSSKey
ots index	uint32
height	uint32
private key	bytes32
private random hashing key	bytes32
full tree nodes	List[List[bytes]]
Peer Message	
subject	PeerMessageSubject
length	uint16

contents	bytes
Peer	
host	str
port	uint16
peer type	PeerType
current conversation	PeerConversation
advertised to block hash	Optional[bytes32]
latest block height	uint64
latest block hash	Optional[bytes32]
latest block work	uint256
common prefix length	uint64
reputation	Optional[uint8]
stream reader	Optional[StreamReader]
stream writer	Optional[StreamWriter]
lookup key	Optional[bytes32]
peer name	Optional[str]
Proposed Block Subheader	
height	uint64
parent block hash	bytes32
parent tree hash	bytes32
state trie root	bytes32
transaction tree root	bytes32
genesis timestamp	uint64
parent work	uint256
Proposed Block	
header	BlockHeader
subheader	ProposedBlockSubheader
body	BlockBody
Trie Node	
jump	uint8
links	List[bytes32]
mask	bytes
One-time Signature	
public key	ExternalXMSSKey
message hash	bytes32
message randomizer	bytes32
wots index	uint32
auth path	List[bytes32]
sig ots	List[bytes32]
Targeted Work Plan	
id	uint64
miner	bytes32
root height	uint8
target	uint256
Transacted Batch Proof	
ancestor block height	uint64
subheader hash	bytes32
parent hash	bytes32
nonce	uint128
membership proof	List[bytes32]

Transacted Batch	
commitment	bytes32
starting reference block number	uint64
ending reference block number	uint64
starting sequence number	uint64
ending sequence number	uint64
target	uint256
proofs	List[TransactedBatchProof]
Transactions	
sender address	bytes32
reference block hash	bytes32
sequence number	uint64
destination address	bytes32
value transferred	uint128
transaction fee	uint128
batches	List[TransactedBatch]
signature	Optional[OneTimeSignature]
Enumerator	
Enumeration	**Value**
Peer Type	
INCOMING	auto()
OUTGOING	auto()
Peer Message Subject	
GREET	uint8(0)
ADVERTISE BLOCK	uint8(1)
SOLICIT BLOCK	uint8(2)
PROVIDE BLOCK	uint8(3)
Peer Message Template	
GREETING	PeerMessage(GREET, 'RA')
GREETING RESPONSE	PeerMessage(GREET, 'NC')
Peer Conversation	
INTRO	auto()
EXCHANGE	auto()
UIInstructions.Mining	
CREATE PROFILE	0x00
MINE PROFILE	0x01
UIInstructions.Batching	
CREATE BATCH	0x02
SMELT BATCH	0x03
UIInstructions.Wallet	
CREATE ACCOUNT	0x04
CREATE TRANSACTION	0x05
UIInstructions.UI	
STATE UPDATE	0x06
EXPLORE	0x07
UIInstructions.Network	
CREATE SERVER	0x08
ENABLE SERVER	0x09
CREATE PEER	0x0A
ENABLE PEER	0x0B

A.2.1 Trees

We use two tree structures to commit to data in our Reward-All implementation, Binary Merkle Trees and Merkle-Patricia Tries. While we use the former for committing to ordered sets of data, we employ the latter in key-value store functionalities. The names of the classes are *Hash Tree* and *Trie Node*.

Hash Tree. Unlike standard Binary Merkle Tree implementations, our *Hash Tree* data structure supports the option to retain only partial trees in memory, whereby only nodes with heights modulo some integer N are stored. When generating inclusion proofs, we efficiently reconstruct the necessary intermediate nodes from the existing data. This added feature is necessary for efficiently managing slab data. Furthermore, we assume that the leaves are 32-byte commitments to the data, leaving the implementation unaware of any specific leaf data-type. Using these trees, we commit to the following data:

- In each block, we commit to the full list of transactions included in the block. This allows efficient proving and verification of the inclusion of a transaction in a block.
- In each block, we commit to the full list of preceding blocks in the chain. This allows efficient block ancestry proofs to be created.
- For each batch of size 2^N of slabs, we commit to the series of slabs, for some integer N defined by the user. This allows us to provide an efficient smelting process.
- In each minting transaction, we commit to the full list of slabs claimed to have been mined by the prover. This enables secure and efficient sample slab verification.

The Hash Tree datastructure embodies a Binary Merkle Tree as follows.

- The *root* field refers to the root of the tree.
- The *root height* field denotes how far away the root node is from the leaves of the tree.

- The *leaf height* field denotes the starting height of the leaves, which defaults to zero.
- The *leaves* field is a list of leaf hashes.

Moreover, the datastructure provides the following class and instance methods.

- The *from range roots* class method allows the instantiation of a Hash Tree using a list of Merkle Mountain Range roots, returning a Hash Tree instance without leaves.
- The *from leaves* class method allows creating a Hash Tree instance using a list of leaves, returning a fully defined Hash Tree instance.
- The *from leaves with height limit* class method returns a list of fully defined Hash Tree instances from a list of leaves using a tree height limit h, such that the n^{th} instance only commits to the n^{th} series of length 2^h from the list of leaves.
- The *relative root height* instance property returns the difference between the *root height* and the *leaf height.*
- The *to node list* instance method returns the full list of internal nodes of the Hash Tree.

Trie Node. The *Trie Node* datastructure is our recursive implementation of the Merkle-Patricia Trie, whereby each structure instance corresponds to a single node that can take the role of a leaf, extension node, or branch node. We assume that the radix of the trie is 16, which allows us to support hexadecimal keys of arbitrary length. Similarly, our Trie Nodes only store commitments to the data they encapsulate. However, only a single Merkle-Patricia Trie is maintained in our implementation, with the Trie's referred to as the State Trie. We use the State Trie to easily keep track of different versions of blockchain data, storing only pointers to updated information as needed, and minimizing data replication across forks. We use that Trie to store commitments to the following data:

- At each account address, we store a commitment to the state of the account. This enables us to later use the account state commitment to retrieve the underlying data.

Furthermore, this allows us to look up an account's state in a given block using the State Trie's root node as a starting point.

- At each slab batch address, we store a commitment to the state of the slab batch. Similarly, we use the commitment to look up minted slab batch data as of a certain block, verifying efficiently how many smelted proofs were provided so far for the slab batch.

When a leaf is encoded:

- The *jump* value is equal to zero.
- The *links* field is a list of only the value stored in the leaf.
- The *mask* field is empty.

When the structure encodes an extension node:

- The *jump* field in this instance encodes the number of nibbles that are to be matched with this node during traversal.
- The *links* field specifies a list of only one item, the next node in the tree that will be reached after traversing the current node.
- The *mask* field lastly encodes the prefix of the key that is to be matched for successful traversal.

When the structure encodes a branch node:

- *jump* is set to zero in this case.
- *links* is used to retain a list of the hashes of all of the children of this node. When *links* contains one more element than the number of ones in *mask*, the last element in *links* denotes the value mapped to the key of the branch node.

- The *mask* field in this case encodes a 16-bit bitmask that indicates which hexadecimal value each link branches to, such that the total number of ones in *mask* is equal to the number of elements in *links*, or smaller by one.

Furthermore, the datastructure provides the following methods.

- The *create trie nodes* internal class method takes as input a list of key, key length, and value tuples, a node storage method, and a radix, and returns the root Trie Node instance that results from creating a Merkle-Patricia Trie using the tuples with the provided radix. During the instance creation, the provided save node method is used to store every intermediate node.

- The *create trie* class method takes as input a list of key, value pairs, a save node method, a key length value, and a radix, and utilizes the internal *create trie nodes* class method to return the resulting Merkle-Patricia Trie while saving its intermediate nodes.

- The *create ref trie* class method takes as input a list of 32-byte references and a save node method, and returns the root Trie Node instance from creating a Merkle-Patricia Trie where the references act as both Key and Value.

- The *create kv trie* class method takes as input a list of keys, a list of values, and a save node method, and returns the root Trie Node instance from creating a Merkle-Patricia Trie with the provided keys and values.

- The *get node with trail* class method takes as input a ancestor node hash, a key, and a get node method and returns the Trie Node stored under the provided key and ancestor node hash.

- The *update trie nodes* internal class method takes as input an ancestor node hash, a key, a value, a get node method, and a save node method, and returns the root Trie Node resulting from updating the value stored under the ancestor node hash's Trie using the provided key to the newly provided value.

- The *update trie with changes* class method forwards its inputs to the *update trie nodes* internal class method, but additionally saves the resulting root Trie Node instance using the provided save node method.

A.2.2 Authentication

To keep the number of cryptographic primitives used in the implementation minimal, we implemented the Winternitz One-Time Signature Plus (WOTS+) and Extended Merkle Signature Scheme (XMSS) algorithms in our Reward-All full-node. Both schemes leverage only hashing in order to enable a public-private key pair to authenticate messages. While these schemes produce significantly larger signatures than modern Elliptic Curve schemes, they are much simpler to implement and describe, while providing security against quantum attacks.

External XMSS Key. The external XMSS key structure encodes an XMSS based public key. We use this structure to represent accounts for which the local miner does not have a private key, and match signed transactions to their accounts.

- The *public seed* field contains a randomly generated seed value that is to be used for pseudo-random number generation when creating and validating signatures related to the public key address.
- The *public tree root* encodes the XMSS Merkle-tree root commitment that is used to authenticate messages signed by this public key.

Additionally, the following methods are provided.

- The *account address* instance property returns the chain account address that results from hashing both of the *public tree root* and the *public seed fields.*

One-time Signature. In this structure we encode the necessary authentication data which proves that a message was signed by the private key behind a specified public key. The encoded data corresponds to a WOTS+ signature, along with the XMSS authentication path data that associates the signature with the public key.

- The first field, *public key*, stores an External XMSS Key as previously described.
- The *message hash* represents the hash digest of the signed message.
- The *message randomizer* field represents a pseudo-randomly generated value used to protect the signature scheme against cryptographic attacks.
- The *wots index* field specifies the Merkle-tree index of the used one-time signature pair leaf.
- The *auth path* contains the Merkle authentication path which proves that the WOTS+ leaf is part of the root commitment.
- Lastly, the *sig ots* fields contains the WOTS+ signature data corresponding to the message digest.

Aside from its storage fields, the One Time Signature class also provides the following methods.

- The *is valid* instance method returns true if and only if the instance's fields represent a valid one-time signature by their specified public key on their specified message hash.
- The *create signature and update private key* class method takes as input an Internal XMSS Keypair instance, and a message hash, and returns both a One Time Signature instance on the provided message hash, and the resulting Internal XMSS Keypair instance after creating the signature.

Internal XMSS Key. The internal XMSS key datastructure corresponds to a stored XMSS private key. We use this structure to create one-time signatures, making sure to update the private key's state after every signature, as to remain in line with XMSS specifications.

- The *public key* field is used to store the public key corresponding to this private key, as the *public seed* value is necessary for signing.
- The *ots index* field stores which WOTS+ leaf should be used to sign the next message. This value is to be incremented by at least one each time a new message is signed, due to the one-time nature of the WOTS+ scheme.
- The *height* field indicates the height of the Merkle-tree used to store the WOTS+ key pairs. In essence, 2^{height} messages can be signed using this XMSS key.
- The *private key* field stores the randomly generated set of bytes that are used in the XMSS scheme to deterministically generate all of the WOTS+ private keys.
- The *private random hashing key* is used to deterministically generate randomizers for each signed message.
- Lastly, the *full tree* nodes field stores a full list of commitments which represent the full Merkle tree that the public tree root commits to.

In addition, the class contains the following methods.

- The *from signing instance* class method takes as input a native XMSS object, and returns a new Internal XMSS Key instance populated with the native object's data.
- The *to signing instance* instance method returns a new native XMSS object using the Internal XMSS Key instance's fields.
- The *from private* class method takes as input an ots index, a public seed, a private key, and a private random hashing key, and returns a new Internal XMSS Key instance using the provided input.

- The *create* class method returns a new Internal XMSS Key instance using freshly generated random data.
- The *account address* property returns the public key account address value.
- The *sign* instance method takes as input a message hash and returns both a One Time Signature instance on the provided message hash, and an updated Internal XMSS Keypair instance.

A.2.3 Ledger

We designed our Reward-All implementation to support an account-based transaction scheme. To accomplish this, we created several datastructures which decouple the data associated with accounts and transactions in each block from the data that is processed in the full-node for mining.

Account. This data structure specifies, for a public account address, all outgoing transactions, and total accrued balance. We use the data in this structure to check whether sufficient balance exists in an account when transferring coins to another one, and to credit the recipient with the appropriate amount of coins.

- The *address* field stores a 32-byte cryptographic commitment to the account's public key.
- The *outgoing transaction count* field stores the number of outgoing transfers made by this account.
- The *balance* field stores the account's total coin balance.
- The *highest sequence number* stores the highest sequence number used by this account for minting.

Transaction. Our Reward-All implementation of a transaction is designed to support two basic functionalities: transferring coins from a source account address to a destination, and minting new coins. Consequently, an instance of this datastructure may carry a significant amount of slab proving data.

- The *sender address* field refers to the source of the coin transfer, or the account to which newly minted coins will be credited.
- The *reference block hash* field is used to refer to a reference block which must be present in the blockchain that executes this transaction.
- The *sequence number* field is used to order this transaction with respect to the *sender address.* At its time of execution, each transaction's *sequence number* must be equal to the account's current *outgoing transaction count* value.
- The *destination address* specifies the account address of the recipient of the transferred coins.
- The *value transferred* field specifies how many coins will be transferred from the source to the destination.
- The *transaction fee* field is used to specify how many coins will be transferred from the *source address* to the miner who mines a block that executes this transaction.
- The *batches* field is used to specify the batches of slabs that will be used in minting in this transaction.
- The *signature* field is a one-time signature on the above fields by the *source address.*

Aside from storage, the Transaction class has the following methods.

- The *unsigned* instance method returns a new Transaction instance populated with the current instance's data, but with the signature field omitted.

- The *signed* instance method takes as input a One Time Signature instance, and returns a new Transaction instance populated with the current instance's data, and amended the provided input signature, after validating that signature corresponds to the unsigned version of the Transaction instance.

Block Header. We designed the header of a block to contain the data required for mining, smelting and minting. To support our Reward-All coin issuance model, we store block header instances which meet the local miner's minting difficulty, or commitments to batches thereof. The data in this structure adds up to only 160 bytes, sparing our implementation from having to store or provide transaction data for every slab.

- The *miner* field contains the account address of this block's miner.
- The *ancestor block hash* is used to refer to the hash of the K^{th} ancestor of this block.
- The *parent hash* refers to the hash of the block's immediate parent.
- The *subheader hash* is used to commit to the block's subheader.
- The *nonce* field contains the nonce used for mining.
- The *sequence number* and *reference block number* fields refer to the additional mining metadata required by Reward-All.

Block Subheader. The block subheader is what contains the data necessary for ledger progression. In this structure, we include the data authentication primitives using which a block commits to an immediate parent, to all of its ancestors, to a state trie root, and to the transactions it appends to the ledger. Furthermore, we implement our implementation's main difficulty adjustment algorithm in this structure using the methods in [IWSK21]. Essentially, this allows our subheader structure to be a self-contained indicator of where each block lies in the chain.

- The *height* field refers to the block's height in the chain.

- The *parent block hash* field commits to the block's immediate parent.
- The *parent tree hash* is used to commit to the Merkle tree which contains all of the blocks mined so far in the chain.
- The *state trie root* field commits to the root of a Merkle-Patricia Trie which maps each account address to a commitment of its account state.
- The *transaction tree root* field commits to a Merkle tree of all the transactions executed in this block.
- The *timestamp* field is used for timestamping the block.
- The *work* field stores the cumulative work done so far to mine this block's chain.

The class also provides the below methods.

- The *difficulty at* class method takes as input a block height and age, and returns the corresponding mining difficulty of the Subheader instance.
- The *difficulty* instance method takes as input a genesis block, and returns the corresponding mining difficulty of the Subheader instance.
- The *hashing target at* class method takes as input a block height and age, and returns the corresponding mining target of the Subheader instance.
- The *hashing target* instance method, takes as input a genesis block, and returns the corresponding mining difficulty for the Subheader instance.

Block Body. This datastructure contains only a list of individual commitments to all of the transactions that a block executes. We use this list to create the transaction tree commitment in the subheader.

- The *transaction ref list* is a list of commitments to the transactions contained in a block.

Block. Lastly, this structure combines the above decoupled structures to form a cohesive block of transactions.

- The first field is the block's *header*, which contains a Block Header datastructure.
- The second field is the block's *subheader*, containing a Block Subheader.
- The last field is the block's *body*, specifying a Block Body structure.

What's more, the class contains the following method.

- The *from byte chunks* class method takes as input a header, subheader, and body, all encoded as byte strings, and returns a new Block instance populated with the results of parsing each byte string.

A.2.4 Rewards

The implementation's Reward-All reward datastructures are designed to support Reward-All's mining, smelting and minting from a local miner's perspective and from a global blockchain viewpoint.

Accounted Batch. This datastructure operates from a global blockchain viewpoint, representing a batch of slabs that is attributed to an account in the chain.

- The *creator* field refers to the address of the account.
- The *commitment* field is the Merkle-tree root of the entire batch of slabs.
- The *starting reference block number*, *ending reference block number*, *starting sequence number*, and *ending sequence number* fields outline the reference blocks used for mining and the number of slabs per reference block, as specified in Reward-All.
- The *target* field is the slab mining target.

- The *proofs* field denotes the number of sample slabs successfully validated.

The datastructure class provides a set of methods as well.

- The *smelted value* instance method takes as input the number of proofs verified, and returns the number of coins that can be redeemed in exchange from this batch.
- The *total value* instance property returns the total number of coins that can be redeemed from fully validating the batch.
- The *width* instance property returns the number of reference blocks used in this batch.
- The *height* instance property returns the number of sequence numbers used in this batch.
- The *total size* instance property returns the total number of slabs in this batch.
- The *nth challenge* instance method takes as input a parameter n and returns the n^{th} slab that should be sampled from this batch to answer the n^{th} challenge.
- The *address* instance property returns the batch address of this instance used for storing this batch in the chain.
- The *address of* class method takes as input a batch creator address, a starting sequence number, and an ending sequence number and returns their corresponding batch address.

Header Batch. This structure is only used by a local miner to store its batches of slabs. The header batch datastructure is equivalent in specification to the Accounted Batch datastructure, but without the *proofs* field, the *address* instance property, or the *address of* class method.

Header Batch Proof. The header batch proof structure is also used locally. It is instantiated for each proof created to validate a slab batch.

- The *batch commitment* field refers to the Merkle-tree root commitment of the header batch.
- The *challenge* field denotes that the proof corresponds to the n^{th} challenge on the batch.
- The *block header* field stores the header used to answer the challenge.
- The *membership proof* field stores the Merkle inclusion proof that ties the *block header* to the *commitment* of the header batch.

Proposed Block Subheader. This datastructure is used by a local miner to hold the Block Subheader for the block it is actively proposing through mining.

- Its *height*, *parent block hash*, *parent tree hash*, *state trie root* and *transaction tree root* fields serve the same purpose as those of the Block Subheader datastructure.
- However, the *genesis timestamp* and *parent work* fields are used to store the information necessary to derive the cumulative work done in the block's chain at a given timestamp.
- The *genesis timestamp* field stores the timestamp of the genesis block of the chain being mined on.
- The *parent work* field stores the cumulative work done as of the parent block being mined on.

The class includes one method as well.

- The *to full header* instance method takes as input a timestamp and returns a Block Subheader instance populated with this instance's fields, the provided timestamp, and the additional work required to mine this subheader's block at the specified height and timestamp.

Proposed Block. Similar to the Block datastructure, this structure combines three fields to denote the block being currently proposed.

- The *header* field represents the Block Header datastructure of the proposed block.
- The *body* field represents the Block Body datastructure of the proposed block.
- The *subheader* field points to the Proposed Block Subheader structure of the proposed block.

Targeted Work Plan. This structure is used in Reward-All's implementation to locally mining plans, which denote which miner slabs are attributed to, and their difficulty.

- The *id* field denotes the mining plan's unique numeric number.
- The *miner* field contains the miner's account address.
- The *root height* field denotes the level at which the work data is compressed.
- The *target* denotes the slab mining target.

Alongside these fields, the following method is defined.

- The *difficulty* instance property returns the slab mining difficulty of the work plan.

Transacted Batch. This datastructure serves the same global purpose in the blockchain as the Accounted Batch. However, it is used by transactions to mint new coins through answering more challenges with new proofs.

- The *commitment*, *starting reference block number*, *ending reference block number*, *starting sequence number*, *ending sequence number*, and *target* fields are defined in the same way as those of the Accounted Batch structure.

- However, the *proofs* field in this structure contains a list of Transacted Batch Proof instances.

Like the Header Batch datastructure, the same six methods are provided.

Transacted Batch Proof. The Transacted Batch Proof is similar to the Header Batch Proof structure, but is used globally in the blockchain.

- The *ancestor block height* contains the ancestor block height of the slab.
- The *subheader hash* field contains a commitment to the subheader of the slab.
- The *parent block hash* field contains a commitment to the slab's parent block.
- The *nonce* is the nonce found to allow the header to qualify as a slab.
- The *membership proof* field includes the Merkle inclusion proof of this header within the root commitment of the batch.

A.2.5 Networking

The network datastructures are a simple collection of classes and enumerators designed to enable the implementation's peer-to-peer gossip protocol to function.

Peer Message Subject. We use this enumerator to determine the topic of a network message.

- *Greet* is the subject for peer introduction.
- *Advertise Block* is the subject for declaring that some block exists.
- *Solicit Block* is the subject for requesting a certain block.
- *Provide Block* is the subject for transferring a block's data.

Peer Message. This datastructure encodes a message sent by a peer.

- Its first field, *subject*, is an element from the Peer Message Subject enumerator.
- Its *length* field denotes the size in bytes of the message.
- The *contents* field contains the message.

Two methods are accessible in this structure.

- The *create* class method takes as input a Peer Message Subject instance and a byte string, and returns a new Peer Message instance using the provided parameters.
- The *create str* class method takes as input a Peer Message Subject instance and a string, returning a new Peer Message instance using the input.

Peer Message Template. This enumerator contains two basic frequently used Peer Message instances as elements.

- The *Greeting* template has a subject of *Greet* and encodes the string 'SA' as a message.
- The *Greeting Response* template also has a subject of *Greet*, and encodes the string 'AS' as a message.

Peer Conversation This enumerator describes, for a given peer, which conversation it is currently in.

- The *Intro* element denotes that the peer is in a greeting phase.
- The *Exchange* element denotes that the peer is in a data exchange phase.

Peer Type. We use this enumerator to determine whether the peer is from an incoming or outgoing connection.

- *Incoming* denotes that the peer initiated the connection to the local machine.
- *Outgoing* denotes that the local machine initiated the connection to the peer.

Peer. This datastructure combines all of the networking structures above to define a communication channel with a peer.

- The *host* and *port* fields contain the network connection information.
- The *peer type* field contains this peer's Peer Type element.
- The peer's *current conversation* field contains an element from the Peer Conversation enumerator denoting what conversation the peer's current messages fall under.
- The *advertised to block hash* field stores the latest block hash advertised by the local machine to this peer.
- The *latest block height*, *latest block hash*, and *latest block work* fields store the information provided by this peer about the latest block it holds.
- The *common prefix length* field stores the number of blocks in the locally known canonical chain that are accepted by the peer as canonical.
- The peer's *reputation* field numerically grades how well the peer behaves.
- The *stream reader* and *stream writer* fields store the network streams used for reading and writing messages.
- The *peer name* internal field is used to store the peer information in human readable form.
- Lastly, the peer's *lookup key* internal field is used to store a commitment to the peer's information that uniquely identifies it for in-memory lookups.

What is more, the following methods are provided.

- The *peer name* instance property returns the internal peer name field value, or the host and port concatenated in string format if the peer name field is not defined.
- The *lookup key* instance property returns the internal lookup key field value, or a hash of the host and port values if the internal lookup key field value is undefined.
- The *from name* class method takes as input a peer name string and a peer type, and returns a new Peer instance by parsing the provided name as a host and port.
- The *from rw pair* class method takes as input a network stream reader, a network stream writer, and a peer type, and returns a new Peer instance using the provided parameters.
- The *connect* instance method initiates a network connection to the peer.
- The *receive messages* instance method takes as input a message queue and attempts to read incoming messages into the queue.
- The *send message* instance method takes as input a Peer Message instance and attempts to send the message to the peer.
- The *close* instance method closes the peer's network stream writer.
- The *is closing* instance method returns whether the peer's network stream writer is being closed.
- The *send or pop* instance method takes as input a peer message instance and a cache instance and attempts to send the message to the peer, removing the peer using its lookup key if from the provided cache if the message delivery fails.
- The *set lcp* instance method takes as input a length parameter and sets the peer's common prefix length to the provided parameter.

A.2.6 UI Instructions

As the application is composed of multiple threads and coroutines, several queues are used for communication between different procedures. Elements passed in user-interface queues carry a respective UIInstructions enumerator value as an instruction type. These enumerators are categorized under *Mining*, *Batching*, *Wallet*, *UI*, and *Network* instructions.

Mining

- *Create Profile* denotes that the instruction is for creating a new local mining profile.
- *Mine Profile* denotes that the instruction is for starting or stopping mining using a local mining profile

Batching

- *Create Batch* denotes that the instruction is for creating a new batch of slabs for smelting.
- *Smelt Batch* denotes that the instruction is for toggling smelting on an existing batch of slabs.

Wallet

- *Create Account* denotes that the instruction requests a new internal account to be created.
- *Create Transaction* denotes that the instruction is for creating a new transaction to be published.

UI

- *State Update* denotes that the instruction carries a new application state to be rendered in the user-interface.
- *Explore* denotes that the instruction carries a ledger exploration query.

Network

- *Create Server* denotes that the instruction is for creating a new local server entry.
- *Enable Server* denotes that the instruction is for toggling listening for connections using a local server.
- *Create Peer* denotes that the instruction is for creating a new remote peer entry.
- *Enable Peer* denotes that the instruction is for toggling a connection with a remote peer.

A.3 Data Tables

This section presents the database tables created to support Reward-All's prototype implementation. Table A.2 shows the full scheme in brief form, while the remainder of this section elaborates on the purpose of each table and its fields.

Similar to Section A.2, the remainder of this section is split across *Trees*, *Authentication*, *Ledger*, *Rewards*, and *Networking* discussions. Each discussion introduces the set of data schemes designed to support the efficient storage and retrieval of their related datastructures.

Table A.2: Data storage scheme reference table. 14 table schemes, followed by 15 indices distributed across 8 tables, are shown below in a compact form. Each table and field name is prefixed by a number for easy lookup. Index fields are defined numerically in terms of the field numbers of the tables they reference.

Table Name			
Field	**Format**	**Key**	**Unique**
0. Accounts			
0) account hash	text	✓	✓
1) data	blob		
1. Accounted Batches			
0) batch hash	text	✓	✓
1) data	blob		
2. Aggregates			
0) root	text	✓	✓
1) tree	blob		
3. Weighted Work Batches			
0) commitment	text	✓	✓
1) miner	text		
2) starting ref. block number	int		
3) ending ref. block number	int		
4) starting seq. number	int		
5) ending seq. number	int		
6) target	text		
7) work plan id	int	10	
8) ending block hash	text	5	
4. Weighted Work Batch Proofs			
0) batch commitment	text	3	
1) challenge	small uint		
2) targeted header	blob		
3) membership proof	blob		
5. Blocks			
0) height	big int		
1) parent tree	blob		
2) block hash	text	✓	✓
3) genesis hash	text		
4) header	blob		
5) subheader	blob		
6) body	blob		
7) main chain	tiny int		
8) is tip	tiny int		
6. Internal Keypairs			
0) id	integer	✓	✓
1) account address	text		✓
2) data	blob		
7. Local Servers			
0) id	int	✓	✓
1) host	text		
2) port	int		
8. Remote Peers			
0) id	int	✓	✓
1) host	text		
2) port	int		
9. Trie Nodes			
0) node hash	text	✓	✓
1) node	blob		
10. Targeted Work Plans			
0) id	int	✓	✓
1) miner	text		
2) root height	tiny int		
3) target	text		
11. Targeted Work Headers			
0) work plan id	int	10	
1) header hash	text		
2) ancestor block hash	text	5	
3) parent hash	text		
4) subheader hash	text		
5) nonce	int		
6) sequence number	int		
7) reference block number	int		
12. Transactions			
0) transaction hash	text	✓	✓
1) transaction sender	text		
2) transaction destination	text		
3) transaction fee	big int		
4) data	blob		
13. Transaction Inclusions			
0) id	int	✓	✓
1) transaction hash	text	12	
2) block hash	text	5	

Indexed Table		
Index Name	**Fields**	**Unique**
3. Weighted Work Batches		
plan batches	7	
4. Weighted Work Batch Proofs		
proofs	0	
proof challenges	0, 1	✓
5. Blocks		
block height	0	
main chain	7	
main blocks	7, 0	
is tip	8	
7. Local Servers		
local servers	1, 2	✓
8. Remote Peers		
remote peers	1, 2	✓
10. Targeted Work Plans		
plans	1, 3	✓
11. Targeted Work Headers		
work	0, 2, 5, 6, 7	✓
work height	0, 5, 7	
13. Transaction Inclusions		
block transactions	2	
transaction blocks	1	
inclusions	1, 2	✓

A.3.1 Trees

We defined the below two tables for efficient node storage and lookup of our Binary Merkle Tree and Merkle-Patricia Tree datastructure implementation instances.

Aggregates Table. This table is designed to hold the Hash Tree datastructure as blobs, indexing them by the commitments to their roots.

- The *root* field stores the Merkle Tree root of the structure and acts as the unique primary key of the table.
- The *tree* field stores Hash Tree instances as blobs.

Trie Nodes Table. Similar to the Aggregates Table, but intended for Merkle-Patricia Trie nodes, this table stores Trie Nodes as blobs indexed by their hash commitments.

- The *node hash* field stores the commitment to the Trie Node.
- The *node* field stores a Trie Node instance as a blob.

A.3.2 Authentication

Long term efficient storage and retrieval of authentication credentials in our Reward-All implementation is only reserved for the local miner's private keys. Consequently, only Internal Keypairs have a dedicated table.

Internal Keypairs Table. The Internal Keypairs table stores the local miner's private keys as blobs.

- The *id* field is a numeric unique key created to index the keypair.
- The *account address* field stores the public account address related to the keypair.
- The *data* field stores an Internal XMSS Key datastructure instance as a blob.

A.3.3 Ledger

As our implementation must allow efficient context switching between chain forks, our account-based ledger storage scheme is designed to heavily leverage Merkle-Patricia Trie nodes for efficient hash-indexed storage and retrieval.

Accounts Table. The accounts table maps hash commitments of Account datastructures to their instances. Under this approach, given a hash of the account information, the full account data can be retrieved.

- The *account hash* field stores the hash of the instance data, and acts as the unique primary key of the table.
- The *data* field stores a blob of the Account datastructure instance.

Transactions Table. The Transactions table scheme, while similarly designed mainly a store for datastructure instances, also includes information about the transfer.

- The *transaction hash* field contains a hash of the transaction datastructure, and acts as the unique primary key of the table.
- The *transaction sender* field stores the transaction's sender account address.
- The *transaction destination* field stores the transaction's destination account address.
- The *transaction fee* field stores the number of coins paid as a fee to the transaction's miner.
- The *data* field stores the Transaction datastructure instance as a blob.

Blocks Table. The blocks table is used to permanently store blocks that have been mined as part of a chain.

- The *height* field denotes the block number in its chain.
- The *parent tree* field stores the root of the Merkle tree that commits to all blocks that precede this block.
- The *block hash* field stores the cryptographic commitment to the block and acts as the primary unique key of this table.
- The *genesis hash* field stores the seed hash of this chain.
- The *header* field stores the Block Header structure instance of this block as a blob.
- The *body* field stores a blob of the block's Block Body structure instance.
- The *main chain* field stores a binary value denoting whether the local miner considers this Block as part of the canonical chain.
- The *is tip* field stores a binary value denoting whether this block is the last block in its chain in view of the local miner.

In addition to the table's 8 fields, 4 additional indices are defined in its scheme to enable efficient block-related queries.

- The *block height* index creates an ordering only on the *height* field.
- The *main chain* index orders blocks based on their canonical status using only the *main chain* field.
- The *main blocks* index creates an ordering that utilizes both the *height* and *main chain* fields, allowing efficient lookups of canonical blocks using their height.
- The *is tip* index similarly uses only the *is tip* field to separate chain tips from other blocks.

Transaction Inclusions Table. Transactions are considered to have been included in a block when an entry linking the transaction hash to the block hash is inserted into this table.

- The *id* field acts as the unique primary key for the inclusion relationship between a transaction and a block.
- The *transaction hash* field is a foreign key entry used to reference a unique transaction from the Transactions table.
- The *block hash* field is similarly a foreign key that references a block from the Blocks table.

Other than these 3 fields, 3 indices are included in the scheme.

- The *block transactions* index uses the *block hash* field to enable efficient lookups of block's entire transaction set.
- The *transaction blocks* index uses only the *transaction hash* field to efficiently allow queries to find all blocks which include a transaction.
- The *inclusions* index is used to ensure that all pairs of *transaction hash* and *block hash* entries are unique.

A.3.4 Rewards

The rewards tables are designed to hold the data that supports both global and local mining, smelting, and minting operations.

Targeted Work Plans Table. This table stores the information required by a local miner to pursue its mining operations. Each entry represents a *targeted work plan*, which details the account address of the beneficiary miner, as well as the minting target.

- The *id* field is the table's unique numeric primary key used to identify different work plans.
- The *miner* field stores the account address of the beneficiary miner which resulting slabs are attributed to.
- The *root height* field is used to determine the Merkle tree height threshold at which slabs are stored.
- The *target* field denotes the slab mining target.

Additionally, one index is defined for this table.

- The *plans* index ensures that all *miner*, and *target* pairs are unique.

Targeted Work Headers Table. This table stores the resulting slabs that are produced by the local miner under each targeted work plan.

- The *work plan id* field is a foreign key that reference the work plan id that resulted in creating this work header.
- The *header hash* field stores the hash of the mined slab.

- The *ancestor block hash* field stores the hash of the K^{th} ancestor block of the mined slab.
- The *parent hash* field stores the hash of the immediate parent block of the slab.
- The *subheader hash* field stores the hash of the slab's Block Subheader.
- The *nonce* field stores the nonce found to satisfy the required proof of work.
- The *sequence number* stores the slab's sequence number.
- The *reference block number* stores the slab's reference block number.

Two indices are also defined for this table.

- The *work* index combines the *work plan id*, *ancestor block hash*, *nonce*, *sequence number*, and *reference block number* into a unique index that ensures no replicated work is stored.
- The *work height* index orders the *work plan id*, *nonce*, and *sequence number* fields for efficient smelting lookups.

Weighted Work Batches Table. This table is used to primarily store smelting metadata that enables a local miner to combine its locally stored slabs into a smelted proof.

- The *commitment* field stores the root of the Merkle tree commitment to the batch of slabs, and acts as the unique primary key of the table.
- The *miner* field stores the account address of the miner of the slabs.
- The *starting reference block number*, *ending reference block number*, *starting sequence number*, and *ending sequence number* specify the slab reference blocks and number of slabs per block.
- The *target* field stores the slab mining target.

- The *work plan id* stores a foreign key reference to the Targeted Work Plan that resulted in these headers.
- The *ending block hash* field stores a foreign key reference to the Block which appears at the height of the *ending reference block number* used.

In addition, one index is defined for this table.

- The *plan batches* index orders the table using the *work plan id* field, such that all batches related to a work plan can be efficiently retrieved.

Weighted Work Batch Proofs Table. This table is designed to support the local miner's incremental smelting procedure.

- The *batch commitment* field is a foreign key that references which Weighted Work Batch the proof is created for.
- The *challenge* field stores the challenge number that the proof answers.
- The *targeted header* field stores a blob of the sampled slab.
- The *membership proof* field stores the Merkle-tree inclusion proof of the *targeted header* in the *batch commitment.*

Furthermore, two indices are defined for this table.

- The *proofs* index orders the table using the *batch commitment* field for efficient lookups of all proofs stored for a batch commitment.
- The *proof challenges* index ensures that all *batch commitment*, and *challenge* pairs are unique, such that no redundant challenge answers are stored.

Accounted Batches Table. This table scheme provides a simple hash-based lookup table that allows access to global minting data.

- The *batch hash* field contains the hash commitment of the stored Accounted Batch instance.
- The *data* field stores a blob of the Accounted Batch instance.

A.3.5 Networking

The implementation's networking tables are designed to store the information required for a local miner to sustain long-term relationships with the peers it has discovered in the network.

Local Servers Table. The local servers table is designed to store the addresses and ports using which the local miner listens for connections.

- The *id* field is the table's unique primary key used to identify listening points.
- The *host* field defines the network interface address on which the local miner listens for a connection.
- The *port* field defines the port number on which connections are listened for.

Additionally, one index is specified for this table.

- The *local servers* index dictates that all *host*, and *port* pairs are unique in the table.

Remote Peers Table. This table stores remote connection points using which the local miner can connect to remote peers.

- The *id* field is the table's unique primary key used to identify remote peers.

- The *host* field defines the network interface address where the remote peer is located.
- The *port* field defines the port number on which the remote peer accepts connections.

Moreover, one index constrains this table.

- The *remote peers* index ensures uniqueness on all *host*, and *port* pairs.

A.4 Data Storages

Data tables in Reward-All's prototype are not directly queried by its algorithms, but are instead interfaced with using different storage classes. Each storage class exposes a set of methods for reading and writing data to the local miner's data tables.

One main storage class, named Storage, creates an instance of and allows access to each storage class, such that all methods need only access to a single instance of Storage, rather than separate access to each storage class. This single Storage instance is shared by all threads and procedures in Reward-All's process.

Storage is first instantiated with a database connection created by the main launching process. This connection is accessed using the Serialized Write DB class, which allows concurrent read queries to be performed, but ensures that write queries occur in a serialized manner. This is achieved by requiring that a lock is acquired by any thread that executes a query not prefixed with the word 'SELECT', and keeping queries confined to single statements.

Using its serialized connection, Storage instantiates all 9 storage classes, each of which will be discussed in the remainder of this section. In the same reductionist spirit as before, the classes are split across *Trees*, *Authentication*, *Ledger*, *Rewards*, and *Networking* discussions.

A.4.1 Trees

The core storage classes for tree structures in our implementation offer very direct storage and retrieval methods.

HT Storage. The Hash Tree Storage class provides access to the Aggregates table.

- The *create* class method takes a database connection as a parameter and returns a new instance of HT Storage after ensuring that the database contains the Aggregates table if it is not already defined therein.
- The *store aggregate* instance method takes a Hash Tree instance as a parameter and inserts it into the Aggregates table.
- The *store aggregate list* instance method takes a list of Hash Tree instances and inserts them into the Aggregates table.
- The *get aggregate* instance method returns the Hash Tree instance corresponding to the provided root hash parameter.
- The *get membership proof* instance method takes a root hash and a leaf index parameters, and returns the Merkle tree inclusion proof for the element located at the leaf index within the Merkle tree corresponding to the provided root hash.

PT Storage. The Pointer Storage class provides access to the Trie Nodes table.

- The *create* class method takes a database connection as a parameter and creates the Trie Nodes table if it does not already exist in the database, returning a new instance of the PT Storage class.
- The *store pointer* instance method takes a node hash and node parameter and stores the node under the node hash.

- The *store pointer map* instance method takes as input a dictionary of hashes that maps to Trie Node instances and stores each entry in the dictionary into the Trie Nodes table.
- The *get node* instance method takes a node hash as input and returns the Trie Node stored under said hash in the Trie Nodes table.

A.4.2 Authentication

Only a single storage class for Authentication is present in our Reward-All implementation.

Keypair Storage. The Keypair Storage class provides access to the Internal Keypairs table.

- The *create* class method takes as input a database connection and adds the Internal Keypairs table into the database if it does not exist, then returns a new Keypair Storage instance.
- The *add internal address* instance method takes an Internal XMSS Keypair instance and inserts it into the table.
- The *update internal address* instance method takes an Internal XMSS Keypair instance, and updates the data stored at its account address, returning True if the update was successful.
- The *get internal account addresses* instance method returns a list of the account addresses of all stored keypairs in the table.
- The *get internal account keypair* instance method takes as input an account address and returns the Internal XMSS Keypair instance stored for that address.

A.4.3 Ledger

Compared to the other storage classes, the ledger storage classes are the most involved.

Account Storage. This class handles access to both the Accounts and Accounted Batches tables.

- The *create* class method takes as input a database connection and ensures that both the Accounts and Accounted Batches tables exist before returning an new Account Storage instance.
- The *get account* instance method, given the hash of an account, returns the Account instance stored in the accounts table under that hash.
- The *get accounted batch* instance method similarly returns the Accounted Batch structure instance stored in the accounted batches table given the batch hash as a parameter.
- The *add account* instance method takes an account and an account hash as input and inserts the account into the accounts table under the given account hash.
- The *add accounted batch* instance method similarly adds an Accounted Batch structure instance to the accounted batches table using the Accounted Batch instance and its hash as input.

Transaction Storage. The Transaction Storage class enables interaction with both the Transactions table, and the Transaction Inclusions table.

- The *create* class method takes a database connection as input and creates both the Transactions and Transaction Inclusions tables, along with the Block Transactions Index, the Transaction Blocks Index, and the Inclusion Index. It returns a new instance of the Transaction Storage class.

- The *get transaction* instance method takes a transaction hash as input and returns the Transaction datastructure instance stored for that hash.
- The *get block transactions* instance method takes a block hash as input and returns all the Transaction datastructure instances stored that have been recorded as included in that block.
- The *store transactions* instance method takes a list of Transaction structure instances and stores them in the transactions table.
- The *store transaction inclusions* instance method takes a block hash and a list of transaction hashes as input, and stores a record in the transaction inclusions table between the block hash and each transaction hash.
- The *get transaction confirmation block hash* instance method takes a transaction hash and returns the canonical block hash which includes the transaction.
- The *get confirmed account transactions* instance method takes an account address as input and returns the list of Transaction structure instances denoting all transactions that have been sent by the account in the canonical chain.

Block Storage. The Block Storage class enables access to the blocks table.

- The *create* class method takes as input a database connection, and ensures that the blocks table, as well as the block height, main chain, main blocks, and is tip indices, all exist in the database, before returning a new Block Storage instance.
- The *get tips* instance method takes as input a boolean flag, and returns only the last block of the main chain if that flag is true, or returns all known chain tip blocks if the flag is false.
- The *block exists* instance method takes as input a block hash, and a boolean flag and returns true if the flag is set to true and a block with said hash is included in the main

chain, or true if the flag is false and a block with the input hash is known by the local miner but is not in the main chain. Otherwise it returns false.

- The *block known* instance method takes as input a block hash and returns true if the local miner has a block with said hash stored in the blocks table.

- The *block in main chain* instance method takes as input a block hash and returns true if a main chain block with said hash is stored in the blocks table.

- The *get block* instance method takes as input a block hash and a boolean flag and returns the main chain Block datastructure instance stored in the blocks table under the input block hash if the flag is true, the fork chain Block instance stored under the hash if the flag is false, or the Block instance with the given block hash regardless of its canonical status if the flag is not set.

- The *get block parent tree* instance method takes as input a block hash and returns the parent tree data stored for the block with that hash.

- The *get block height* instance method takes as input a block hash and a boolean flag and, if the flag is true, it returns the height of the main chain block with said block hash. Otherwise, if the flag is false, it returns the height of the non-canonical chain block with said block hash.

- The *get main chain block at height* instance method takes as input a block height and returns the stored Block datastructure instance with that height in the canonical chain from the blocks table.

- The *get main chain block hash* instance method takes as input a block height and returns the hash of the block at that height in the main chain from the blocks table.

- The *get block genesis hash* instance method takes as input a block hash and a boolean flag and returns the genesis hash of the block with the given block hash from the blocks table. If the flag is true, only main chain blocks are queried. If the flag is false, main chain blocks are excluded from the query. If the flag is unset, all blocks are queried.

- The *add block* instance method takes as input a Block instance, a list of hashes as a parent tree, and a boolean flag. The method appends appends the block as the new tip of its corresponding chain the blocks table. If the block is of height 0, it is treated as a genesis block. Otherwise, it is appended as the new tip of its chain. If the flag is set to true, it is treated as a main chain block.

- The *make chain auxiliary* instance method takes as input a starting height parameter and proceeds to set all main chain blocks with a height equal to or greater than the starting height as not belonging to the main chain.

- The *make block main tip* instance method takes as input a block hash and sets the main chain block stored with that hash as the new main chain tip.

A.4.4 Rewards

Given that the ledger storage classes provide sufficient functionality to carry out minting tasks, the implementation's reward storage classes encompass only the mining and smelting functionalities.

Targeted Work Storage. The Targeted Work Storage class enables access to the targeted work plans and targeted work headers tables, which store mining directives and their products.

- The *create* class method takes as input a database connection. Its primary task is creating the targeted work plans and targeted work headers tables according to there schemes, and creating the unique plan index, the unique work index, and the work height index. It ends by returning a new instance of the Targeted Work Storage method.

- The *row to work plan* internal class method takes as input an entry from the work plans table and returns a Targeted Work Plan datastructure instance.

- The *row to work header* internal class method takes as input and instance from the targeted work headers table and returns a Block Header structure instance.

- The *create work plan* instance method takes as input a miner account address, a Merkle tree storage root height, and a slab target and inserts into the targeted work plans table an entry with the parameter values.

- The *get all work plans* instance method returns a list of Targeted Work Plan datastructure instances populated by all of the entries in the targeted work plans table.

- The *get work plan by id* instance method takes as input a numeric work plan id and returns a Targeted Work Plan structure instance with the data stored for that id in the targeted work plans table.

- The *get work header count* instance method takes as input a numeric work plan id and a boolean main chain flag and returns the number of slabs mined under the work plan. If the main chain flag is set to true, only valid slabs are counted. Otherwise, all slabs for the work plan are counted.

- The *get work header hashes* instance method takes as input a work plan id, reference block number and sequence block number ranges, and a main chain boolean flag. It returns a list of all slab hashes mined under the work plan and within the two block number ranges. Similarly, the main chain flag constrains to query to only valid slabs if set to true.

- The *get work header* instance method takes as input a Targeted Work Plan instance, reference and sequence block numbers, and a main chain flag, and returns the slab designated by these parameters as a Block Header datastructure instance.

- The *save work header* instance method takes as input a Targeted Work Plan instance and a Block Header instance and saves the latter as a slab in the targeted work headers table.

- The *get ref seq schedule* instance method takes as input a Targeted Work Plan instance, an ending reference block number, and a schedule size N and returns the next N sequence and reference block number pairs to be used for mining under the specified plan and until the specified ending reference block number.

- The *size* instance method returns the total combined size in bytes of both the targeted work headers and targeted work plans tables.

Batch Storage. This class enables access to the two main smelting tables, the weighted work batches and weighted work batch proofs tables.

- The *create* class method takes as input a database connection and uses it to ensure that the weighted work batches and weighted work batch proofs tables are present in the database, along with the plan batch index, the batch proof index, and the batch proof commitment challenge index. It returns a new Batch Storage instance.
- The *row to batch* internal class method takes as input an entry from the weighted work batches table and parses it as a Header Batch datastructure instance, returning the instance.
- The *row to batch proof* internal class method takes as input an entry from the weighted work batch proofs table and returns a corresponding Header Batch Proof datastructure instance.
- The *save batch* instance method takes as input a Header Batch instance, a work plan id, and a block hash. It stores the Header Batch in the weighted work batches table under the input work plan id and using the block hash as the ending block hash of the entry.
- The *get batches* instance method returns all stored entries in the weighted work batches table as Header Batch instances.
- The *get batch* instance method takes as input a batch commitment and returns the corresponding Header Batch structure instance of its entry in the weighted work batches table.

- The *get batch work plan id* instance method takes as input a batch commitment and returns its corresponding work plan id.
- The *get batch ending block hash* instance method takes as input a batch commitment hash and returns the ending block hash corresponding to its entry in the weighted work batches table.
- The *get batch proofs* instance method takes as input a batch commitment and returns all of its corresponding entries from the weighted work batch proofs table as Header Batch Proof instances.
- The *get batch proof count* instance method takes as input a batch commitment and returns the number of entries that correspond to it in the weighted work batch proofs table.
- The *save batch proof* instance method takes as input a Header Batch Proof datastructure instance and stores it as an entry in the weighted work batch proofs table.
- The *size* instance method returns the total combined size in bytes of both the weighted work batches and weighted work batch proofs tables.

A.4.5 Networking

Networking related storage is only managed by the Networking Storage class.

Networking Storage. This class facilitates access to both the remote peers and local servers tables.

- The *create* class method takes as input a database connection and creates both the local servers and remote peers tables if they do not exist in the database. The local server index, and remote peers index, are also instantiated by the method. The method returns a new Networking Storage instance.

- The *add local server* instance method takes as input a host and a port and creates a new corresponding entry in the local servers table.
- The *get local servers* instance method returns a list of all entries in the local servers table.
- The *get local server* instance method takes as input a server id and returns its corresponding host and port entry from the local servers table.
- The *add remote peer* instance method takes as input a host and a port and creates a new corresponding entry in the remote peers table.
- The *get remote peers* instance method returns a list of all entries in the remote peers table.
- The *get remote peer* instance method takes as input a peer id and returns its corresponding host and port entry from the remote peers table.

A.5 Data Caches

Verifying and applying block state transitions requires repeated queries to several tables, especially the Trie Nodes table. Because of this, a cache layer was introduced to reduce block state transition burdens on the database.

All employed caches in the implementation are instances of the same class, the Hash Cache, which is designed to map a 32-byte hash value to any object instance. This is accomplished using the native python dictionary class, with the minor additions of an explicit *store value* method and a *is empty* method.

Pointer Cache. This cache instance is used to map hash commitments of Trie Nodes to their respective Trie Node structure instances.

Accounts Cache. This cache instance is used to map hash commitments of Accounts to their respective Account structure instances.

Accounted Batches Cache. This cache instance is used to map hash commitments of Accounted Batches to their respective Accounted Batch structure instances.

Work Plans Mining Cache. This cache instance is used to keep track of which work plans are currently being mined on.

Batches Smelting Cache. This cache instance is used to keep track of which slab batches are currently being smelted.

Local Servers Cache. This cache instance is used to track which local servers are currently being used to listen for connections.

Remote Peers Cache. This cache instance is used to track which remote peers are currently connected.

Heavier Chain Tips Cache. This cache instance is used to track which chain tips with more accumulated work than the local miner's canonical chain are currently sought after.

A.6 Data Managers

Data Manager classes in our implementation are akin to stored procedures for databases. They provide more complex data storage functionality than what the relational database query language is able to directly offer. These classes are defined to contain a set of class methods that execute pieces of application logic that are either heavily reused across different executive functions or require too much direct data manipulation to carry out. This keeps the executive portion of the implementation free from redundancy and complex low-level code.

A.6.1 Pointer Manager

This class manages storing and retrieving Trie Nodes from Cache and Storage instances.

- The *cache null pointer* class method takes as input an optional Cache instance and returns the hash commitment of an empty Merkle-Patricia Trie root, storing it in the cache instance if provided.

- The *cache ref pointers* class method takes as input a list of reference and an optional Cache instance, and returns the hash commitment of the root of a Merkle-Patricia Trie that maps the references to themselves, storing all Trie nodes in the cache instance if provided.

- The *cache key pointers* class method takes as input a list of 32-byte keys, a list of 32-byte values, and an optional Cache instance, and returns the hash of the root of the Merkle-Patricia Trie that maps all provided keys to the provided values, storing the Trie nodes in the Cache if provided.

- The *get node* internal class method takes as input a node hash, a Storage instance, and a Cache instance, and returns the Trie Node corresponding to the provided node hash, searching first in the Cache instance, and subsequently in the Storage instance if the cache lookup failed.

- The *get pointer with trail* internal class method takes as input a node hash, a 32-byte key, a Storage instance, and a Cache instance, and returns both the value stored under the provided key in the Merkle-Patricia Trie whose root commitment is the provided node hash as well as all Trie Nodes visited from root to leaf during the lookup in the provided Storage and Cache instances.

- The *get pointer* class method takes as input a node hash, a 32-byte key, a Storage instance, and a Cache instance, and returns the value stored under the provided key in the Merkle-Patricia Trie whose root commitment is the provided node hash using the input Storage and Cache instances.

- The *update pointers with changes* class method takes as input a node hash, a 32-byte key, a 32-byte value, a Storage instance, and a Cache instance, and updates the Merkle-Patricia Trie whose root commitment is the specified node hash to store the input value at the provided key parameter, saving all resulting Trie Node modifications in the Cache.

- The *get cached pointers* internal class method takes as input a node hash, and a Cache instance, and returns all of the internal trie nodes and leaves that are stored in the cache under the Merkle-Patricia Trie whose root hash is the provided node hash.
- The *commit cached pointers* class method takes as input a node hash, a Storage instance, and a Cache instance, and copies the nodes of the Merkle-Patricia Trie stored in the Cache instance under the provided node hash to the provided Storage instance, returning the leaves of the Trie.

A.6.2 Account Manager

This class provides many functionalities that allow the creation, update, storage, and retrieval of account data from cache and storage.

- The *create internal account* class method takes as input a Storage instance and inserts a freshly generated private key into the instance's Keypair Storage.
- The *save account* class method takes as input a state pointer hash, an Account instance, a Storage instance, and a Cache instance, and returns the hash of the resulting Merkle-Patricia Trie root after storing a hash of the Account instance under the account address as a key in the Merkle-Patricia Trie whose root is the provided state pointer hash, saving the newly resulting Trie Nodes in the provided cache.
- The *save accounted batch* class method takes as input a state pointer hash, an Accounted Batch instance, a Storage instance, and a Cache instance, and returns the hash of the resulting Merkle-Patricia Trie root after storing a hash of the Accounted Batch instance under its address as a key in the Merkle-Patricia Trie whose root is the provided state pointer hash, saving the newly resulting Trie Nodes in the provided cache.
- The *create account* class method takes as input a state pointer hash, an account address, a Storage instance, and a Cache instance, and returns the hash of the resulting

Merkle-Patricia Trie root after storing, under the account address as a key, a hash of a new Account instance with the provided address and no transaction history or balance in the Merkle-Patricia Trie whose root is the provided state pointer hash, saving the newly resulting Trie Nodes in the provided cache.

- The *create accounted batch* class method takes as input a state pointer hash, an account address, a set of slab batch parameters, a Storage instance, and a Cache instance, and returns the hash of the resulting Merkle-Patricia Trie root after storing, under the batch address as a key, a hash of a new Accounted Batch instance with the provided parameters and no proof history in the Merkle-Patricia Trie whose root is the provided state pointer hash, saving the newly resulting Trie Nodes in the provided cache.

- The *get account* class method takes as input an account hash, a Storage instance, and a Cache instance, and returns the Account structure instance saved in the Cache instance, or retrieved from the Storage instance if the cache lookup fails.

- The *get accounted batch* class method takes as input an accounted batch hash, a Storage instance, and a Cache instance, and returns the Accounted Batch structure instance saved in the Cache instance, or retrieved from the Storage instance if the cache lookup fails.

- The *get or create account* class method class method takes as input a state pointer hash, an account address, a Storage instance, and a Cache instance, and returns the cached or stored Account instance with the given address under the Merkle-Patricia Trie tree root with the input state pointer hash. If the account does not exist in the given Trie, the account is inserted into the Trie, and an updated Trie root hash is additionally returned.

- The *get account at address* class method takes as input a state pointer hash, an account address, a Storage instance, and a Cache instance, and returns the cached or stored Account instance with the given address under the Merkle-Patricia Trie tree root with the input state pointer hash.

- The *get or create accounted batch* class method takes as input a state pointer hash, an account address, a set of slab batch parameters, a Storage instance, and a Cache instance, and returns the cached or stored Accounted Batch instance with the given parameters saved under the Merkle-Patricia Trie tree root with the input state pointer hash. If the batch does not exist in the given Trie, the batch is inserted into the Trie, and an updated Trie root hash is additionally returned.

- The *update account* class method takes as input a state pointer hash, an Account instance, a Storage instance, and a Cache instance, and updates the Merkle-Patricia Trie whose root is the provided state pointer hash such that the hash of the provided Account instance is stored under the account address as a key, returning the resulting Trie root after the update, and saving all new intermediate nodes in the cache instance.

- The *update accounted batch* class method takes as input a state pointer hash, an Accounted Batch instance, a Storage instance, and a Cache instance, and updates the Merkle-Patricia Trie whose root is the provided state pointer hash such that the hash of the provided Accounted Batch instance is stored under the batch address as a key, returning the resulting Trie root after the update, and saving all new intermediate nodes in the cache instance.

- The *commit cached accounting updates to storage* class method takes as input a list of hashes, referring to cached Account or Accounted Batch instances, a Storage instance, and a Cache instance, and copies all Accounts and Accounted Batches saved under the supplied hashes in the input Cache instances to their respective tables in the provided Storage instance.

A.6.3 Block Manager

The Block Manager provides a set of methods for safely saving and retrieving valid Blocks and their data.

- The *get main chain tip* class method takes as input a Storage instance and returns the last block in the locally stored canonical chain.
- The *get main chain genesis* class method takes as input a Storage instance and returns the genesis block of the locally stored canonical chain.
- The *create genesis block* class method takes as input a Storage instance and a timestamp and locally creates a new genesis block with the supplied timestamp.
- The *append block* internal class method takes as input a Storage instance, a Block instance, the block's genesis block, the block's parent block, the parent block's parent tree, the list of Transaction instances of the Block, and a main chain flag. The method verifies that the provided block is valid with respect to the chain it extends, appending it after the specified parent either as a canonical or auxiliary block, and returning True if all validations are successful.
- The *append main chain block* class method takes as input a Storage instance, a Block instance, and a list of Transactions, and appends the provided Block to the locally stored canonical chain, returning True if the block is valid.
- The *append auxiliary block* class method takes as input a Storage instance, a Block instance, and a list of Transactions, and appends the provided Block to a locally stored auxiliary chain, returning True if the block is valid.

A.6.4 Work Manager

This class provides a small set of routines for mining and smelting functionalities.

- The *create work plan* class method takes as input a miner address, and a Storage instance, and creates a new local mining plan with default parameters in the database.
- The *check batch chain validity* class method takes as input a batch commitment hash, and a Storage instance, and returns True if the last used reference block hash for the provided batch is part of the canonical chain.

A.7 Threads

From a high-level point of view, the codebase offers a single executable python script, which runs a multi-threaded application comprised of three main threads that operate under a single process. Each thread runs a set of coroutines, which are a set of asynchronous functions that can operate concurrently in the same thread that hosts them.

The first thread, referred to as the primary thread, hosts the main set of coroutines which offer the main blockchain functionality. These coroutines encompass not only mining, smelting, and minting, but also local wallet management, new block proposal, and main chain selection.

The second thread hosts the set of coroutines associated with the networking procedures that enable the peer-to-peer overlay to gossip about the state of the blockchain. These coroutines take care of listening for, and establishing connections with remote peers, along with processing incoming and outgoing network messages.

The last thread hosts the Text-User-Interface (TUI) rendering procedures, which collectively ensure that the terminal allows the user to visually interact with information displays that are up to date with the current internal application and blockchain state.

This mixture between parallel threads and concurrent coroutines permits the implementation to follow a design whereby shared memory concerns, such as deadlocks and race-conditions, can be easily avoided.

A.7.1 Primary Thread

Initially, the primary thread encompassed all of the implementation coroutines. However, as networking and user-interface operations became increasingly complex and demanded lengthy input and output waiting times, they were phased out of the primary thread. Consequently, the primary thread still encompasses all of the main application logic, executing all of mining, smelting, minting, block proposal, consensus, and user-interface state update procedure as python coroutines in a single asynchronous event loop.

In the remainder of this section, each of the six primary thread coroutines are presented and explained in terms of its requirements and role in our implementation.

User-Interface State Coroutine. This coroutine runs regularly every 500 milliseconds by default[1], and mainly updates the application-state information that is used to render the user-interface.

The first main task in this procedure is to process all messages present in the *ui explorer fetch* queue. These messages denote queries by the ledger exploration interface, which allows the end-user to retrieve information on blocks, accounts, and transactions.

The second main task is querying the application storage and retrieving the up-to-date state information that needs to be displayed to the user. On each invocation of this coroutine, a new key-value store instance is created, and passed through the *ui state update* queue. This key-value store is comprised of the following keys:

- *db:* The full database size in bytes.
- *latest block:* The Block instance representing the tip of the canonical chain.
- *genesis block:* The Block instance representing the genesis block of the canonical chain.
- *work plans:* A list of Targeted Work Plan instances representing all stored local miner work plans.
- *work plan header counts:* A list of the number of slabs found for each work plan.
- *work size:* The byte size of the slab and work plan storage tables.
- *batches:* A list of Header Batch instances representing all stored smelted slab batches.
- *batch in main chain:* A list of boolean flags for each batch denoting whether it is valid with respect to the canonical chain.

[1]The DEFAULT_TUI_REFRESH_TIME constant determines this value.

- *batch proofs:* A list generated proof count for each smelted batch.
- *batches size:* The storage size in bytes occupied by smelted slab batches and their proofs.
- *explored data:* The ledger explorer data resulting from processing the latest *ui explorer fetch* queue message.
- *internal addresses:* A list of public account addresses of all stored internal keypairs.
- *local servers:* The list of all stored local servers.
- *remote peers:* The list of all stored remote peers.
- *consensus peers:* The number of connected remote peers that share the same canonical chain tip with the local miner.
- *connected peers:* The number of connected remote peers.

Mining Coroutine. This coroutine performs two main functions within the primary thread. Firstly, it processes messages from the *ui instruction mining* queue. Second, it launches mining tasks for Targeted Work Plan instances waiting in the *work plan* queue.

Messages from the *ui instruction mining* queue are either for creating new mining profiles, or toggling active mining on existing profiles. In the first case, a new Targeted Work Plan is created for the specified miner. In the latter, the state of an existing work plan is toggled between active and inactive, which adds or removes it from the *work plan* cache.

Each Targeted Work Plan instance present in the *work plan* cache periodically appears in the *work plan* queue. For each instance in said queue, the coroutine launches a mining tasks, which decides the next reference and sequence numbers to be used for mining under the work plan, and launches a separate thread which actively searches for slabs and blocks using that information. The work plan instance is put back in the queue once a slab or block is found, and the process is repeated as long as the work plan is active.

Smelting Coroutine. This coroutine similarly performs two main functions, processing messages from the *ui instruction batching* queue, and launching smelting tasks for batches in the *smelt vault* queue.

ui instruction batching queue messages instruct the coroutine to either designate a collection of slabs as a new batch, creating a new corresponding Header Batch instance and saving it in storage, or to toggle whether proofs for an existing batch are being actively smelted.

Header Batch instances that require new proofs to be generated are marked as active by simply having an entry in the *batches smelting* cache, and periodically appearing in the *smelt vault* queue. Each time the instance appears in the queue, the coroutine creates a new smelting task that, upon completion, places the instance back in the queue and returns the next proof to be created upon completion. As long as the slab batch is marked as active, this process is repeated.

Wallet Coroutine. The Wallet Coroutine is dedicated to processing instructions received from the user-interface related to internal keypairs. The instruction types supported by this coroutine are either transaction signing using known internal accounts or new account creation.

When a transaction is signed, the signing key is updated to utilize an updated nonce for its next signature, and the resulting signed transaction is then passed to the local miner's transaction memory pool. Otherwise, when a new account is created, a new random internal keypair is saved to storage.

Block Proposal Coroutine. This coroutine handles the main task of ensuring that a proposed block is always available for the local miner as a basis on which to launch any new mining tasks. It ensures that a global variable named *current block proposal* contains a ProposedBlock instance that greedily attempts to mine the highest-fee transactions currently present in the miner's pending transactions memory pool on top of the local miner's current canonical chain tip.

Consensus Coroutine. This coroutine regularly monitors the *block introduction* queue for newly appended blocks to the local miner's storage. For each new block passed into the queue, the coroutine revisits its view of the canonical chain, either adopting the block as the new canonical chain tip, or ignoring it as an auxiliary block. The procedure only adopts a new canonical chain if the local miner has all of its blocks and it is indeed the heaviest chain.

A.7.2 Networking Thread

Reward-All's networking thread enables a set of simple peer-to-peer communication coroutines that allow the local miner to advertise and share its blocks with interested parties.

Networking Instructions Coroutine. The Networking Instructions Coroutine handles user-interface instructions related to networking, which fall under one of only four possible types of instructions.

The first type is server creation, where the user-interface provides the local address and port on which to listen for connections. Upon receipt of this instruction, the coroutine creates a new storage entry with the provided information.

The second type is peer creation, where the remote address and port to which to connect is provided. This instruction leads the coroutine to create a new database entry for the provided remote peer information in storage.

The third type is server toggling. This instruction refers to a pre-existing local server entry, and leads the coroutine to either start listening for new connections using the local server information if the local server is not being actively listened on, or to close all active connections made using the local server if it is already active.

Lastly, the fourth instruction type is peer toggling, which initiates a new connection or terminates the existing connection with a known remote peer.

Network Messages Coroutine. This coroutine processes all incoming messages from peers connected over the network. Peers may be in one of two conversation phases, the Greeting phase or the exchange phase. In the former, peers are only allowed to greet each other through simple handshake messages. After the handshake is complete, a peer is considered by the local miner to be in the Exchange conversation phase, which allows the peer to transmit chain-related messages.

The first incoming message type is for block advertisement, which is used by a peer to announce its canonical chain tip to its neighbors. If the advertised block is invalid, already known, or does not contain more work than the local miner's canonical chain tip, the message is ignored. Otherwise, the advertised block is cached and the local miner begins to solicit its ancestors if they are not already known.

The second message type is for block solicitation, in which a peer asks for a specific block from the local miner's canonical chain. If the block is known by the local miner to be in the canonical chain, the local miner responds with the block's contents.

The third message type is block provision, whereby a peer provides a requested block to the local miner. If the block is not sought after by the local miner, the message is dropped. Otherwise, the block is appended to the local miner's chain database, and the *Network Synchrony Possible* event is fired.

Synchrony Coroutine. The Synchrony Coroutine is designed to drive the consensus process between a local miner and its peers through sending block solicitations and advertisements. This coroutine waits for the *Network Synchrony Possible* event to be fired, and once the event is set, it examines the local miner's view of its blocks and peers. The list of chain tips that are sought after by the local miner is first pruned to include only those with the most amount of work. Subsequently, each remote peer is sent a block advertisement for the local miner's latest block, and is solicited for the next block in the chain.

A.7.3 Text User-Interface Thread

This thread is in charge of rendering and receiving instructions from Reward-All's Text User-Interface (TUI), which is built using python's 'urwid' library, and enables users to interact visually with the application from within the console. Unlike the other threads, the TUI thread does not use coroutines, due to urwid's design. Instead, only urwid's main rendering loop is periodically iterated.

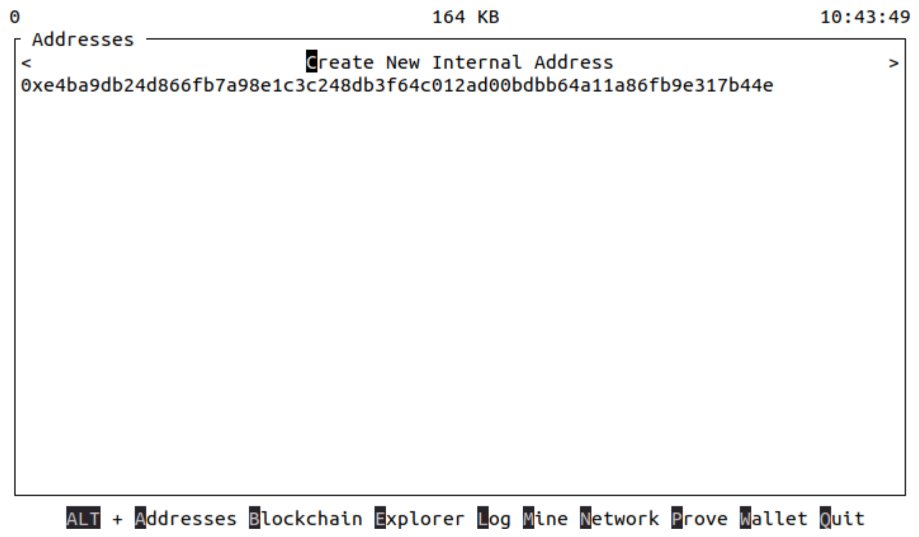
To create this loop, the thread first defines the user-interface layout using urwid's object. Then, the color palette of the interface is also defined, along with the input handling logic for any interaction which the layout objects could not capture or interpret. Using these three elements, the loop is then defined to render the interface on the default output screen provided by the host console. To keep the user-interface contents up to date with the application state, a periodic state update task is run by the loop, by default twice every second.

Layout Construction. The TUI layout consists of three main components arranged in a vertical stack, and each taking up the full width of the application window.

- The header, which shows the current block height, storage usage, and local time.
- The main body, which shows the currently selected view.
- The footer, which displays the keyboard shortcuts for switching between views.

The header and footer take up one line of text each to render, while the main body takes up the remaining vertical space in the application window to show the currently selected view. There are eight views to choose from.

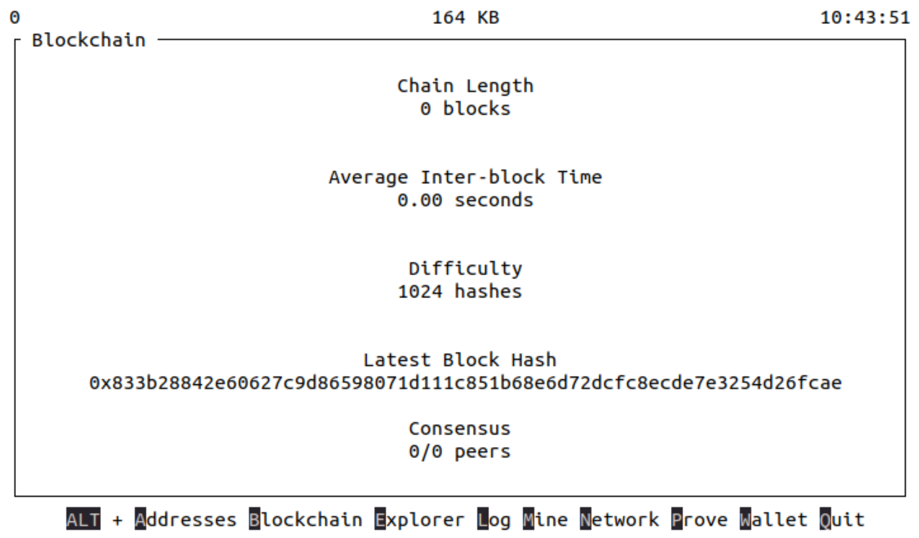
Addresses View. This view shows a list of all locally stored internal addresses, as well as enables the creation of a new random internal address.



```
0                              164 KB                              10:43:49
┌ Addresses ─────────────────────────────────────────────────────────────┐
│<                      Create New Internal Address                     >│
│0xe4ba9db24d866fb7a98e1c3c248db3f64c012ad00bdbb64a11a86fb9e317b44e      │
│                                                                        │
└────────────────────────────────────────────────────────────────────────┘
    ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.1: Addresses

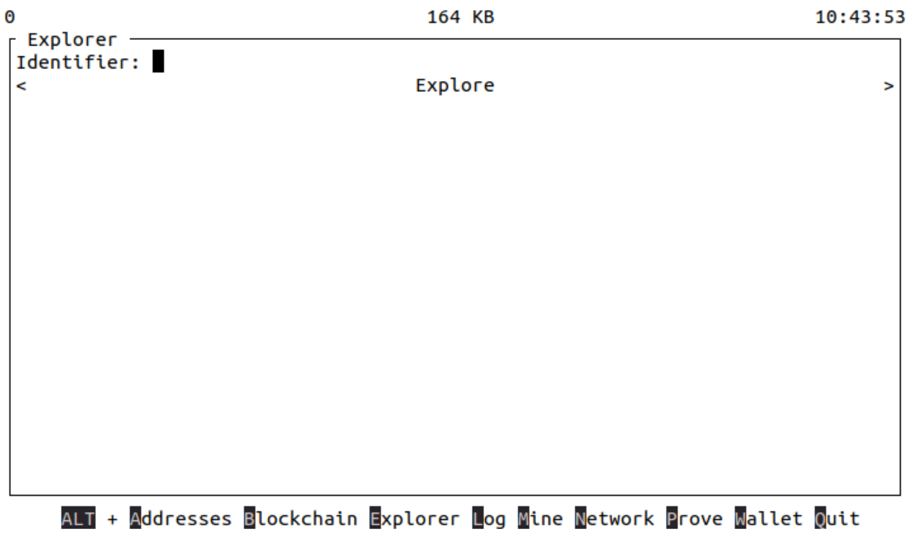
Blockchain View. This view shows the current state of the canonical chain. It lists the chain length, average inter-block arrival time, the current mining difficulty, the latest block hash, and the number of peers that share the same canonical chain tip.



```
0                              164 KB                              10:43:51
┌ Blockchain ────────────────────────────────────────────────────────────┐
│                             Chain Length                               │
│                               0 blocks                                 │
│                                                                        │
│                        Average Inter-block Time                        │
│                             0.00 seconds                               │
│                                                                        │
│                              Difficulty                                │
│                             1024 hashes                                │
│                                                                        │
│                           Latest Block Hash                            │
│   0x833b28842e60627c9d86598071d111c851b68e6d72dcfc8ecde7e3254d26fcae   │
│                                                                        │
│                               Consensus                                │
│                              0/0 peers                                 │
└────────────────────────────────────────────────────────────────────────┘
    ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.2: Blockchain

Explorer View. The explorer takes as input an identifier and returns the data associated with it. If the identifier is purely numeric, the explorer returns the block stored at the height specified by the identifier. Otherwise, if a hexadecimal identifier, the explorer returns the account, transaction, or block, associated with the identifier.

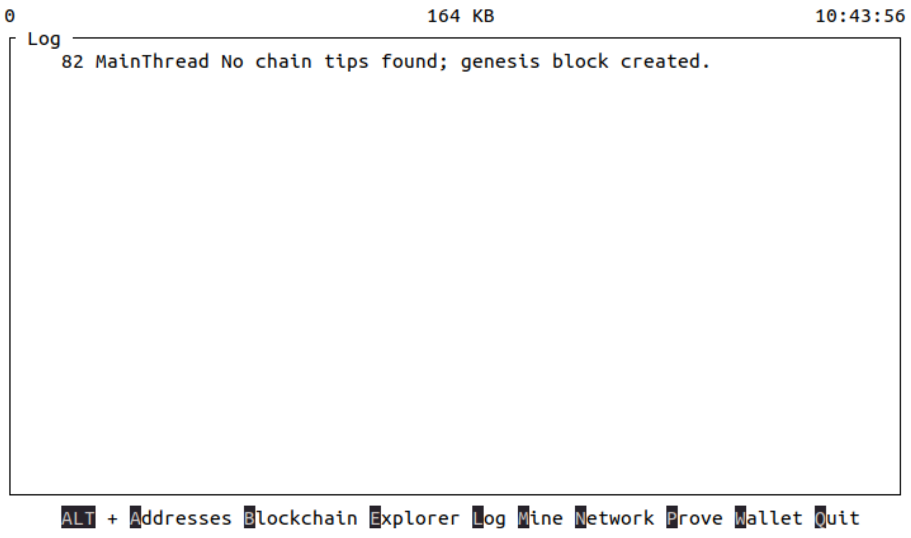


```
0                                 164 KB                            10:43:53
┌ Explorer ──────────────────────────────────────────────────────────────┐
Identifier: █
<                                 Explore                                >
```

ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit

Figure A.3: Explorer

Log View. This view shows the local miner's log data, which contains information relevant for debugging and introspection of application behavior.

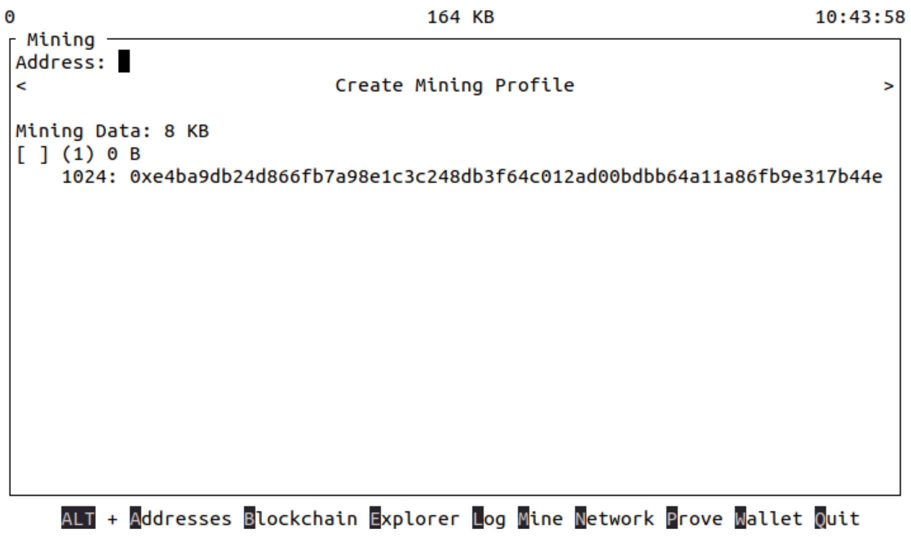


```
0                                 164 KB                            10:43:56
┌ Log ───────────────────────────────────────────────────────────────────┐
    82 MainThread No chain tips found; genesis block created.
```

ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit

Figure A.4: Log

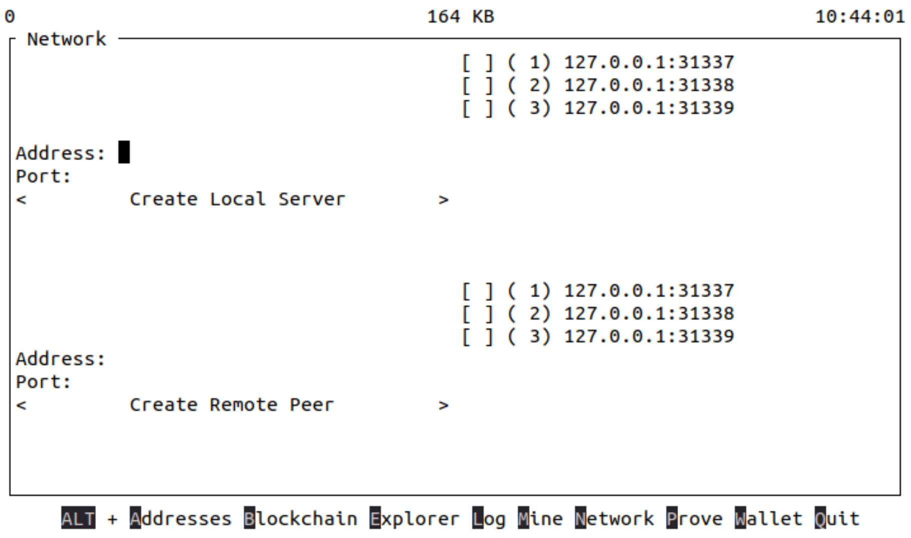
Mine View. This view shows the list of targeted work plans stored by the local miner, and allows the creation of new work plans. For each work plan listed, the view allows the user to toggle mining using the specified work plan. Furthermore, given an account address, the view enables the creation of a new work plan.



```
0                                 164 KB                              10:43:58
┌ Mining ─────────────────────────────────────────────────────────────────────┐
│Address: █                                                                   │
│<                          Create Mining Profile                            >│
│                                                                             │
│Mining Data: 8 KB                                                            │
│[ ] (1) 0 B                                                                  │
│    1024: 0xe4ba9db24d866fb7a98e1c3c248db3f64c012ad00bdbb64a11a86fb9e317b44e │
└─────────────────────────────────────────────────────────────────────────────┘
    ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.5: Mining

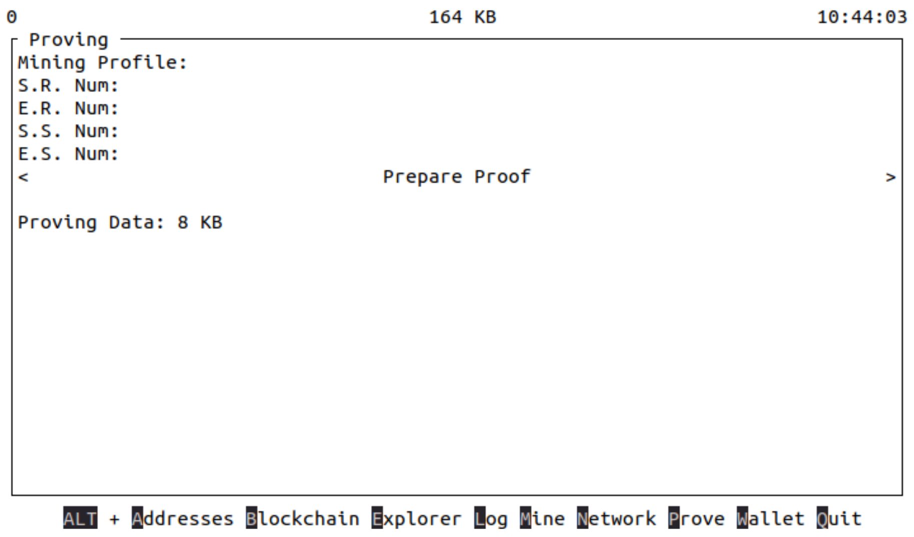
Network View. This view shows two lists, one containing all local servers supported by the local miner, and all remote peers known by the miner. For each local server, the user can toggle accepting connections, and for each remote peer, the user can toggle the connection with that peer. Moreover, given a local server or remote peer address and port, a new entry can be created.



```
0                                 164 KB                              10:44:01
┌ Network ────────────────────────────────────────────────────────────────────┐
│                                      [ ] ( 1) 127.0.0.1:31337               │
│                                      [ ] ( 2) 127.0.0.1:31338               │
│                                      [ ] ( 3) 127.0.0.1:31339               │
│                                                                             │
│Address: █                                                                   │
│Port:                                                                        │
│<         Create Local Server        >                                       │
│                                                                             │
│                                                                             │
│                                      [ ] ( 1) 127.0.0.1:31337               │
│                                      [ ] ( 2) 127.0.0.1:31338               │
│                                      [ ] ( 3) 127.0.0.1:31339               │
│Address:                                                                     │
│Port:                                                                        │
│<         Create Remote Peer         >                                       │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘
    ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.6: Network

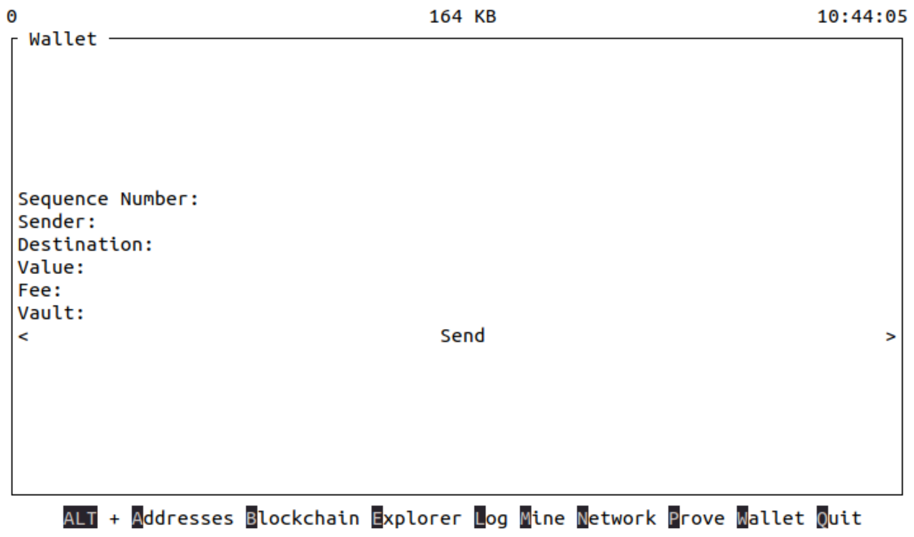
Proving View. This view shows a list of all slab batches stored by the local miner, as well as enables the creation of new batches. For each batch shown, the view allows toggling smelting on the selected batch.



```
0                                   164 KB                                10:44:03
┌ Proving ─────────────────────────────────────────────────────────────────────────┐
│Mining Profile:                                                                   │
│S.R. Num:                                                                         │
│E.R. Num:                                                                         │
│S.S. Num:                                                                         │
│E.S. Num:                                                                         │
│<                               Prepare Proof                                    >│
│                                                                                  │
│Proving Data: 8 KB                                                                │
└──────────────────────────────────────────────────────────────────────────────────┘
     ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.7: Smelting

Wallet View. This view enables the user to create a new transaction and insert it into the local miner's transaction pool. Given the transaction sequence number, sender address, destination address, transfer amount, and fee, the new transaction can be defined. Optionally, the user can provide the address of a smelted batch in the vault field to leverage newly minted coins in the transaction.



```
0                                   164 KB                                10:44:05
┌ Wallet ──────────────────────────────────────────────────────────────────────────┐
│                                                                                  │
│Sequence Number:                                                                  │
│Sender:                                                                           │
│Destination:                                                                      │
│Value:                                                                            │
│Fee:                                                                              │
│Vault:                                                                            │
│<                                   Send                                         >│
│                                                                                  │
└──────────────────────────────────────────────────────────────────────────────────┘
     ALT + Addresses Blockchain Explorer Log Mine Network Prove Wallet Quit
```

Figure A.8: Wallet

Input Handling. For all inputs not handled by the loop's main Frame object, the *Handle Control Input* method gets invoked. We designed this method to handle switching between different parts of the interface, and exiting the application altogether. The following key combinations, and their lower case variants, are valid:

- **ALT + Q** Exits the application.
- **ALT + A** Switches to the addresses view.
- **ALT + B** Switches to the blockchain view.
- **ALT + E** Switches to the explorer view.
- **ALT + L** Switches to the log view.
- **ALT + M** Switches to the mining view.
- **ALT + N** Switches to the network view.
- **ALT + P** Switches to the smelting view.
- **ALT + W** Switches to the wallet view.

State Updates. All of the information displayed in the TUI is based on the current application state. However, the TUI's access to reading the current state is indirect, where the only exception is the application log data, which the TUI reads directly. The rendered information gets updated using the *sync tui with state* method, which gets periodically executed every 500 milliseconds by default. The method begins by updating the current log view with all new entries added to the *log record queue*. Subsequently, the method fetches the latest state update provided in the *ui state update queue*, and uses it to update the data displayed in the remaining views.