Blockchain Light Clients from Proofs of Sequential Work Schemes

Master in Formal Methods in Computer Science
Complutense University of Madrid
School of Computer Science
2024/2025

Jorge González

Directed by
Hamza Abusalah and Pedro Moreno-Sanchez

June 2025

Abstract

As cryptocurrencies have become more popular, it has become more common to operate them from mobile devices. This can be problematic, since blockchains can take hundreds of gigabytes in disk storage. Typically, this is solved by connecting to some supplier that holds the blockchain, but that contradicts the decentralized nature of cryptocurrencies. It would be interesting to make this connection trustless.

In this work, we will focus on the FlyClient protocol, presented by Luu, Bünz, Kiffer and Zamani in 2019. This protocol allows light clients, *i.e.* users that do not have the blockchain stored locally, to verify cryptocurrency transactions by downloading a very small subset of blocks. This work has raised public interest as it presents a simple and efficient solution to the mentioned problem. However, it suffers from some problems. Mainly, there is a general lack of formalization along the paper which makes the exact functionality of the protocol unclear. Moreover, the security guarantees of the protocol are depicted in an extremely vague way.

We have formalized the protocol as a proof system. Furthermore, we have explored the connection between FlyClient and Cohen and Pietrzak's Proof of Sequential Work scheme, using this to specify the security guarantees of the FlyClient protocol.

Finally, we have reviewed the most well known alternatives to this problem, namely Non-Interactive Proofs of Proof-of-Work (NIPoPoWs, from now on) and Succint Non-Interactive Arguments of Chain Knowledge (SNACKs, from now on), and compared their features and guarantees to FlyClient's. We have concluded that FlyClient has remarkable advantages with respect to NIPoPoWs, mainly in the form of compatibility with variable difficulty blockchains. On the other hand, we have noted many similarities between FlyClient and SNACKs, with SNACKs being silghtly more efficient and versatile.

Keywords. Blockchains, Light clients, Proofs of Sequential Work

Resumen

A medida que las criptomonedas se han popularizado, se ha vuelto más frecuente operar con ellas desde dispositivos móviles. Esto puede resultar problemático, puesto que las blockchains pueden llegar a ocupar cientos de gigabytes de almacenamiento en disco. Habitualmente, esto se soluciona mediante la conexión a algún proveedor que posea la blockchain completa. Sin embargo, esto contradice la naturaleza descentralizada de la tecnología blockchain. Sería interesante lograr que la seguridad de estas conexiones no dependiera de la honestidad del proveedor.

En este trabajo nos centraremos en el protocolo FlyClient, presentado por Luu, Bünz, Kiffer y Zamani en 2019. Este protocolo permite a un cliente ligero, es decir, aquel que no tiene la blockchain en almacenamiento local, verificar transacciones de criptomonedas con tan solo descargar una pequeña cantidad de bloques. Este trabajo ha despertado bastante interés puesto que ofrece una solución simple y eficiente al problema mencionado. Sin embargo, sufre de algunos problemas. En primer lugar, hay una falta general de formalización que impide determinar con precisión el funcionamiento del protocolo. Además, las garantías de seguridad del protocolo fueron mostradas de una forma muy vaga.

Hemos formalizado el protocolo como un sistema de pruebas. Además, hemos explorado las conexiones entre el protocolo FlyClient y el esquema de pruebas de trabajo secuencial de Cohen y Pietrzak. Esta conexión nos ha sido útil para especificar las garantías de seguridad del protocolo FlyClient.

Finalmente, hemos analizado las principales soluciones alternativas al problema en cuestión, esto es, los Non-Interactive Proofs of Proof-of-Work (NIPoPoWs, de aquí en adelante) y los Succint Non-Interactive Arguments of Chain Knowledge (SNACKs, de aquí en adelante), y hemos comparado sus características y garantías con las de FlyClient. Hemos concluido que FlyClient tiene ventajas notables sobre los NIPoPoWs, principalmente por ser compatible con cadenas de dificultad variable. Por otra parte, hemos notado varias similitudes entre FlyClient y los SNACKs, siendo los SNACKs ligeramente más eficientes y versatiles.

Términos clave. Blockchains, Clientes ligeros, Pruebas de Trabajo Secuencial

Contents

1. Introduction

1.1. Blockchains

A blockchain is a decentralized distributed ledger that records data across many computers in such a way the entries cannot be easily modified. Data is stored in blocks, which are sequentially added to the blockchain. Additionally, each block has a public identifier, which typically consists of a cryptographic hash, and is linked to the previous block in the chain. This way, the sequential construction of the chain can be publicly verified. In order to prevent the introduction of malicious information into the chain, every block will contain the solution to a cryptographic hash game, making the validity of each block publicly verifiable too.

Unlike traditional databases controlled by a central authority, a blockchain is maintained by a network of users, each of whom stores a copy of the structure itself. Users will expand the chain by adding blocks to the chain and sharing them with the network. Conversely, users will verify the information received from the network by checking the identifier of each block.

Since the chain is expanded in a decentralized way, it is necessary to deal with the possibility of having forks in the chain, *i.e,* the situation in which two users simultaneously add a new block to the chain and share it with the network, forcing the rest of the network to decide which chain should they accept as the new valid one. Typically, this problem will be dealt with by introducing a method of measuring the chain, making the users accept the "heaviest" chain at every moment. For example, one could measure the computational hardness of solving the cryptographic game of each block and measure any given chain by summing the difficulties of its blocks.

This model provides strong security features: In order to introduce malicious information into the chain, an attacker will need to outpace the rest of the network and obtain a chain of maximal weight such that every block has a valid identifier. Also, the decentralized network guarantees the reliability of the data stored in the blockchain, since it is stored by every user of the network.

Most well known applications of blockchain are the following:

- Cryptocurrencies: Cryptocurrencies are digital cash systems that are meant to work without a central authority. This is achieved by employing blockchain as a transaction database, which will be maintained by a network of users (typically called miners) who gather the incoming transactions into blocks and add them to the chain. In order to prevent attacks, blocks will be signed with the solution to a given cryptographic game, which is expected to be moderately hard to solve. In exchange, miners will receive financial incentives for their work.

- Smart contracts: Smart contracts are self-executed contracts whose terms are directly written in the blockchain. Once the conditions specified in a given contract are achieved, smart contracts will automatically execute and ensure the contract terms are satisfied. Blockchains ensure smart contracts are both publicly verifiable and unalterable.

- Many other areas such as supply chain management, voting systems, identity verification, and decentralized finance (DeFi).

In this work, we will focus in cryptocurrencies. As we said, blockchains will consist on sequences of blocks containing transaction data. This data will be stored in Merkle Trees. Therefore, we will distinguish between full blocks and their headers, which only contain the Merkle Tree root to which the transaction data is committed. The protocols used in this work will only use block headers to make proof sizes smaller.

Moreover, we will focus on a certain kind of cryptographic game which is called Proof of Work (PoW). This consists on manipulating blocks in order to make the block's hash to contain a certain amount of zeros at its beginning. This serves as the proof that a miner did a certain amount of computational work before adding a block to the chain. In Section 2.2 we give a more detailed explanation on how Proofs of Work are implemented.

1.2. Light client blockchain protocols

Since the introduction of Bitcoin in 2008 [Nak08], cryptocurrencies have rapidly increased in popularity. This new kind of currency is characterized for its decentralized nature, relying on the collective maintenance

of blockchains. While this makes them independent from any central authorities, it also introduces several technical problems. One of them is blockchain size: As blockchains grow linearly in time, their size is unbounded. Indeed, the blockchains of the main cryptocurrencies weight hundreds of gigabytes.

This forces us to distinguish between two types of cryptocurrency users: Full nodes and light clients. A full node possesses the full blockchain in local storage, while light clients do not. In order to operate cryptocurrencies, light clients connect to full nodes, usually in a trustful way. While this allows regular users to operate from any kind of device, it contradicts the decentralized nature of blockchain technology and allows malicious entities to pose as legitimate full nodes. It would be interesting to make this connection trustless, this is, ensuring light clients have security guarantees when interacting with a full node, regardless of the full node's honesty.

One particular instance of this problem is the verification of cryptocurrency transactions. A transaction is verified by ensuring it is stored in the blockchain, which requires to have knowledge of the blockchain itself. This problem is acknowledged in the Bitcoin white paper ([Nak08]), to which they offer a simple solution: Instead of downloading a full blockchain, a light client might only download the block headers and check they are correctly produced with respect to blockchain rules. While this helps to save some space, it still makes proof sizes linearly proportional to blockchain sizes. This makes this solution unfeasible for most light clients.

1.3. Related work

Some solutions have been proposed attempting to verify a blockchain with a sublinear (*i.e.* polylogarithmic) proof size. This implies accepting a blockchain by downloading a very small subset of its blocks, and without any previous assumption over the honesty of the full node providing the blockchain.

The first of this solutions are NIPoPoWs ([KMZ17]), proposed by Kiayias *et. al.* in 2017. This solution takes advantage on the heuristic nature of Proofs of Work, assuming the quality of PoW solutions will be uniformly distributed along the chain. However, this solution is only valid for static difficulty blockchains. Since most (if not all) cryptocurrencies use variable difficulty blockchains, this seems to put a high burden on NIPoPoW viability. Additionally, NIPoPoWs only ensure optimal proof sizes when the adversarial computational power is restricted to a third of the total computational power of the network.

The next proposed solution is the FlyClient protocol, which we will center on during this work. This protocol offers a very elegant solution to our problem: By replacing the traditional blockchains' chain graph structure for a certain kind of Merkle Trees, we obtain a stronger blockchain model that makes the detection of invalid blocks easier. This is coupled with what is called the (c, L)-Assumption, which states that sequentially solving Proofs of Work is computationally hard.

More recently, Abusalah *et.al.* presented SNACKs ([Abu+22]). This solution replaces the traditional chain graph used in most cryptocurrencies with a directed acyclic graph. In order to expand the chain, a miner will have to compute a graph label, together with a Proof of Work solution. This creates stronger block dependencies, together with the guarantee that the chain blocks were sequentially computed. SNACKs share a lot of ideas with FlyClient, indeed using the (c, L)-Assumption.

1.4. Goals of this thesis

We have exposed the general situation of the problem. As we have seen, all the proposed solutions rely on assumptions over the adversarial behavior which haven't been proven so far. Therefore, all the solutions' security guarantees are heuristic in practice. The most notable case is the (c, L)-Assumption, which is used in two out of the three presented solutions. While it seems reasonable to think an adversary won't be able to generate a long sequence of valid PoW solutions, it is not clear for which parameters c, L it could hold. As we will see in the future, the security guarantees of FlyClient and SNACKs, as well as their associated proof sizes, depend on those parameters.

As for the FlyClient protocol, the original presentation in [Bün+19] leaves some questions to be asked. The first one is the exact implementation of the FlyClient protocol, many of its features being left as a sketch. It would be desirable to formalize the protocol and concretize its features. Additionally, FlyClient's security guarantees are presented in a very simplistic way, only claiming that a cheating adversary will get rejected with overwhelming probabilities.

Also, it is left to know the relation between FlyClient and Cohen and Pietrzak's Proof of Sequential Work scheme ([CP18]). Intuitively, a PoSW is a scheme in which a prover shows to a verifier he did a certain amount of sequential computations. In particular, the mentioned protocol uses a form of graphs that are very similar to the blockchain model used in FlyClient. This is mentioned in [Bün+19], but is left unresolved. Studying this relation could lead us to replicate the posw's security guarantees in FlyClient in order to obtain a more concrete security result.

Finally, it would be desirable to compare the mentioned solutions to check which one is the most suitable one. This involves comparing their security guarantees, efficiency and dependencies on additional assumptions.

Our results. As we said before, this work is centered on the FlyClient protocol. The main results we obtained are:

1. We have extracted the FlyClient blockchain model and proven it contains the Cohen and Pietrzak graph structure.

2. We have formalized the FlyClient protocol, including a version supporting variable difficulty blockchains and a non-interactive one. We have also implemented all the algorithms that are required to run the protocol.

3. We have taken advantage on the connection to Cohen and Pietrzak's Proof of Sequential Work scheme to elaborate a security theorem (Theorem 5) that links adversarial chances of success to their ability to do sequential calls to a hash function.

4. We have compared FlyClient with NIPoPoWs and SNACKs. We have concluded that FlyClient has several advantages with respect to NIPoPoWs. On the other hand, while FlyClient and SNACKs are similar in many ways, SNACKs seem to be more efficient and versatile.

In Section 2, we introduce all the concepts we need to define the FlyClient protocol. This includes Merkle Mountain Ranges (MMR), the graph structure used in [Bün+19], Merkle Mountain Graphs (MMG), the graph structure we obtained from combining MMR with CP-graphs and the FlyClient blockchain model. This section includes some theoretical results, mainly proving that the FlyClient blockchain has the same graph structure as CP-graphs. Finally, we have formalized the (c, L)-Assumption inspired by the work done in [Abu+22], overcoming some of the shortcomings of the original definition.

In Section 3, we formalize the FlyClient protocol. Section 3.1 introduces the sampling distribution used in [Bün+19]. Section 3.2 describes the non-interactive FlyClient protocol for static difficulty blockchains, the simpler version of the protocol. Section 3.3 gives a proof on the protocol's security guarantees with respect to adversarial ability to do sequential work. Section 3.4 calculates the proof size of the FlyClient protocol. Section 3.5 introduces a more complex version of the protocol that supports chains of variable difficulty. Finally, Section 3.6 uses the Fiat-Shamir heuristic ([FS86]) to make the protocol non-interactive.

In section 4, we provide a short review of the main alternative solutions to the transaction verification problem, NIPoPoW and SNACKs. We also compare the guarantees offered by each protocol.

Finally, Section 5 displays some unresolved problems that could be interesting for future work.

2. Basic concepts and definitions

2.1. Basic notation

We denote by $\{0,1\}^{\leq n}$ the set of all binary strings of length at most n, the empty string being denoted by ε. Concatenation of bitstrings is denoted by $||$. For $x \in \{0,1\}^*$, $x[i]$ denotes its i-th bit, $x[i \dots j]$ denotes $x[i]||\dots||x[j]$ and $|x|$ denotes the bitlength of x.

Given a sequence x of n elements, we denote the i-th element by $x[i]$, where $x[0]$ denotes the first element and $x[n-1]$, the last one.

A Direct Acyclic Graph (DAG, from now on) is a graph which edges have a defined direction and that doesn't contain any cycles.

An algorithm is said to be probabilistic polynomial time if it runs in polynomial time and is allowed to introduce randomness in its computations.

A function $f : \mathbb{N} \to \mathbb{R}$ is said to be negligible if, for every constant $c \geq 0$, there exists an integer k_c such that $f(x) \leq \frac{1}{2^c}$ for $x \geq k_c$. In such case, we say $f \in \mathsf{negl}(\cdot)$.

2.2. Bitcoin

Bitcoin is a blockchain-based decentralized cryptocurrency. It was the first of its kind to be introduced at the market and it is still the most well known one. It was introduced in 2008 by the pseudonymous Satoshi Nakamoto when the white paper called "Bitcoin: A Peer-to-Peer Electronic Cash System" [Nak08] was released in public. In it, a sketch of the internal behavior is drawn, together with a glimpse of its supposed security properties. However, in order to prove the inability of an attacker to catch up with the honest chain, it assumes the honest miners will act in a coordinated way, which doesn't seem to match the performance of a decentralized network.

A more in detail analysis was provided in [GKL15], where the protocol for the static difficulty model was formalized. We will provide a short review of it here:

Bitcoin blockchain. Let $G(\cdot), H(\cdot)$ be cryptographic hash functions with output in $\{0,1\}^w$. We define a block as a triple $B = \langle s, x, ctr \rangle$, where $s \in \{0,1\}^w, x \in \{0,1\}^*, ctr \in \mathbb{N}$ such that

$$(H(ctr, G(s,x)) < T) \wedge (ctr < q)$$

where $T \in \mathbb{N}$ is called the block's difficulty level and q is the allowed size for ctr. In practice, q will represent the allowed number of queries to H of any party involved in the protocol. For the sake of simplicity, we will assume T, q stay constant for the whole protocol. However, more sophisticated versions of the protocol have been instantiated so that this limitation is overcome.

We will say that a block $B = \langle s, x, ctr \rangle$ is valid with respect to parameters T, q if the former predicate is satisfied.

A blockchain $\mathcal{C}$ is a sequence of blocks. The rightmost block is called the head of the chain, denoted $\mathsf{head}(\mathcal{C})$. The empty chain ε will satisfy $\mathsf{head}(\varepsilon) = \varepsilon$ by convention.

A chain $\mathcal{C}$ with head $B' = \langle s', x', ctr' \rangle$ can be extended with a valid block $B = \langle s, x, ctr \rangle$ such that $s = H(ctr', G(s', x'))$. The empty chain ε can be extended with any block by default. In any case we get an extended chain $\mathcal{C}' = \mathcal{C}B$ such that $\mathsf{head}(\mathcal{C}') = B$. The length of a chain $\mathcal{C}$, $\mathsf{len}(\mathcal{C})$ is its number of blocks.

Consider a chain $\mathcal{C}$ of length m. We denote by $\mathcal{C}^{\lceil k}$ the result of removing the last k blocks from the chain. If $k \geq m$, then $\mathcal{C}^{\lceil k} = \varepsilon$. If a chain $\mathcal{C}_1$ is a prefix of $\mathcal{C}_2$ we denote it by $\mathcal{C}_1 \preceq \mathcal{C}_2$.

We note that we used two different hash functions G, H for the processing of transactions and blocks, respectively. This is not the case in the practical implementation of bitcoin, where both functions are instantiated as SHA-256.

Backbone protocol. The backbone protocol is modeled as a system of Interactive Turing Machines (ITM). The protocol is driven by an environment program $\mathcal{Z}$ that may spawn many instances running the protocol itself. Those programs can be seen as ITM that have communication, input and output tapes. The spawning of those instances as well as the interaction between them is managed by a control program C, which is an ITM itself.

We will consider a fixed set of n parties $P_1, \ldots P_n$, each of which is modeled as an ITM. Additionally, an adversary $\mathcal{A}$ (modeled as an ITM too), will be able to corrupt and take control of at most t of the parties.

The protocol will work on the basis of a round system. At the beginning of each round, the adversary $\mathcal{A}$ will be spawned, being able to corrupt at most t parties of its choice. Next, the parties will be spawned in order. Every time a corrupt party is spawned, the adversary will be spawned instead.

Every time a party P_i is instantiated, it will have access to the following two functionalities:

- Proof of Work: Every party will be allowed to make q queries to the hash function H in order to expand a blockchain. The q limit is imposed to reflect the limited computational capability of real life miners. An honest party might as well want query H to verify the work done by other parties involved in the protocol. No limitation will be imposed in this case. Corrupted parties will not be able to do verification queries.

- Diffusion: Every party has an input tape, which it can check at any moment. Conversely, it will be able to diffuse messages to the rest of the participants. It is guaranteed that every party has received the

message by the end of the round it was sent in. Adversarial-controlled parties have enhanced diffusion powers, being able to receive any message at the moment and manipulate the input tape of any party.

The enhanced properties of adversaries are meant to represent the possibility of diffusion-based attacks, in which an adversary focuses on fast message diffusion rather than fast block computation. This could lead to de-synchronize the honest parties, slowing down their performance. Additionally, we are assuming that the adversarial parties will work on a coordinated way, this puts us in a worst case scenario that will ensure the sturdiness of whichever security properties we might infer from the protocol.

Conversely, we might note we barred the adversary from doing any verification queries. This is not a real burden since we are assuming the adversary to be working in a centralized manner. On the other hand, this will prevent the possibility of an adversary saturating the network with malicious messages obtained from verification queries.

It must be noted that the adversary can corrupt parties at will every round, this simulates the nature of a decentralized network: parties will not be able to determine which messages were shared by an honest party and which weren't.

The previously mentioned constants t, n are hard-coded into the control program C. Table 1 provides a list of the different variables the backbone protocol is instantiated with.

Backbone protocol parameters
w: Security parameter. Length of the hash function.
λ: Typical execution parameter.
n: Number of parties in the protocol, t of which are controlled by the adversary.
T: Difficulty target for the Proof of Work.
δ: Advantage of honest parties, $t/(n-t) \leq \delta$.
f: probability for at least one honest party to succeed in finding a POW in a round.
ϵ: quality of concentration of random variables in typical executions.
k: number of blocks for the common prefix property.
ℓ: number of blocks for the chain quality property.
μ: chain quality parameter.
s: number of rounds for the chain growth property.
γ: chain growth parameter.
m: epoch size (variable difficulty only).
τ: dampening filter for difficulty target recalculation.

Table 1: The parameters in the backbone protocol analysis. $n, t, s, \ell, T, k, w, m$ are positive integers. $f, \epsilon, \delta, \mu, \gamma, \lambda, \tau$ are positive reals, where $f, \epsilon, \delta, \mu \in (0, 1)$ and $\tau > 1$.

Protocol properties. We want to ensure the protocol is secure, *i.e.* honest parties will eventually agree on an honestly produced chain. We will model this through two properties: The common prefix property and the chain quality property.

Definition 1 (Common prefix property)**.** *The common prefix property with parameter $k \in \mathbb{N}$ states that, for any two honest parties P_1, P_2 adopting the chains $\mathcal{C}_1, \mathcal{C}_2$ in rounds $r_1 \leq r_2$, it holds $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$.*

Definition 2 (Chain quality property)**.** *The chain quality property with parameters $\mu \in (0, 1), \ell \in \mathbb{N}$ states that for any honest party P holding a chain $\mathcal{C}$, it holds that for any ℓ consecutive blocks the ratio of valid blocks is at least μ.*

Additionally, the chain growth property addresses the chain growth rate issue:

Definition 3 (Chain growth property)**.** *The chain growth property with parameters $\gamma \in (0, 1), s \in \mathbb{N}$ states that for any honest party holding a chain $\mathcal{C}$, it holds that after any s consecutive rounds it adopts a chain that is at least $\gamma \cdot s$ blocks longer.*

It rests to infer suitable parameters for the three properties. For that purpose, we should first specify the conditions under which an execution of the backbone protocol is "appropriate". The first one relates to the rate of corrupted parties:

Assumption 1 (Honest majority assumption)**.** *A number t out of n parties is corrupted such that $t/(n-t) \leq \delta$, where $3f + 3\epsilon < f$.*

The next "inappropriate" event we will be treating is the possibility of a blockchain not being sequentially computed.

Definition 4. *An insertion occurs when, given a chain $\mathcal{C}$ with two consecutive blocks B and B', a block B^* is created such that B, B^*, B' form a valid chain. A copy occurs when some block exists in two different positions of a chain. A prediction occurs when a block extends some block that was created in a later round.*

Now we will relate to the statistical behavior of Proofs of Work. We will define the binary random variables X_i, Y_i, Z_i as follows: If at round i an honest party obtains a valid Proof of Work, then $X_i = 1$, otherwise $X_i = 0$. If at round i exactly one honest party obtained a valid PoW, then $Y_i = 1$, otherwise $Y_i = 0$. If at round i a malicious party obtained a valid PoW, then $Z_i = 1$, otherwise $Z_i = 0$.

Definition 5 (Typical execution)**.** *An execution is (ϵ, λ)-typical, for $\epsilon \in (0, 1), \lambda \geq 2/f$ if, for any set S of at least λ it holds:*

1. $(1 - \epsilon)\mathbb{E}[X(S)] < X(S) < (1 + \epsilon)\mathbb{E}[X(S)]$.
2. $(1 - \epsilon)\mathbb{E}[Y(S)] < Y(S)$.
3. $Z(S) < \mathbb{E}[Z(S)] + \epsilon\mathbb{E}[X(S)]$.
4. *No insertions, no copies and no predictions occurred.*

A typical execution is one in which the chain was computed sequentially and the queries to the hash function behaved close to the expected. It can be proven that an execution of the backbone protocol will be typical with overwhelming probability:

Lemma 1. *An execution is typical with probability $1 - e^{-\Omega(\epsilon^2 \lambda f + w - \log(L))}$.*

Now we can infer parameters for the given properties:

Theorem 1. *In a typical execution the common prefix property holds with parameter $k \geq 2\lambda f$.*

Theorem 2. *In a typical execution the chain quality property holds with parameters*

$$l \geq 2\lambda f \text{ and } \mu = 1 - (1 + \frac{\delta}{2})\frac{t}{n-t} - \frac{\epsilon}{1-\epsilon} > 1 - (1 + \frac{\delta}{2})\frac{t}{n-t} - \frac{\delta}{2}.$$

Theorem 3. *In a typical execution the chain growth property holds with parameters $\gamma = (1-\epsilon)f$ and $s \geq \lambda$.*

Variable Difficulty in Bitcoin. The backbone protocol was extended to support chains of variable difficulty in [GKL16]. We provide here an explanation of the difficulty transition rules.

First place, we will have to expand the block header structure to support timestamps. A block in now a quadruple $B = \langle t, s, x, ctr \rangle$, where $t, ctr \in \mathbb{N}$ and $s, x \in \{0, 1\}^w$. t is meant to be a block timestamp. A block is valid with respect to a difficulty target T and a nonce size q if

$$(H(ctr, G(t, s, x)) < T) \wedge (ctr < q)$$

Each block in a chain will be weighted with the inverse of its associated difficulty target, making it proportionally heavy with respect to the "hardness" of its associated Proof of Work. The cumulative weight of a chain will be measured by the sum of the weights of the blocks. Unlike the static difficulty model, honest nodes participating in the protocol will seek to agree on the heaviest chain, instead of the longest one.

Bitcoin organizes the blocks in its chain in sets of a fixed length m, known as *epochs*. Each time the blocks in a certain epoch i have been mined, the difficulty target for the next epoch is calculated. It is

desired for epochs to be mined at a fixed rate of m/f rounds, where f is the expected probability for a block to be successfully mined in a single round.

Assuming that an epoch i was mined with a difficulty target T in a number of rounds Δ (which can be calculated by summing the timestamps of the blocks in that epoch), the number of miners in epoch i is estimated as

$$n(T, \Delta) = \frac{2^w m}{qT\Delta}$$

where w is the digest size of a hash function and q is the number of hash queries allowed to each miner per round.

As the number n of miners varies, so will do the difficulty targets T of the blocks. It will be done in a way such that the probability for n miners to mine a block in a single round with a difficulty target T, represented by

$$f(T, n) = \frac{qTn}{2^w}$$

remains close to f. The genesis block of every chain will contain an initial number of miners n_0 and an initial difficulty target T_0 such that $f(T_0, n_0) = f$.

We will adjust the difficulty target proportionally to the variation of the hashing power of the estimated number of players. However, this could allow an adversary to take advantage of an excessive difficulty variation in order to either mine blocks faster or drastically increase the cumulative difficulty of its chain. In order to avoid this, we will introduce a dampening filter τ which will represent the maximum rate of difficulty variation we will allow. If the difficulty target was going to variate by a bigger rate, we would modify it by a rate of τ or $1/\tau$ instead.

To accomplish this, we calculate the difficulty target for epoch $i+1$ as follows:

$$D(T, \Delta) = \begin{cases} \frac{1}{\tau}T & \text{if } \frac{n_0}{n(T,\Delta)}T_0 < \frac{1}{\tau}T \\ \tau T & \text{if } \frac{n_0}{n(T,\Delta)}T_0 > \tau T \\ \frac{n_0}{n(T,\Delta)}T_0 & \text{otherwise} \end{cases}$$

Observation 1. *Bitcoin blockchain is expected to be expanded with a new block every 10 minutes on average. In order to do so, the parameters are instantiated as* $m = 2016, f = 0.03, \tau = 4$.

2.3. Hash function basics

As we said before, blockchains ensure security by employing cryptographic hash games. In order to prove our protocols' security, we will require every hash function to be collision resistant and sequential.

Definition 6 (Collision Resistant Hash Function)**.** *A hash function* $H : \{0,1\}^* \to \{0,1\}^w$ *is said to be collision resistant if, for any adversary* $\mathcal{A}$ *that computes two inputs* $x, y \in \{0,1\}^*$, $H(x)$ *will equal* $H(y)$ *with negligible probability in* w.

We will introduce the notion of H-Sequences. Intuitively, an H-Sequence captures the notion of a set of data that is computed by making sequential queries to a given hash function H. In the future, we will show that the opening of a commitment in a Merkle Tree implies computing an H-Sequence. Furthermore, we will say that a hash function H is Sequential if every adversary trying to compute an H-Sequence of length s using less than s sequential queries to H will fail with overwhelming probability.

Definition 7 (H-sequence [CP18])**.** *Given a hash function* $H : \{0,1\}^* \to \{0,1\}^w$, *an* H*-sequence of length* s *is a sequence* $x_0, \ldots, x_s \in \{0,1\}^*$ *such that,* $\forall i, 0 \leq i < s$, $H(x_i)$ *is contained in* x_{i+1} *as a continuous substring, i.e.,* $x_{i+1} = a||H(x_i)||b$ *for some* $a, b \in \{0,1\}^*$.

Definition 8 (Sequential Hash function [CP18])**.** *A hash function* $H : \{0,1\}^* \to \{0,1\}^w$ *is said to be sequential if, for any adversary that tries to compute an* H*-sequence of length* s *doing at most* $s-1$ *sequential queries to* H, *the probability of success is negligible in* w.

Unless it is otherwise specified, hash functions will be instantiated in the Random Oracle Model (ROM). This means that, given a hash function $H : \{0,1\}^* \to \{0,1\}^w$, H will behave as a black box that, on any input $x \in \{0,1\}^*$, outputs an uniformly distributed random element $H(x) \in \{0,1\}^w$. We will use the work done in [CP18] to prove that random oracles have the two mentioned properties.

Lemma 2 (Random Oracles are Collision Resistant [CP18])**.** *Consider any adversary $\mathcal{A}$ that is given access to a random oracle $H : \{0,1\}^* \to \{0,1\}^w$. If $\mathcal{A}$ makes at most q queries, the probability it will make two colliding queries is at most $q^2/2^{w+1}$.*

Lemma 3 (Random Oracles are Sequential [CP18])**.** *Consider any adversary $\mathcal{A}$ that is given access to a random oracle $H : \{0,1\}^* \to \{0,1\}^w$ that it can query for at most $s-1$ rounds, where in each round it can make arbitrary many parallel queries. If $\mathcal{A}$ makes at most q queries of total length Q bits, then the probability that it outputs an H-sequence $x_0, \ldots, x_s$ is at most*

$$q \cdot \frac{Q + \sum_{i=0}^{s} |x_i|}{2^w}$$

2.4. Proofs of Sequential Work

Proofs of Sequential Work (PoSW, from now on) are protocols in which a prover will convince a verifier it did an amount of sequential computational work [MMV13]. PoSWs have many applications to blockchain design. For example, the cryptocurrency Chia [CP19] uses a restrictive version of PoSW, called Verifiable Delay Functions [Bon+18], which serve as part of a consensus algorithm that is more resource-efficient that the traditional Proofs of Work. Also, the blockchain light-client protocol SNACK, which we will further discuss in future sections, employs a blockchain model inherently based on PoSWs.

We provide a unifying PoSW primitive, which is in turn a non-incremental variation of the one provided in [AC23]. Our protocol will be parametrized by a time parameter N, representing the amount of sequential work the prover is expected to do, and instantiated by a family of weighted DAGs $\Gamma := (\Gamma_N = (G_N, \Omega_N))_{N \in \mathbb{N}}$ and a family of oracles $H := (H_v)_{v \in G_N}$, with $H_v : \{0,1\}^* \to \{0,1\}^w$ (typically instantiated a a random oracle hash function). Recall that $\Omega_N : G_N : [0,1]$ is a weight distribution such that $\Omega_N(G_N) = 1$. Given a sequence $Q = (Q_0, \ldots, Q_n)$ of parallel queries to H, we will define its sequential weight as $\Omega_{seq}(Q) = \sum_{i=0}^{n} \max\{\Omega_N(v) : Q_i \text{ contains a call to } H_v\}$.

To start the protocol, the verifier will compute a statement χ, which will send to the prover. Then, the verifier will use χ and an oracle H to compute the labeling of a DAG Γ_N and send a commitment to it to the verifier. Finally, the verifier will send a series of challenges to the prover to check the received commitment is legit. If the prover opens the challenges correctly, the verifier will output 1. Otherwise, it will output 0.

On top of being secure, we will be requiring our PoSW protocols to be succinct. This is, all the proofs generated by them shall be polylogarithmic in size with respect to the security parameters.

Definition 9 (PoSW [AC23])**.** *Let $\Gamma = (\Gamma_N = (G_N, \Omega_N))_{N \in \mathbb{N}}$ be a family of weighted DAGs such that for all N, G_N has an unique sink N. A tuple of oracle aided PPT algorithms $(P^H, V^H := (V_0^H, V_1^H))$ for an oracle $H : \{0,1\}^* \to \{0,1\}^w$ is a Proof of Sequential Work with respect to H if the following properties hold:*

Completeness: For every $w, N \in \mathbb{N}$, every $(\chi, N) \leftarrow V_0^H(1^w, N)$ and $\pi \leftarrow P^H(1^w, 1^N, \chi)$ it holds that

$$\Pr[V_1^H(\chi, N, \pi) = 1] \geq 1 - \mathsf{negl}(w)$$

(α, ϵ)-Soundness: For every $w, N \in \mathbb{N}$ and every PPT adversary $\tilde{P}^H$ which makes a sequence Q of parallel queries to H of sequential weight $\Omega_{seq}(Q) < \alpha$:

$$\Pr\left[\begin{array}{c} (\chi, N) \leftarrow V_0^H(1^w, N) \\ \pi \leftarrow \tilde{P}^H(1^w, 1^N, \chi) \end{array} \ : \ V_1^H(\chi, N, \pi) = 1 \right] \leq \mathsf{negl}(w) \tag{1}$$

Succinctness: For every $w, N \in \mathbb{N}$ and every honestly generated proof for parameter N, we have $|\pi| \leq \mathsf{poly}(w, \log(N))$ and V^H runs in time $\mathsf{poly}(w, \log(N))$.

We will instantiate this primitive with the PoSW depicted in [CP18]. This protocol was developed by Cohen and Pietrzak in 2018. As we will see in the future, there are some similarities between the Cohen and Pietrzak (CP, from now on) and the FlyClient protocol. Mainly, the DAG model that instantiates the CP-PoSW is very simillar to the FlyClient blockchain model.

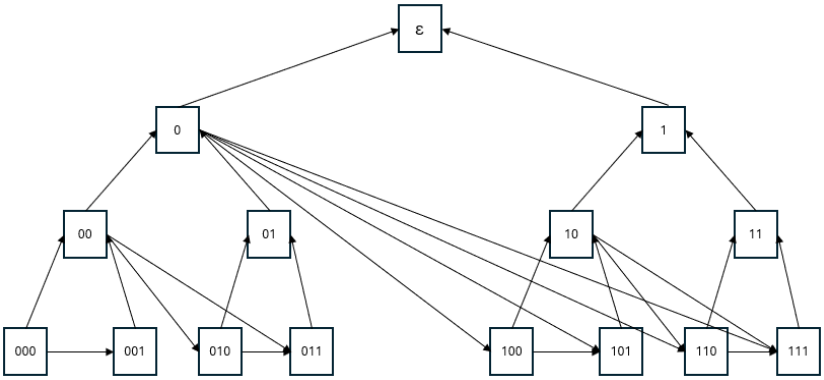

Figure 1: Illustration of the CP-Graph G_3^{CP}

Definition 10 (CP-Graphs)**.** *For $n \in \mathbb{N}$, let $N = 2^{n+1} - 1$ and $M_n = (V, E')$ a complete binary graph of depth n. We identify the N nodes with the set $V = \{0,1\}^{\leq n}$ of the binary strings of length at most n and E' with the set of directed edges that go from the bottom to the top of the graph*

$$E' = \{(x||b, x) : b \in \{0,1\}, x \in \{0,1\}^{<n}\}$$

We define the DAG $G_n^{CP} := (V, E)$, with $E = E' \cup E''$, where E' is as defined before and E'' is the set of directed edges that connects every DAG leaf with all the left children of the elements in the path going from the leaf to the DAG's root, namely:

$$E'' = \{(v, u) : u \in \{0,1\}^n, u = a||1||a', v = a||0\}$$

Given a *DAG* $G_N^{\mathsf{CP}} = (V, E)$ on N nodes, we will equip it with the trivial weight distribution $\Omega(v) = 1/N$. From now on, we will omit it for simplicity. Figure 1 illustrates a CP-Graph G_3^{CP} on $8 = 2^3$ leaves and $15 = 2^{3+1} - 1$ nodes.

Figure 2 displays the CP-PoSW protocol. It rests to specify the label, open and ver algorithms. We describe them here:

- $(\phi, aux) \leftarrow \mathsf{PoSW.label}(\chi, N)$. Computes the labeling of a CP-Graph $G_N = (V, E)$. For every $i \in V$ it computes its associated label as $\ell_i = H(\chi, i, p_1, \ldots, p_d)$ where $(p_1, \ldots, p_d)$ are the labels of i's parents in G_N. ϕ is the label of G_N's root, while aux are whichever auxiliary labels the prover wants to store. A prover can either store all computed labels in order to maximize computational efficiency or keep only the labels of some of the uppermost nodes in the graph, sacrificing computational efficiency in favor of storage efficiency.

- $o_i \leftarrow \mathsf{PoSW.open}(\chi, N, aux, \tau_i)$. Given a challenge τ_i on any of G_N's leaves, the prover generates a Merkle Tree opening of τ_i with respect to G_N. If the whole opening is not contained in aux, the prover will need to recompute some labels.

- $b_i \leftarrow \mathsf{PoSW.ver}(\chi, N, \phi, \tau_i, o_i)$. Given a challenge τ_i and an opening o_i for it, the verifier checks the opening and outputs a boolean b_i.

Finally, we can decribe the security guarantees of the CP-PoSW protocol.

Theorem 4. *The protocol described in Figure 2 instantiated by security parameters t, w is an (α, ϵ)-sound Proof of Sequential Work, with*

$$\epsilon = (1 - \alpha)^t + \frac{2 \cdot n \cdot w \cdot q^2}{2^w}$$

Verifier V = (V_0,V_1,V_2)
Stage V_0: On input N:
1. get $\chi \xleftarrow{\$} \{0,1\}^w$
2. send (χ, N) to V

Stage V_1: On input ϕ:
1. for $i \in \{0,\dots,t\}$ do $\tau_i \xleftarrow{\$} \{0,1\}^n$
2. send $\tau := (\tau_i)_{i=0}^t$ to V_1

Stage V_2: On input $o := (o_i)_{i=0}^t$:
1. for $i \in \{0,\dots,t\}$ do
   $b_i \leftarrow \mathsf{PoSW.ver}(\chi, N, \phi, \tau_i, o_i)$
2. output $\bigwedge_{i=0}^t b_i$

Prover P = (P_0,P_1):
Stage P_0: On input (χ, N):
1. $(\phi, aux) \leftarrow \mathsf{PoSW.label}(\chi, N)$
2. send ϕ to V_1

Stage P_1: On input $\tau := (\tau_i)_{i=0}^t$:
1. for $i \in \{0,\dots,t\}$ do $o_i \leftarrow \mathsf{PoSW.open}(\chi, N, aux, \tau_i)$
2. send $o := (o_i)_{i=0}^t$ to V_2

Figure 2: Description of the CP-PoSW protocol. The protocol is instantiated by challenge size $t \in \mathbb{N}$ and hash function size $w \in \mathbb{N}$.

2.5. Merkle Mountain Ranges

In order to achieve stronger block commitment, the FlyClient protocol introduces a new blockchain model based on binary hash trees. The particular type of trees that are going to be used is called Merkle Mountain Ranges.

Definition 11 (Merkle Mountain Range)**.** *A Merkle Mountain Range (MMR) on n leaves, M_n, is a binary hash tree with n leaves and the following properties:*

1. *M_n has depth $\lceil \log_2(n) \rceil$.*
2. *If $n > 1$, let $n = 2^i + j$, with $i = \lfloor \log_2(n-1) \rfloor$. Then:*
   - *M_n's left child is an MMR of 2^i leaves.*
   - *M_n's right child is an MMR of j leaves.*

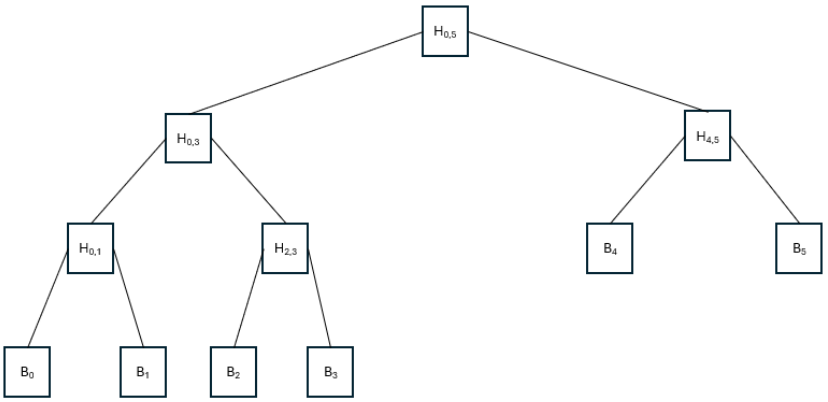

Figure 3: Illustration of an MMR of 6 leaves.

We will denote the left and right children of an MMR M_n by $\mathsf{lc}(M_n)$ and $\mathsf{rc}(M_n)$, respectively. Additionally, $\mathsf{root}(M_n)$ will denote M_n's root.

Definition 12. *Let M, H_i be an MMR and one of its leaves, respectively. We denote by $\mathsf{path}(H_i, M)$ the set of nodes in the path from H_i to M's root. Additionally, we denote the set of leaves of M by $\mathsf{leaves}(M)$.*

In order to simplify notation, we will denote the nodes in an MMR in the following way:

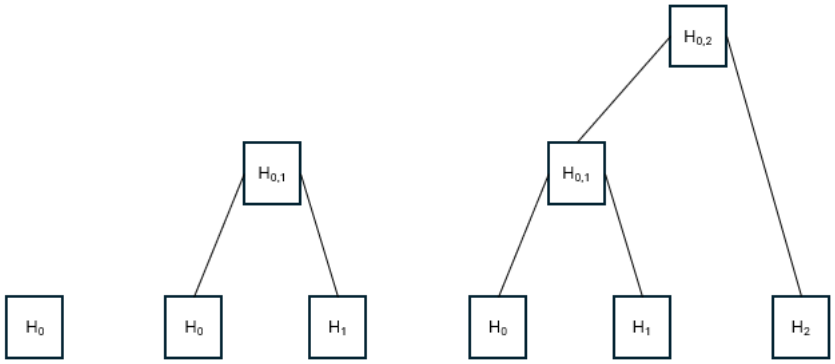

Figure 4: Application of the AppendLeaf algorithm to add two leaves H_1, H_2 to a single-leaf MMR (left). As a result, we successively obtain a two-leaf MMR (center) and a three-leaf MMR (right).

- We will orderly denote MMR leaves as $\{H_i\}_{i\in\mathbb{N}}$. For the moment we will assume they contain generic hash digests.
- If an internal node is the parent of two leaves H_i, H_{i+1}, with i being even, we will denote it as $H_{i,i+1}$. Recall that such a node will contain value $H(H_i, H_{i+1})$.
- If an internal node is the parent of two nodes $H_{i,j}, H_{j+1,k}$, we will denote it as $H_{i,k}$. Similarly to the previous case, such a node will contain value $H(H_{i,j}, H_{j+1,k})$.

We observe this notation suggests the equality

$$H(H(H_{i,j}, H_{j+1,k}), H_{k+1,l}) = H_{i,l} = H(H_{i,j}, H(H_{j+1,k}, H_{k+1,l}))$$

which doesn't hold in practice. However, this won't be a problem since the nodes we hash into a common parent are determined by the MMR structure and not by the notation itself.

Figure 3 displays an MMR M_6 of 6 leaves with its associated labelling. From now on, we will identify every node in an MMR with its associated label. Additionally, in order to state that a given node N belongs to an MMR M_n, we will say $N \in M_n$. We can see that $\mathsf{path}(H_2, M_6) = \{H_{2,3}, H_{0,3}, H_{0,5}\}$ and $\mathsf{leaves}(M_6) = \{H_i\}_{i=0}^{5}$.

Observation 2. *In the notation we are using for MMR nodes, a node denoted by $H_{i,j}$ will represent the root of $\mathsf{MMR}(H_i, \ldots, H_j)$.*

As we said before, the blocks in the new blockchain model will be stored in the MMR leaves. In order to allow full node miners to expand their blockchains, we will introduce the algorithm AppendLeaf that adds a new leaf to an existing MMR. Given an MMR M_n of n leaves and a leaf x, AppendLeaf will work in the following way:

- If n is a power of 2, then a new MMR M_{n+1} will be created, such that $\mathsf{lc}(M_{n+1}) = M_n$, $\mathsf{rc}(M_{n+1}) = x$ and $\mathsf{root}(M_{n+1}) = H(\mathsf{root}(M_n), x)$.
- If n is not a power of 2, then we will recursively call AppendLeaf on $\mathsf{rc}(M_n)$ and recalculate $\mathsf{root}(M_n)$.

The AppendLeaf algorithm is explained in Figure 5. Given a set of leaves $H_1, \ldots, H_n$, we will denote by $\mathsf{MMR}(H_1, \ldots, H_n)$ the MMR obtained from recursively applying AppendLeaf on $H_1, \ldots, H_n$. Figure 4 shows the application an the AppendLeaf algorithm, adding two leaves to a single leaf MMR.

Similarly to the size of the MMR, we want to determine the computational cost of calling the AppendLeaf algorithm on a given MMR.

Lemma 4. *Applying the AppendLeaf algorithm on an MMR M_n on n leaves will have computational cost $O(\log(n))$.*

Algorithm $\mathsf{AppendLeaf}(M_n, x)$

$//M_n$ is an MMR on n leaves, x is a leaf. Returns an MMR on $n+1$ leaves

if $n = 2^k$ for some $k \geq 0$:	else:
1. Create new MMR M_{n+1}	1. Get rc'← AppendLeaf($\mathsf{rc}(M_n)$,x)
2. Set $\mathsf{lc}(M_{n+1})$ as M_n	2. Set $\mathsf{rc}(M_n)$ as rc'
3. Set $\mathsf{rc}(M_{n+1})$ as x	3. Set $\mathsf{root}(M_n)$ as $H(\mathsf{root}(\mathsf{lc}(M_n)), \mathsf{root}(rc'))$
4. Set $\mathsf{root}(M_{n+1})$ as $H(\mathsf{root}(M_n), x)$	4. return M_n
5. return M_{n+1}	

Figure 5: Description of the AppendLeaf algorithm

Proof. Given an MMR M_n on n leaves, we distinguish the two following cases:

- If $n = 2^i$, then the algorithm takes a fixed number of operations, so it will have computational cost $O(1)$.
- If $n = 2^i + j$ such that $i = \lfloor \log_2(n-1) \rfloor$ and $j > 0$, then AppendLeaf will be recursively called on $\mathsf{rc}(M_n)$, which is by definition an MMR with j leaves. This step will be repeated until we reach some subtree with 2^k leaves, for some $k \geq 0$. The worst case scenario is that we have to wait until reaching a subtree with 1 leaf.

  Since $i = \lfloor \log_2(n-1) \rfloor$, we have that $2^i < n = 2^i + j < 2^{i+1}$, so it must hold that $j < \frac{n}{2}$. Therefore, we will have to perform at most $\lceil \log_2(n) \rceil$ recursive calls of AppendLeaf.

□

We want to check how much space will it take for a full node miner to store an MMR. Fortunately, it is easy to prove the number of nodes in an MMR is linearly related to its number of leaves:

Lemma 5. *Every MMR M_n on n leaves has $2n-1$ nodes.*

Proof. We prove the result by induction on n.

- If $n = 1$, then M_n consists of a single node. It trivially holds that $2n - 1 = 1$.
- If $n > 1$, then M_n is the result of inserting a new block in an MMR M_{n-1} of $n-1$ leaves. We distinguish two cases:
  - If $n - 1 = 2^i$ for some $i \in \mathbb{N}$, then M_n contains M_{n-1} on its left child and the newly inserted block on its right child. As a consequence, M_n has two more nodes than M_{n-1}, and it holds that $2 * 2^i - 1 + 2 = 2 * (2^i + 1) - 1$.
  - If $n - 1 = 2^i + j$ such that $i = \lfloor \log_2(n-2) \rfloor$ and $j > 0$, then the new block is recursively inserted in the root's right child, which is an MMR with j leaves itself. It holds that $j < n - 1$, so, by induction hypothesis, we get that the insertion will result in an MMR with $j + 1$ leaves and $2(j+1) - 1 = 2j - 1 + 2$ nodes. We conclude that M_n will have $2(n-1) - 1 + 2 = 2n - 1$ nodes.

□

Once a Prover $\mathcal{P}$ commits an MMR M_n in the FlyClient protocol by sending its root, any Verifier $\mathcal{V}$ that stored M_n's root will be able to check that any leaf H sent by $\mathcal{P}$ is indeed contained in M_n. This will be done through Merkle Proofs, which let $\mathcal{V}$ compute the path from H to M_n's root.

Definition 13 (Merkle Proof)**.** *Given an MMR M_n and a leaf $H_i \in M_n$, a Merkle Proof for H_i is a sequence of MMR nodes that contains every children of the nodes in the path from H_i to M_n's root, excepting those that are contained in the path itself.*

Observation 3. *Given an MMR M_n and a leaf $H_i \in M_n$, the Merkle Proof of H_i in M_n will have length equal to the depth of H in M_n.*

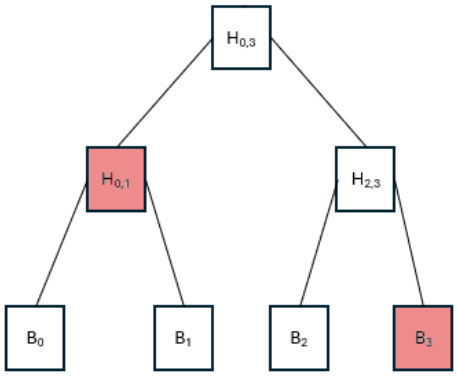

Figure 6: Illustration of an MMR of 4 leaves. The Merkle Commitment of H_2 is highlited in red.

Proof. Definition 13 implies that the Merkle Proof of H_i in M_n will have the same length as the path from H_i to $\mathsf{root}(M_n)$. This length equals the depth of H_i in M_n.

□

We will denote the Merkle Proof of a leaf H_i in an MMR M_n as $\mathsf{mp}(H, M_n)$. Intuitively, a Merkle Proof contains the minimal set of nodes that a verifier needs to compute M_n's root from H_i. Figure 6 illustrates an MMR M_4 of 4 blocks. It is easy to see that $\mathsf{mp}(H_2, M_4) = \{H_3, H_{0,1}\}$.

In order to open a Merkle Proof, the verifier will have to orderly hash its associated block with the elements in the proof itself in order to compute the elements in the path from the given leaf to the MMR's root. Returning to the example in Figure 6, a verifier will first compute $H_{2,3} = H(B_2, B_3)$, then $H_{0,3} = H(H_{0,1}, H_{2,3})$. We deduce from this that $B_2||B_3, H_{0,1}||H_{2,3}, H_{0,3}$ is an H-sequence as exposed in Definition 7.

It is interesting to check whether an adversary could create a valid Merkle Proof for a block not included in M_n. The following Lemma will prove it is sufficient to assume a sequential hash function to make this possibility negligibly probable.

Lemma 6. *Given an MMR M_n that was computed using a hash function H, a leaf $H_i \in M_n$ and a Merkle Proof of H_i in M_n, $\mathsf{mp}(H_i, M_n)$. If the Merkle Proof is valid, then some of the following possibilities hold:*

- *$\mathsf{path}(H_i, M_n)$ was sequentially computed.*
- *H's sequentiality was broken.*
- *H's collision resistance was broken.*

Proof. We assume $\mathsf{mp}(H_i, M_n)$ is valid while $\mathsf{path}(H_i, M_n)$ was not sequentially computed, with the aim of proving either H's sequentiality or H's collision resistance were broken.

Let d be the depth of H_i in M_n and $\mathsf{mp}(H_i, M_n) = \{x_j\}_{j=1}^{d}$. In order to verify the Merkle Proof, the verifier will compute a sequence $\{y_j\}_{j=1}^{d+1}$, such that:

1. $y_1 = H_i||x_1$ or $y_1 = x_1||H_i$.
2. For any $j \in \{2, \ldots, d\}$, $y_j = H(y_{j-1})||x_j$ or $y_j = x_j||H(y_{j-1})$.
3. $y_{d+1} = H(y_d)$ equals M_n's root.

We observe that the newly created sequence is a $d+1$-length H-Sequence as stated in Definition 7. We distinguish the following possibilities:

- The Merkle Proof contains the siblings of $\mathsf{path}(H_i, M_n)$, as described in Definition 13. In this case, it holds $\mathsf{path}(H_i, M_n) = \{H(y_j) | j \in \{1, \ldots, d\}\}$ where $\{y_j\}_{j=1}^{d}$ are the first d elements of the sequence computed by the verifier. Therefore, the given sequence can be created by concatenating elements in $\mathsf{path}(H_i, M_n)$ with elements in the Merkle Proof.

Since $\mathsf{path}(H_i, M_n)$ was not sequentially computed, we can compute the sequence $\{y_j\}_{j=1}^{d+1}$ with less than d sequential queries to H, therefore violating H's sequentiality.

- Otherwise, $\mathsf{mp}(H_i, M_n)$ must have been maliciously computed. Let $\{x'_j\}_{j=1}^{d}$ be the correct Merkle Proof according to Definition 13 and $\{y'_j\}_{j=1}^{d+1}$ the obtained sequence when verifying it. Clearly, there must exist some $j \in \{1, \dots, d\}$ such that $x_j \neq x'_j$, which implies $y_j \neq y'_j$. However, it must hold that $y_{d+1} = y'_{d+1}$ (both of them equal M_n's root). This implies the existence of some collision in H.

□

2.6. Merkle Mountain Graphs

When proving the FlyClient protocol's security guarantees, we will want to model adversaries that are able to do a certain amount of sequential work. For that purpose, we will extend the MMR definition with a graph structure that captures the sequential order of computation of the nodes in an MMR.

We will connect every leaf x in an MMR M with every left child of the nodes in the path that connects x with $\mathsf{root}(M)$. For this purpose, it will be useful to introduce the following definitions:

Definition 14. *Given an MMR M and $x \in \mathsf{leaves}(M)$, we define $\mathsf{lp}(x, M)$ as the set $\{y : \exists z \in \mathsf{path}(x, M)$ s.t. $y = \mathsf{lc}(z) \wedge \mathsf{rc}(z) \in \mathsf{path}(x, M)\}$.*

We note that in the second definition we excluded those nodes that were contained in the path itself. Now, we can define the graph structure of an MMR.

Definition 15 (Merkle Mountain Graphs)**.** *A Merkle Mountain Graph of n leaves is a Directed Acyclic Graph $G_n = (V, E)$ where V is the vertex set of an MMR M on n leaves and $E = E' \cup E''$, where:*

$$E' = \{(x, y) : y \in V \text{ and } x = \mathsf{lc}(y) \text{ or } x = \mathsf{rc}(y)\}$$

$$E'' = \{(x, y) : y \in \mathsf{leaves}(M) \text{ and } x \in \mathsf{lp}(y, M)\}$$

Given an MMG G_n and $x \in G_n$, we will denote by $\mathsf{parents}(x, G_n)$ the set of parents of x in G_n. We will define $\mathsf{root}, \mathsf{leaves}, \mathsf{lc}, \mathsf{rc}$ and mp as we did in the previous section.

Observation 4. *The edge set E' corresponds to the edge set of an MMR, thus $M = (V, E')$. This is, it connects every node with its parent in the MMG. The edge set E'' connects every leaf in an MMG with the left children of every element in the leaf's path. Indeed, given the vertex set of an MMR V and the edge set $E = E' \cup E''$ as defined in Definition 15, the graph (V, E') is an MMR as defined in Definition 11.*

Figure 7 displays an MMG on 6 leaves. As we can see, our MMG construction joins every internal node with its two children and every leaf with the left siblings of the nodes in the path from the leaf itself to the MMG's root.

Observation 5. *The AppendLeaf algorithm exposed in Table 5 can be easily extended on an MMG $G_n = (M_n, E)$ by applying it to M_n and adding whichever edges are necessary to E. We will denote by $\mathsf{MMG}(H_1, \dots, H_n)$ the MMG obtained from recursively applying AppendLeaf on leaves $H_1, \dots, H_n$. The extended algorithm is shown in Figure 8.*

Lemma 7. *Let G_n be an MMG of n leaves and H_i any of its leaves. Then, any call of the algorithm AppendLeaf on G_n will leave $\mathsf{parents}(H_i, G_n)$ unchanged.*

Proof. We prove the result by induction on n.

If $n = 1 = 2^0$, then G_n contains H_i as its single node. By MMG definition, $\mathsf{parents}(H_i, G_n) = \emptyset$. Also, the call $\mathsf{AppendLeaf}(G_n, N)$ for any node N will return an MMG G_2 of two leaves, where H_i, N are the root's left and right child, respectively. The path from H_i to G_2's root will contain H_i and the root itself, so $\mathsf{parents}(H_i, G_n) = \emptyset$. Therefore, the result holds.

If $n = 2^k$ for some $k \in \mathbb{N}$, then the call $\mathsf{AppendLeaf}(G_n, N)$ for any node N will return an MMR G_{n+1} of $n + 1$ leaves that contains G_n as its left child, therefore H_i will be contained in $\mathsf{lc}(G_{n+1})$. The path from H_i

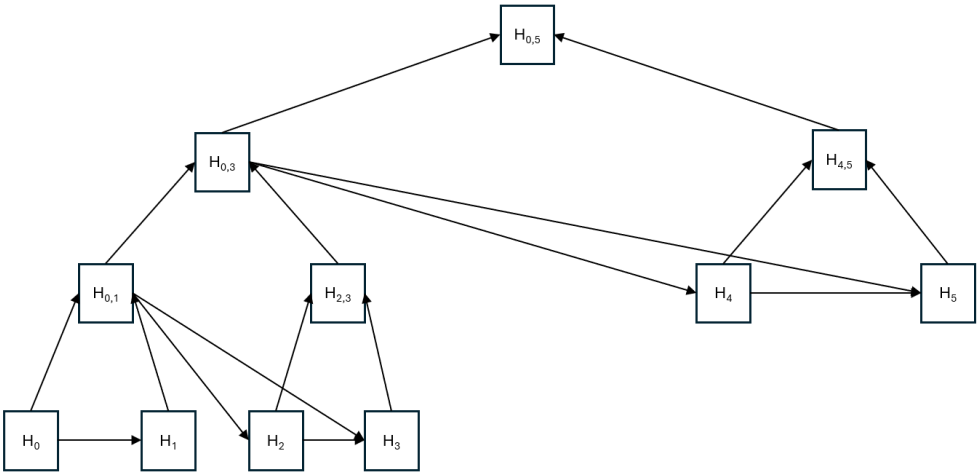

Figure 7: Illustration of an MMG G_6 of 6 leaves. It is easy to see that $\mathsf{lp}(H_3, G_6) = \{H_{0,1}, H_2\}$.

Algorithm $\mathsf{AppendLeaf}(G_n, x)$	
$//G_n$ is an MMG on n leaves, x is a leaf. Returns an MMG on $n+1$ leaves	
if $n = 2^k$ for some $k \geq 0$:	else:
1. Create new MMG G_{n+1}	1. Get rc'$\leftarrow$ AppendLeaf($\mathsf{rc}(G_n)$,x)
2. Set $\mathsf{lc}(G_{n+1})$ as G_n	2. Set $\mathsf{rc}(G_n)$ as rc'
3. Set $\mathsf{rc}(G_{n+1})$ as x	3. Set $\mathsf{root}(G_n)$ as $H(\mathsf{root}(\mathsf{lc}(G_n)), \mathsf{root}(rc'))$
4. Set $\mathsf{root}(G_{n+1})$ as $H(\mathsf{root}(G_n), x)$	4. Set $\mathsf{parents}(x, G_n) =$
5. Set $\mathsf{parents}(x, G_{n+1}) = \mathsf{parents}(x, G_n) \cup \mathsf{root}(G_n)$	$\mathsf{parents}(x, G_n) \cup \mathsf{root}(\mathsf{lc}(G_n))$
6. Return G_{n+1}	5. Return G_n

Figure 8: Description of the extended AppendLeaf algorithm

to G_{n+1}'s root will consist on the path from H_i to G_n's root plus an extra edge to G_{n+1}'s root. Therefore, the set $\mathsf{parents}(H_i, G_n)$ will remain unchanged.

If $n \neq 2^k$ for any $k \in \mathbb{N}$, then the call $\mathsf{AppendLeaf}(G_n, N)$ will return an MMR of $n+1$ leaves by recursively calling $\mathsf{AppendLeaf}(\mathsf{rc}(G_n), N)$, then recalculating G_n's root. Therefore $\mathsf{lc}(G_n) = \mathsf{lc}(G_{n+1})$. If $H_i \in \mathsf{lc}(G_n)$, then $\mathsf{parents}(H_i, G_n)$ will remain unchanged as previously seen.

Otherwise, if $H_i \in \mathsf{rc}(G_n)$, then $\mathsf{parents}(H_i, G_n) \cap \mathsf{lc}(G_n) = \{\mathsf{root}(\mathsf{lc}(G_n))\}$. Additionally, $\mathsf{rc}(G_n)$ is an MMG containing H_i as one of its leaves. By induction hypothesis, we have that $\mathsf{parents}(H_i, G_n) \cap \mathsf{rc}(G_n)$ will remain unchanged after the call $\mathsf{AppendLeaf}(\mathsf{rc}(G_n), N)$. Together with the previously obtained result $\mathsf{lc}(G_n) = \mathsf{lc}(G_{n+1})$, this proves the Lemma.

□

It will be useful to give an algebraic characterization of the set of parents of a block in an MMG. If we look at Figure 7, we will see that $\mathsf{parents}(H_2, G_n) = \{H_{0,1}\}$ and $\mathsf{parents}(H_3, G_n) = \{H_{0,1}, H_2\}$. We note there is a relation between the binary decomposition of each leaf index and the parent sets. In the first case, we have block index 2 and $\mathsf{parents}(H_2, G_n) = \{H_{0,1}\} = \{H_{0,2^1-1}\}$. In the second case, we have block index $3 = 2^1 + 2^0$ and $\mathsf{parents}(H_3, G_n) = \{H_{0,1}, H_2\} = \{H_{0,2^1-1}, H_{3-1}\}$. We will formalize this relation in the following lemma:

Lemma 8. *Let G_n be an MMG of n leaves and $H_i \in G_n$ the i-th leaf in G_n such that $i > 0$ and $i = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_j}$ for $k_1 > \ldots > k_j \in \mathbb{N}$. For every $l \in \{1, \ldots, j\}$, let $i_l = 2^{k_1} + \cdots + 2^{k_l}$. Then, it holds*

$$\mathit{parents}(H_i, G_n) = \begin{cases} \{H_{0,i_1-1}, H_{i_1,i_2-1}, \ldots, H_{i_{j-1},i_j-1}\} & \text{if } i \mod 2 = 0 \\ \{H_{0,i_1-1}, H_{i_1,i_2-1}, \ldots, H_{i_{j-2},i_{j-1}-1}, H_{i-1}\} & \text{if } i \mod 2 = 1 \end{cases}$$

Proof. Let G_n be an MMG and H_i a leaf. Lemma 7 tells us that the parents of a block wont be altered after appending new blocks to an existing MMG, so we can assume G_n has exactly $i+1$ leaves and H_i is its rightmost leaf.

We prove the result by induction on i. If $i = 1$, then H_i is the second leaf of an MMG with 2 leaves, the first block being its only parent. This matches the second case of the result we are proving.

If $i > 1$ is a power of 2, *i.e.* $i = 2^{k_1}$, then our MMG will consist on a left child with i leaves and H_i as its right child. By MMG definition, it holds that the single parent of H_i is the root of $\mathsf{MMG}(H_0, \ldots, H_{i-1})$. On the other hand, our result tells us that $\mathsf{parents}(H_i, G_n) = \{H_{0,2^{k_1}-1}\} = \{H_{0,i-1}\}$, so the result holds.

If $i > 1$ is not a power of 2, then $i = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_j}$ for some $k_1 < \ldots < k_j \in \mathbb{N}$. Then, our MMG will have a left child with 2^{k_1} leaves and a right child with $2^{k_2} + \cdots + 2^{k_j}$ leaves. By MMG definition, the only parent of H_i not contained in the MMG's right child is the roof of the left child itself. This is, the element $H_{0,2^{k_1}-1}$. We apply the induction hypothesis to obtain the set of parents of H_i in G_n's right child. The union of $H_{0,2^{k_1}-1}$ with that set will give us the desired result.

□

Observation 6. *In the previously shown examples, the case* $\mathsf{parents}(H_2, G_n)$ *corresponds to the first case in Lemma 8, while the case* $\mathsf{parents}(H_3, G_n)$ *corresponds to the second case.*

2.7. FlyClient blockchain model

In section 2.2, we formalized the bitcoin blockchain model. We can now formalize the blockchain model that is used in the FlyClient protocol. Similarly to the bitcoin blockchain, we will use two hash functions $H, G : \{0,1\}^* \to \{0,1\}^w$. Recall we are instantiating all our hash functions in the Random Oracle Model.

Definition 16 (FlyClient Block)**.** *A block is a tuple of the form* $B = \langle r, x, ctr \rangle$ *where* $r \in \{0,1\}^w, x \in \{0,1\}^*, ctr \in \mathbb{N}$.

A block $B = \langle r, x, ctr \rangle$ *is valid with respect to a difficulty target* T *and a nonce size* q *if*

$$(H(ctr, G(r, x)) < T) \wedge (ctr < q)$$

Definition 17 (FlyClient Blockchain)**.** *A blockchain* $\mathcal{C}$ *is a sequence* $B_0, \ldots, B_n$ *of valid blocks such that, for any* $B_i = \langle r_i, x, ctr \rangle \in \mathcal{C}$, $r_i = \mathsf{root}(\mathsf{MMG}(B_0, \ldots, B_{i-1}))$.

Given a chain $\mathcal{C}$, we will refer to its rightmost block as its head, and denote it by $\mathsf{head}(\mathcal{C})$. The length of a chain is its number of blocks, and we denote it by $\mathsf{len}(\mathcal{C})$. Given a chain $\mathcal{C} = B_0, \ldots, B_n$, we will refer to $\mathsf{MMG}(B_0, \ldots, B_{n-1})$ as its associated MMG. We note that the root of that MMG must be stored in B_n's header. Figure 9 illustrates a FlyClient blockchain together with its associated MMG. From now on we will refer to a FlyClient blockchain as a blockchain by default.

It must be noted that for every chain $\mathcal{C} = B_0, \ldots, B_n$, the associated MMG G_n of $\mathcal{C}$ together with B_n contain all the information of the chain. Therefore, we can refer to $\mathcal{C}$ as the pair (G_n, B_n).

If we look carefully at the illustrated example, we will see it is possible to calculate every MMG root stored in the blocks' headers from their set of parents in the associated MMG. For example, B_2 contains the root of $\mathsf{MMG}(B_0, B_1)$ which is in turn its only parent. B_3 contains the root $H_{0,2}$ of $\mathsf{MMG}(B_0, B_1, B_2)$, which can be calculated as $H(H_{0,1}, B_2)$. We observe that calculating those elements implied computing H-sequences.

Lemma 9. *Let* $\mathcal{C} = B_0, \ldots, B_n$ *be a chain with associated MMG* G_n*. For any block* $B_i, i \in \{0, \ldots, n-1\}$*, the set of parents of* B_i *in* G_n *is contained in* $\mathsf{mp}(B_i, G_n)$.

Proof. Definition 13 states that $\mathsf{mp}(B_i, G_n)$ contains every children of the elements in the path from B_i to G_n's root. On the other hand, Definition 15 states the set of parents of B_i in G_n is made of the left children of the elements in the mentioned path of which the right children is a member of the path itself.

□

We now want to prove that any blockchain has to be sequentially computed. This equals proving that its associated MMG has to be sequentially computed. Particularly, we will prove the MMG graph structure

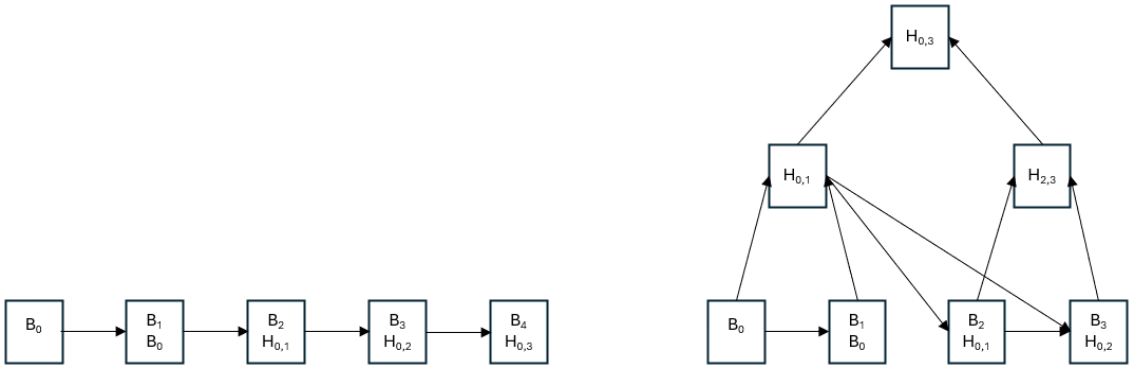

Figure 9: Illustration of a FlyClient blockchain of 5 blocks (left) together with its associated MMG (right). Observe that every block B_i contains the root of $\mathrm{MMG}(B_0, \ldots, B_{i-1})$ in its header, with the exception of the genesis block.

determines the order in which its nodes have to be computed. Since MMG are built by recursively calling the AppendLeaf algorithm, it will be interesting to prove that no call will alter the parents of its nodes.

Now, we will prove that the parents of a node are indeed enough to calculate its value. In the case of a block, this means we will be able to recompute the MMG root that is stored in its header.

Lemma 10. *Given an MMG G_n and a node $N \in G_n$, the set $\mathsf{parents}(N, G_n)$ is sufficient to compute N.*

Proof. If N is an internal node, its parents in the MMG are $\mathsf{lc}(N)$ and $\mathsf{rc}(N)$. Since $N = H(\mathsf{root}(\mathsf{lc}(N)), \mathsf{root}(\mathsf{rc}(N)))$, the result holds.

Otherwise, if N is a leaf, N will be a block as specified in Definition 16. Thus, we will need to compute the root of the MMG of the previous blocks to compute the value of N. By Lemma 7, we know the parents of a block won't be modified after appending a new block to G_n. Therefore, we can assume that N is the last block B_n in an MMG of n leaves. In that case, we have to prove that the root of $\mathsf{MMG}(B_1, \ldots, B_{n-1})$ can be recomputed from $\mathsf{parents}(B_n, G_n)$.

We prove the result by induction on n. If $n = 0$, then $\mathsf{parents}(B_n, G_n) = \emptyset$. However, B_n is G_n's genesis block so its header can still be computed from its parent set.

If $n > 1$, we have to distinguish two situations. If $n = 2^k + 1$ for some $k \in \mathbb{N}$, then G_n is the result of appending a block to an MMG of 2^k leaves. Therefore, G_n will contain $\mathsf{MMG}(B_1, \ldots, B_{n-1})$ on its left child and B_n on its right child. Given the MMG graph structure, the root of $\mathsf{MMG}(B_1, \ldots, B_{n-1})$ is B_n's sole parent, so the result holds.

Otherwise, if $n \neq 2^k + 1$, G_n is the result of appending B_n to an MMG G_{n-1} containing $n - 1 \neq 2^k$ blocks. Therefore, the call $\mathsf{AppendLeaf}(G_{n-1}, B_n)$ will return G_n by recursively calling $\mathsf{AppendLeaf}(\mathsf{rc}(G_{n-1}), B_n)$ and recalculating $\mathsf{root}(G_{n-1})$. Therefore, it will hold that $\mathsf{lc}(G_n) = \mathsf{lc}(G_{n-1})$. Also, the root of G_{n-1} can be calculated as $H(\mathsf{lc}(G_{n-1}), \mathsf{rc}(G_{n-1}))$. Since $\mathsf{root}(\mathsf{lc}(G_{n-1})) = \mathsf{root}(\mathsf{lc}(G_n)) \in \mathsf{parents}(B_n, G_n)$ by definition of MMG, it suffices to check that $\mathsf{root}(\mathsf{rc}(G_{n-1}))$ can be computed from $\mathsf{parents}(B_n, G_n) \setminus \{\mathsf{root}(\mathsf{lc}(G_n))\}$.

First, we recall that, by definition of MMG, $\mathsf{parents}(B_n, G_n)$ will be made up of the left children of the nodes in the path from B_n to $\mathsf{root}(G_n)$. Since $B_n \in \mathsf{rc}(G_n)$, that path will not contain any node in $\mathsf{lc}(G_n)$. Therefore, $\mathsf{parents}(B_n, G_n) \cap \mathsf{lc}(G_n) = \{\mathsf{root}(\mathsf{lc}(G_n))\}$. Additionally, $\mathsf{rc}(G_n)$ is an MMG that equals $\mathsf{AppendLeaf}(\mathsf{rc}(G_{n-1}), B_n)$, therefore it contains B_n as one of its leaves. By induction hypothesis, $\mathsf{root}(\mathsf{rc}(G_n))$ can be computed from $\mathsf{parents}(B_n, G_n) \setminus \{\mathsf{root}(\mathsf{lc}(\mathcal{C}))\} = \mathsf{parents}(B_n, G_n) \cap \mathsf{rc}(G_n)$.

□

Lemma 11. *Let G_n be an MMG and $B_i \in G_n$ a block. Then, the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$ which is stored in B_i's header can be recomputed by hashing the elements in $\mathsf{parents}(B_i, G_n)$ in reverse order.*

Proof. We prove the result by induction on n. If $n = 0$, then G_n is uniquely composed of the genesis block B_0, whose parents set is empty. Since the genesis block doesn't contain any MMG header, the result trivially holds.

If $n > 0$, we can use Lemma 7 to assume that G_n has i leaves and our block B_i is the rightmost block in the chain. Let $k_1 > \ldots > k_j$ such that $i = 2^{k_1} + \cdots + 2^{k_j}$. We know that G_n's left child contains the first 2^{k_1} blocks and its root, $H_{0,2^{k_1}-1}$ is precisely the first element in $\mathsf{parents}(B_i, G_n)$ as stated in Lemma 8. We want that the root $H_{2^{k_1},i-1}$ of $\mathsf{MMG}(B_{2^{k_1}}, \ldots, B_{i-1})$ can be recomputed by hashing $\mathsf{parents}(B_i, G_n) \setminus \{H_{0,2^{k_1}-1}\}$ in reverse order. Once this is proven, the root of $\mathsf{MMG}(B_1, \ldots, B_{i-1})$ will be computed as $H(H_{0,2^{k_1}-1}, H_{2^{k_1},i-1})$.

By definition 15, we know that $\mathsf{parents}(B_i, G_n) \cap \mathsf{lc}(G_n) = \{H_{0,2^{k_1}-1}\}$. Therefore, $\mathsf{parents}(B_i, G_n) \setminus \{H_{0,2^{k_1}-1}\}$ must be contained in $\mathsf{rc}(G_n)$, which equals $\mathsf{MMG}(B_{2^{k_1}}, \ldots, B_{i-1})$. By applying induction hypothesis to this smaller MMG, we get that $H_{2^{k_1},i-1}$ can indeed be computed by hashing $\mathsf{parents}(B_i, G_n) \setminus \{H_{0,2^{k_1}-1}\}$ in reverse order.

□

Finally, we need to prove that any block that wasn't sequentially computed from its parents won't be re-computable from them. This is important since it means it will cause the verifier to reject it during the FlyClient protocol.

Lemma 12. *Given an MMG G_n that was computed using a hash function H and a block $B \in G_n$. If N can be correctly recomputed from its set of parents, then some of the following possibilities hold:*

- *B was honestly computed from its set of parents.*
- *H's sequentiality was broken.*
- *H's collision resistance was broken.*

Proof. Let i be the position of B in G_n. We need to prove we can compute the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$ (*i.e.* all the preceding blocks in the MMG). We assume B was not sequentially computed from its set of parents in order to prove that either H's sequentiality or collision resistance were broken.

Lemma 8 tells us $\mathsf{parents}(B, G_n)$ is of the form $\{H_{0,i_1-1}, H_{i_1,i_2-1}, \ldots, H_{i_{j-1},i_j-1}\}$ (we will assume i is even for simplicity). Additionally, Lemma 11 tells us the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$ can be computed by hashing the elements of $\mathsf{parents}(B, G_n)$ in reverse order. This is, computing an H-sequence $\{y_l\}_{l=1}^{j}$ of length j such that:

- $y_1 = H_{i_{j-2},i_j-2} || H_{i_{j-1},i_j-1}$.
- For any $l \in \{2, \ldots, j\}$, $y_l = H_{i_{l-1},i_l-1} || H(y_{l-1})$.
- $H(y_l)$ equals the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$.

There are two possibilities in which this situation holds. The first one is that the sequence $\{y_l\}_{l=1}^{j}$ wasn't sequentially computed, which clearly violates H's sequentiality. The second one is that the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$ was successfully calculated in a different way that the described in Lemma 11. This violates H's collision resistance.

□

2.8. FlyClient graph labeling

In Section 2.6, we said that MMG are created by recursively applying the AppendLeaf algorithm on its set of leaves. However, that implies computing a series of MMG, which is inefficient if we are trying to calculate the values of the nodes in a given cahin's associated MMG.

Lemma 11 proves the MMG root stored in every block header can be computed by hashing the elements in its parents set in reverse order. Additionally, it is trivial to see that the value of an internal node is calculated by hashing its left and right children, which happen to be its parent set. We can therefore define the labeling of the elements in an associated MMG in the following way:

Definition 18. *Let G_n be an MMG and $N \in G_n$ a node. We define the label of N as*

$$\ell(N, G_n) = \begin{cases} H_r(\mathsf{parents}(N, G_n)) & \text{if } N \text{ is an internal node} \\ \langle H_r(\mathsf{parents}(N, G_n)), x, ctr \rangle & \text{if } N \text{ is a block} \end{cases}$$

where x, ctr represent a block's transaction data and PoW solution and

$$H_r(\mathsf{parents}(N, G_n)) = \begin{cases} \mathsf{parents}(N, G_n)[0] & \text{if } |\mathsf{parents}(N, G_n)| = 1 \\ H(\mathsf{parents}(N, G_n)[0], H_r(\mathsf{parents}(N, G_n)[1..])) & \text{otherwise} \end{cases}$$

Definition 19. *Let G_n be the associated MMG of a blockchain and $N \in G_n$ some node. We say N is consistent if its value matches the label $\ell(N, G_n)$ described before.*

It must be noted that, when checking the consistency of a block, we only take in count the MMG root stored in its header. This seems a natural choice, since the transaction data and the PoW solution of a block don't depend on the preceding blocks.

2.9. Variable difficulty blockchain

In order to support variable difficulty sampling, we will have to extend the FlyClient blockchain structure to support timestamps as well as difficulty weights and targets. We will extend MMG nodes to be tuples of the form $N = \langle h, x, t, T_S, T_N, w, ctr, n\rangle$ where

- $h \in \{0,1\}^w$ is the output of a hash function.
- $x \in \{0,1\}^w$ contains transaction data.
- $t \in \mathbb{N}$ is a timestamp.
- $T_S, T_N \in \mathbb{N}$ are difficulty targets.
- $w \in \mathbb{R}$ is a difficulty weight.
- $ctr \in \mathbb{N}$ is a nonce.
- $n \in \mathbb{N}$ is the number of leaves in the subgraph spawned by N.

Given two MMG nodes $l = \langle h, x, t, T_S, T_N, w, ctr, n\rangle$ and $r = \langle h', x', t', T'_S, T'_N, w', ctr', n'\rangle$ their common parent in the MMG will be $\langle H(l, r), \bot, t + t', T_S, T'_N, w + w', \bot, n + n'\rangle$. Observe that the fields related to transaction data and PoW solution were left void. We will extend the MMG structure we showed in previous sections by replacing the nodes with the new tuples. Therefore, Merkle Proofs will now be sequences of tuples.

We proceed to generalize the FlyClient blockchain model to support variable difficulty targets. We remind that FlyClient blocks are instantiated as the leaves of an MMG.

Definition 20. *A FlyClient block is a tuple $B = \langle h, x, t, T_S, T_N, 1/T_S, ctr, 1\rangle$, where $H, G : \{0,1\}^* \to \{0,1\}^w$ are hash functions. A block is said to be valid if*

$$H(ctr, G(h, x, t, T_S, T_N)) < T_S$$

It is easy to see why were the parameters instantiated this way. T_S is meant to be the block's difficulty target while T_N is meant to be the difficulty target of the next block in the chain. B's difficulty weight is instantiated as $1/T_S$ since, by PoW definition, it is harder to find a PoW solution the smaller the difficulty target is. Finally, the last parameter is systematically instantiated as 1 since the subgraph spawned by a block is the block itself.

Definition 21. *A blockchain is a sequence of valid blocks $B_0, \ldots, B_n$ such that, for every block $B_i = \langle h, x, t, T_S, T_N, 1/T_S, ctr, 1\rangle$, h equals the root of $\mathsf{MMG}(B_0, \ldots, B_{i-1})$.*

This new model introduces the possibility for an adversary to alter difficulty targets. It will be therefore interesting to see which techniques could be used by a verifier to catch an adversary by sampling a single MMG node $\langle h, x, t, T_S, T_N, w, ctr, n\rangle$.

We start bounding T_N as a function of T_S. In order to do so, we will take advantage on the limit Bitcoin imposes difficulty transitions. That way, we can reject a cheating adversary by proving no set of difficulty transitions could lead from T_S to T_N.

Lemma 13. *Given an MMG node* $\langle h, x, t, T_S, T_N, w, ctr, n\rangle$*, it must hold* $\frac{1}{\tau^p} T_S \leq T_N \leq \tau^p T_S$*, where* $p = \lfloor \frac{n}{m} \rfloor$*.*

Proof. $p = \lfloor \frac{n}{m} \rfloor$ implies the node contains at least p difficulty transitions. Therefore the maximum difficulty increase will be achieved by decreasing the difficulty target by a rate of $\frac{1}{\tau}$ exactly p times, while the maximum difficulty decrease will be achieved by increasing the difficulty target by a rate of τ exactly p times. □

In order to verify the node's weight, we will take advantage on the fact that the weight of a block equals the inverse of its assigned difficulty target. Remember that the weight of a node equals the sum of the weights of the blocks below it. We will obtain an upper bound of w by increasing the difficulty of the blocks below the node as much as it is possible. Analogously, we will obtain a lower bound by decreasing the block difficulty as much as we can.

As a start, we will assume that the leftmost block in the node's subtree is at the beginning of an epoch and $T_N = \tau^k T_S$ for some k.

Lemma 14. *Given an MMG node* $\langle h, x, t, T_S, T_N, w, ctr, n\rangle$ *such that* $T_N = \tau^k T_S$*. Assuming the leftmost block is at the beginning of an epoch, we have the following:*

1. $k \leq p$*, where* $p = \lfloor \frac{n}{m} \rfloor$*.*
2. $w \leq \frac{m}{T_S} \sum_{i=0}^{\frac{p-k}{2}-1} \tau^i + \frac{m}{T_S} \sum_{i=0}^{\frac{p+k}{2}-1} \tau^{\frac{p-k}{2}-i} = \frac{m}{T_S} \frac{1-\tau^{\frac{n-k}{2}}}{1-\tau} + \frac{m}{T_S} \tau^{1-k} \frac{\tau^{\frac{n+k}{2}}-1}{\tau-1}$
3. $w \geq \frac{m}{T_S} \sum_{i=0}^{\frac{p+k}{2}-1} \tau^{-i} + \frac{m}{T_S} \sum_{i=0}^{\frac{p-k}{2}-1} \tau^{-\frac{p+k}{2}+i} = \frac{m}{T_S} \tau^{1-\frac{p-k}{2}} \frac{\tau^{\frac{p+k}{2}}-1}{\tau-1} + \frac{m}{T_S} \tau^{-\frac{p+k}{2}} \frac{1-\tau^{\frac{p-k}{2}}}{1-\tau}$

Proof. 1. Consequence of Theorem 13.

2. We will obtain an upper bound of w by maximizing the difficulty of the p epochs underneath our node. This will be achieved by decreasing the difficulty target by a rate of $\frac{1}{\tau}$(*i.e.*, increasing the epoch difficulty by a rate of τ) as many times as we can, then increasing the difficulty target by a rate of τ(*i.e.*, decreasing the epoch difficulty by a rate of $\frac{1}{\tau}$) as many times as we need to verify $D_N = \tau^k D_S$. More specifically, we multiply the difficulty target by $\frac{1}{\tau}$ in the first $\frac{p-k}{2}$ epochs, then we multiply by τ in the remaining $\frac{p+k}{2}$ epochs.

3. We will obtain a lower bound of w by minimizing the difficulty of the p epochs underneath our node. This will be achieved by increasing the difficulty target by a rate of τ(*i.e.*, decreasing the epoch difficulty by a rate of $\frac{1}{\tau}$) as many times as we can, then decreasing the difficulty target by a rate of $\frac{1}{\tau}$(*i.e.*, increasing the epoch difficulty by a rate of τ) as many times as we need, ensuring $D_N = \tau^k D_S$. More specifically, we multiply the difficulty target by τ in the first $\frac{p+k}{2}$ epochs, then we multiply by $\frac{1}{\tau}$ in the remaining $\frac{p-k}{2}$ epochs.

□

Lemma 15. *Given an MMG node* $\langle h, x, t, T_S, T_N, w, ctr, n\rangle$ *such that* $T_N = \tau^{k+s} T_S$*, for some* $k \in \mathbb{Z}, s \in [0, 1)$*. Assuming the leftmost block is at the beginning of an epoch, we have the following:*

1. $k \leq p - 1$*, where* $p = \lfloor \frac{n}{m} \rfloor$*.*
2. $w \leq \frac{m}{T_S} \sum_{i=0}^{\frac{p-k}{2}} \tau^i + \frac{m}{T_S} \sum_{i=0}^{\frac{p+k}{2}-1} \tau^{\frac{p-k}{2}-s-i} = \frac{m}{T_S} \frac{1-\tau^{\frac{p-k}{2}+1}}{1-\tau} + \frac{m}{T_S} \tau^{1-k-s} \frac{\tau^{\frac{p+k}{2}}-1}{\tau-1}$
3. $w \geq \frac{m}{T_S} \sum_{i=0}^{\frac{p+k}{2}} \tau^{-i} + \frac{m}{T_S} \sum_{i=0}^{\frac{p-k}{2}-1} \tau^{-\frac{p+k}{2}-s+i} = \frac{m}{T_S} \tau^{-\frac{p-k}{2}} \frac{\tau^{\frac{p+k}{2}}-1}{\tau-1} + \frac{m}{T_S} \tau^{-\frac{p+k}{2}-s} \frac{1-\tau^{\frac{p-k}{2}}}{1-\tau}$

Proof. 1. Consequence of Theorem 13.

2. We will obtain an upper bound of w by maximizing the difficulty of the p epochs underneath our node. Similarly to Theorem 14, we will decrease T_S in the first $\frac{p-k}{2}$ epochs by a rate of $\frac{1}{\tau}$, obtaining a difficulty $T'_S = \tau^{-\frac{p-k}{2}} T_S$, then we will raise the target by a smaller rate of τ^s, getting $T''_S = \tau^{-\frac{p-k}{2}+s} T_S$. Finally, we will increase the target by a rate of τ in the last $\frac{p+k}{2}$ getting a final difficulty $T_N = \tau^{k+s} T_S$.

3. We will obtain a lower bound of w by minimizing the difficulty of the p epochs underneath our node. Similarly to Theorem 14, we will increase T_S in the first $\frac{p-k}{2}$ epochs, obtaining a difficulty $T'_S = \tau^{\frac{p+k}{2}} T_S$. We will then raise the target by a smaller rate of τ^s, getting $T''_S = \tau^{\frac{p+k}{2}+s} T_S$. Finally, we will decrease the target by a rate of $\frac{1}{\tau}$ in the last $\frac{p-k}{2}$ getting a final difficulty $T_N = \tau^{k+s} T_S$.

□

Now, we bound w in the most possible general case where $n \bmod m \neq 0$ and we don't know how many of the leftmost leaves have difficulty T_S:

Finally, we will do some checking on the node's timestamp. If there has been a difficulty decrease(i.e. $T_S \leq T_N$), then we will be able to provide a lower bound for t.

Lemma 16. *A maximal difficulty decrease will correspond to an epoch length of at least $\tau \frac{m}{f}$.*

Proof. Assume an epoch has been mined in Δ rounds with a difficulty target T. A maximal difficulty decrease will happen when

$$\frac{n_0}{n(T, \Delta)} T_0 > \tau T$$

by definition of $n(T, \Delta)$, this equals

$$\frac{q T_0 n_0 \Delta T}{2^k m} > \tau T \implies \frac{f \Delta}{m} > \tau \implies \Delta > \tau \frac{m}{f}$$

where in the first inequality we used the hypothesis $f(T_0, n_0) = q T_0 n_0 / 2^k = f$. □

Lemma 17. *Let $H = \langle h, x, t, T_S, T_N, w, ctr, n \rangle$ be an MMR node such that $\tau^k T_S \leq T_N$, then it holds $t \geq k\tau \frac{m}{f}$.*

Proof. The minimum possible node timestamp will correspond to the minimum weight we could possibly assign to it. As it was previously discussed, minimizing the weight implies performing at least k maximum difficulty decreases. Theorem 16 implies that every epoch that leads to a maximum difficulty decrease lasts at least $\tau \frac{m}{f}$ rounds. Therefore, it holds $t \geq k\tau \frac{m}{f}$. □

On the other hand, if there has been a difficulty increase(i.e. $T_S \geq T_N$), we will be able to provide an upper bound for t.

Lemma 18. *A maximal difficulty increase will correspond to an epoch length of at most $\frac{m}{\tau f}$.*

Proof. Assume an epoch has been mined in Δ rounds with a difficulty target T. A maximal difficulty increase will happen when

$$\frac{n_0}{n(T, \Delta)} T_0 < \frac{1}{\tau} T$$

by definition of $n(T, \Delta)$, this implies

$$\frac{q T_0 n_0 \Delta T}{2^k m} < \frac{1}{\tau} T \implies \frac{f}{m} \Delta < \frac{1}{\tau} \implies \Delta < \frac{m}{\tau f}$$

where in the first inequality we used the hypothesis $f(T_0, n_0) = q T_0 n_0 / 2^k = f$. □

2.10. The (c, L)-Assumption

The security guarantees of the FlyClient protocol will rely on an assumption on the adversarial ability to produce valid blocks. Concretely, we will assume that an adversary that forks an honest chain and adds blocks to it won't be able to produce more that a certain rate of valid blocks. Interestingly, SNACKs' security guarantees also rely on this assumption. Indeed, the work done in this section is heavily inspired in [Abu+22], which formalizes the original assumption presented in [Bün+19].

We will start focusing ourselves in the static model FlyClient blockchain as exposed in Section 2.7. We will assume that our blockchain protocol satisfies the common prefix property for some parameter $k \in \mathbb{N}$.

This is, honest miners will seek agreement on a chain of maximal length, with the guarantee that every pair of honest chains will differ at most in the k last blocks.

It is easy to see the adversarial ability to produce a fully valid fork will become smaller the longer the fork is. Shorter forks have bigger variance since an adversary will have higher chances to solve all the required Proofs of Work. On top of that, it is not interesting to verify transactions in a block that isn't placed in the stable prefix of the chain. Therefore, we will require an adversary to produce forks longer than the stable prefix parameter.

This, however, doesn't prevent the adversary from choosing between different forking strategies. An adversary could choose to extend its fork beyond the honest chain length, obtaining a longer chain. This opens the possibility for the existence of forks that contain the full stable prefix of the honest chain.

Definition 22 ((c, L)-Fork, static case [Abu+22])**.** *Let Π be a blockchain protocol satisfying the common prefix property with parameter $k \in \mathbb{N}$. Let $n_h \in \mathbb{N}$ the honest length of the chains at some fixed time and let $B_0, \ldots, B_{s_h}$ be the stable prefix of the honest chains, with $s_h := n_h - k$. Let $L > k, c \in (0, 1]$. Then,*

- *An L-fork is an adversarially computed sequence of blocks $B'_0, \ldots, B'_{n^*}$, with $n^* \geq n_h$ such that some of the following possibilities holds:*

  1. *$n^* \geq n_h + L - k$ and $\forall i \in \{0, \ldots, s_h\}, B_i = B'_i$.*
  2. *There exists $f \leq s_h$ such that $B_f \neq B'_f$.*

  We will denote the first adversarially computed block in an L-fork $B'_0, \ldots, B'_{n^*}$ as the forking point. In the second case of the definition, the forking point will be the smaller f such that $B_f \neq B'_f$.

- *A (c, L)-fork is an L-fork $B'_0, \ldots, B'_{n^*}$ such that the suffix $B'_f, \ldots, B'_{n^*}$ contains at least $c(n^* - f)$ valid blocks, where f is the forking point described before.*

The first case in the definition corresponds with the possibility of computing a fork that contains the whole stable prefix. This is only possible if the fork is at least L blocks longer that the stable prefix. In the second case, some block in the stable prefix is not included in the fork. Note that we are only taking in count the stable prefix of the honest chains. Since honest miners' chains can differ after the stable prefix, it is not possible to define a full honest chain.

In order to generalize the (c, L)-Assumption to the dynamic difficulty model, we have to extend the definition of a fork to the variable difficulty blockchain we introduced in Section 2.9. Remember that miners in the variable difficulty model seek to agree in a chain of maximal weight. Given a chain $\mathcal{C} = B_0, \ldots, B_n$, we define its cumulative weight $W(\mathcal{C}) := \sum_{i=0}^{n} W(B_i)$, where $W(B_i)$ is B_i's weight. Given a sequence of blocks $S = B_i, \ldots, B_j$, we analogously define its weight as $W(S) := \sum_{k=i}^{j} W(B_k)$.

It must be noted that the common prefix property in the dynamic difficulty model is identically enounced as in the static difficulty model. However, honest miners will no longer hold chains of the same length. Because of that, we will define the stable prefix of the honest chains as the result of removing the last k blocks from the longest honest chain.

Definition 23 ((c, L)-Fork, dynamic case [Abu+22])**.** *Let Π be a blockchain protocol satisfying the common prefix property with parameter $k \in \mathbb{N}$. Let $w_h \in \mathbb{R}$ be the honest weight of the chain at some fixed time and let $B_0, \ldots, B_{s_h}$ be the stable prefix of the honest chains. Let $L > k, c \in (0, 1]$. Then,*

- *An L-fork $\mathcal{F}$ is an adversarially computed sequence of blocks $B'_0, \ldots, B'_{n^*}$, with $W(\mathcal{F}) \geq w_h$ such that some of the following possibilities holds:*

  1. *$W(\mathcal{F}) \geq W(B_0, \ldots, B_{s_h}) + W(B'_{n^*-L}, \ldots, B'_{n^*})$ and $\forall i \in \{0, \ldots, s_h\}, B_i = B'_i$.*
  2. *There exists $f \leq s_h$ such that $B_f \neq B'_f$.*

  We will denote the first adversarially computed block in an L-fork $B'_0, \ldots, B'_{n^*}$ as the forking point. In the second case of the definition, the forking point will be the smaller f such that $B_f \neq B'_f$.

- *A (c, L)-fork is an L-fork $B'_0, \ldots, B'_{n^*}$ such that the valid blocks in the suffix $B'_f, \ldots, B'_{n^*}$ have cumulative weight greater or equal than $c \cdot W(B'_f, \ldots, B'_{n^*})$, where f is the forking point as defined before.*

We will now formalize the assumption we are making on the adversarial ability to produce a fork. We will restrict the assumption to adversaries that respect the retargeting rules. Otherwise, the assumption might be unrealistic since adversaries could artificially increase the weight of a fork by raising the difficulty targets of the blocks it computes. We will discuss the feasibility of such an attack in the future.

Assumption 2 ((c, L)-Assumption)**.** *Let Π be a blockchain protocol satisfying the common prefix property with security parameter $w \in \mathbb{N}$. Then, an adversary trying to compute a (c, L)-fork that respects the difficulty transition rules will succeed with a negligible probability in w.*

Recall that the security parameter of a blockchain protocol differs from the common prefix property parameter. Typically, this security parameter will be the length of the used hash function.

Inferring parameters for which the (c, L)-assumption holds is an open problem. [Bün+19] offers a solution for the static model by assuming the adversary mines its fork privately. However, this ignores the possibility of tricking honest miners into contributing to the fork, outpowering the adversarial computing capabilities. In the case $c = 1$, it is easy to see a connection between the common prefix property and the (c, L)-assumption:

Lemma 19. *The common prefix property with parameter k implies the $(1, k + 1)$-Assumption.*

Proof. Assume that an adversary managed to compute a fully valid fork $\mathcal{F}$ of $k + 1$ blocks at some fixed time. We can assume, without loss of generality, that the fork length equals the honest chain length.

Since all its blocks are valid, this fork could be accepted by an honest miner as a valid chain. At the same time, a different honest miner might be holding a different chain $\mathcal{C}$ not related with the fork. Since the $k + 1$ last blocks of $\mathcal{F}$ have been adversarially generated, it holds $\mathcal{F}^{\lceil k} \not\preceq \mathcal{C}$ and ${}^{\mathcal{C}\lceil k} \not\preceq \mathcal{F}$, contradicting the common prefix property.

□

3. The FlyClient protocol

We proceed to explain the FlyClient protocol as exposed in [Bün+19]. We will start centering ourselves in the interactive static difficulty model for the moment. After that, we will introduce the slightly extended variable difficulty model and non interactive protocols.

The FlyClient protocol is a protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ in which the former holds a blockchain $\mathcal{C}$ and the later expects to obtain a commitment to the stable prefix of $\mathcal{C}$. In order to do so, $\mathcal{V}$ will query $\mathcal{P}$ a logarithmic amount of blocks according to some sampling distribution f, that will be specified in Section 3.1, and will verify them by checking their Merkle Proof and Proof of Work. If any of the checks fails, $\mathcal{P}$ will get rejected. A sketch of the protocol is shown here:

Prover$(\mathcal{C}, H, f)$		**Verifier**(B_0, H, f)
get $B_n = \mathsf{head}(\mathcal{C})$	$\xrightarrow{B_n}$	verify B_n's PoW
	$\xleftarrow{i}$	sample i according to f
get $B_i, \mathsf{mp}(B_i, \mathcal{C})$	$\xrightarrow{B_i, \mathsf{mp}(B_i, \mathcal{C})}$	verify B_i's PoW, $\mathsf{mp}(B_i, \mathcal{C})$

The complete version of this protocol is shown in Section 3.2. Once a chain $\mathcal{C}$ is accepted, $\mathcal{V}$ will be able to verify any transaction x contained in $\mathcal{C}$'s stable prefix by querying $\mathcal{P}$ for the header of the block B our transaction is contained in, $\mathsf{mp}(B, \mathcal{C})$ and the Merkle Proof that connects x to the header of B. $\mathcal{V}$ will have to verify B's Proof of Work and Merkle Proof and check that x belongs to the Merkle Tree whose root is stored in B's header.

In order to have security guarantees against a malicious prover, we will use the (c, L)-Assumption as mentioned in Section 2.10. We need to ensure that every adversary will need to compute a fork of length at least L. For this purpose, once a chain $\mathcal{C}$ is accepted, we will only verify transactions in $\mathcal{C}^{\lceil L}$.

We will assume that any verifier will be connected to a set of provers, each of whom will commit a chain. The verifier will select the prover with the heaviest chain first. If the chain is rejected, it will continue querying the rest of the provers by order of chain weight. This protocol is displayed in Figure 10.

On input chains $(C_i)_{i\in\mathbb{N}}$, the light client does the following:
1. For all chains, ordered by decreasing values of $\mathsf{len}(\mathcal{C}_i)$:
   (a) Apply the FlyClient protocol to verify $\mathcal{C}_i$
   (b) If $\mathcal{C}_i$ is accepted, Return $\mathcal{C}_i^{\lceil L}$
2. Return $\perp$

Figure 10: Prover selection protocol

There is a caveat: a verifier needs to be connected to an honest prover in order to have security guarantees. Otherwise, an adversary could compute a fully valid chain containing at least L adversarial blocks, getting a verifier to accept it even if it was shorter than the honest chains. We show it here:

Lemma 20. *A verifier $\mathcal{V}$ has no guarantees against a single malicious prover $\mathcal{P}$.*

Proof. Since $\mathcal{P}$ is the only prover taking part in the protocol, $\mathcal{V}$ will always query it. $\mathcal{P}$ could therefore compute a fully honest fork longer than L blocks without having to catch up with the honest longest chain. Since the mentioned fork contains no malicious blocks, $\mathcal{V}$ will accept it with probability 1.

□

This situation could be overcome if the verifier knew the honest length of the chains. In this case, the verifier could automatically reject any chain shorter than the honest chain.

On input chains $(C_i)_{i\in\mathbb{N}}$, the light client does the following:
1. For all chains, ordered by decreasing values of $\mathsf{len}(\mathcal{C}_i)$:
   (a) Apply the FlyClient protocol to verify $\mathcal{C}_i$
   (b) If $\mathcal{C}_i$ is accepted and $\mathsf{len}(\mathcal{C}_i) \geq n_h$, Return $\mathcal{C}_i^{\lceil L}$
2. Return $\perp$

Figure 11: Modified prover selection protocol in the case the honest chain length n_h is known

We proceed to describe the sampling distribution that will be used in the protocol.

3.1. FlyClient sampling distribution

As it was previously mentioned, the FlyClient protocol uses a sampling distribution f that determines the blocks to be queried. We need to determine which distribution has the higher chances to query a malicious block. As a start, we study which strategies could an adversary follow to compute a chain.

First place we will show that an optimal adversary will only proceed by adding blocks to the end of an already computed chain. We will prove this by contradiction, showing that an adversary who modifies an existing chain will get rejected with overwhelming probability.

An adversary can modify an existing chain in two ways: either inserting a block in the middle of the chain or modifying an existing block. The two possibilities are covered below.

Lemma 21. *Assume an adversary takes a chain $\mathcal{C} = B_0, \ldots, B_n$ with its respective MMR M_n and inserts a block in the position k. Then, every block in $B_k \ldots B_n$ will be rejected by the verifier.*

Proof. Since a new block was inserted in the k-th position, every block in $B_k \ldots B_n$ will get its position in the chain increased by 1. If one of such blocks is queried and the prover tries to recycle the Merkle Proofs in the original MMR, the verifier will hash them in the wrong order and will reject the prover.

Otherwise, if it computes the new chain's MMR, every affected block B_i will contain the root of the MMR of $B_1 \dots B_{i-1}$. Since a new block was inserted somewhere between B_1 and B_{i-1}, the verifier won't be able to obtain r_{i-1} from B_i's new Merkle Proof.

□

Lemma 22. *Assume an adversary takes a chain $\mathcal{C} = B_0, \dots, B_n$ with its respective MMR M_n and modifies the block in the position k. If the adversary updates the MMR values, every block in $B_{k+1} \dots B_n$ will be rejected by the verifier. Otherwise, the modified block will be rejected by the verifier.*

Proof. If B_k is modified and the MMR is updated, then for every $j > k$, the root of the MMR of $B_1 \dots B_j$ will be modified, so it wont match the root stored at the header of B_{j+1}.

Otherwise, if the MMR is not updated, the prover wont be able to compute a Merkle Proof for the new block.

□

Since modifying an existing chain does not seem an good strategy, we can assume adversaries will fork an existing chain and add new blocks to its end. The (c, L)-assumption tells us that, if the fork is longer than L blocks, it will contain at least an $(1 - c)$ rate of wrongly computed blocks. We want our sampling distribution to query one of those wrongly computed blocks, hence catching the adversary. Since they are likely to be placed at the very end of the chain, an increasing distribution seems to be the most logical choice. We remember that f is increasing if, $\forall x, y \in \mathbb{N}, x < y \implies f(x) \leq f(y)$.

For convenience issues, we will identify any chain $\mathcal{C} = B_0, \dots, B_n$ as a continuous space between 0 and 1, where every block B_k is identified with the interval $[\frac{k}{n}, \frac{k+1}{n})$. In particular, 0 represents the genesis block and n the rightmost block in the chain. Our sampling distribution will therefore be a function $f : [0, 1] \to [0, 1]$ such that $P(B_k) = \int_{k/n}^{(k+1)/n} f(x)dx$.

Lemma 23. *A non-increasing sampling distribution f is not uniquely optimal, i.e., there exists another distribution with equal or higher probability of catching the adversary.*

Proof. Assume a non-increasing sampling distribution $f : [0, 1] \to [0, 1]$. This implies there exist numbers $x_1, x_2 \in [0, 1], d > 0$ and intervals $I_1 = [x_1, x_1 + d], I_2 = [x_2, x_2 + d]$ such that $x_1 + d \leq x_2 \leq 1 - d$ verifying that, for any $y_1 \in I_1, y_2 \in I_2, f(y_1) > f(y_2)$. This is, elements in I_1 will have higher chances of being queried than those in I_2.

Let $a \in [0, 1)$ be the forking point chosen by an adversary $\mathcal{A}$ who will work under the limitations premised by the (c, L)-assumption. $\mathcal{A}$ will therefore mine the blocks in the positions $[a, 1]$ of the chain, being able to insert at most $c(1 - a)$ valid blocks. Additionally, $\mathcal{A}$ will decide freely in which positions the invalid blocks will be inserted.

Given any strategy in which $\mathcal{A}$ places a valid block in I_2 and an invalid block in I_1, there exists another strategy that places an extra valid block in I_1 and removes it from I_2. The converse is not true, since if $x_1 < a$ it might not be possible to insert more invalid blocks in I_1. We conclude from this that, for every adversarial strategy that includes more invalid blocks in I_1 than in I_2, there exists a different strategy that includes more invalid blocks in I_2 than in I_1.

Let $\epsilon > 0$ be an arbitrarily small number and $f'(x)$ be a sampling distribution such that

$$f'(x) = \begin{cases} f(x) & \text{if } x \notin I_1 \cup I_2 \\ f(x) - \epsilon & \text{if } x \in I_1 \\ f(x) + \epsilon & \text{if } x \in I_2 \end{cases}$$

$f'(x)$ will query blocks in I_2 with slightly higher probability than $f(x)$. We conclude from the analysis done before that $f'(x)$ will query an invalid block with higher chances than $f(x)$. This proves the desired result.

□

Since adversaries are likely to know the sampling distribution f, we should expect them to place all the honestly mined blocks they get at the end of their chain. Assuming they fork at a point $a \in [0, 1)$, and according to the (c, L)-assumption, an adversary will be able to mine at most a $(1 - a)c$ fraction of the chain.

Therefore, an optimal adversary will have wrongly computed blocks in the interval from a to $1-(1-a)c$ and honestly mined blocks in the interval from $1-(1-a)c$ to 1.

For any sampling distribution f, the probability to catch an adversary who forked at a point a in a single sample is

$$\frac{\int_a^{1-(1-a)c} f(x)dx}{\int_0^1 f(x)dx}$$

considering all the points $a \in [0,1)$ the adversary could fork from, the probability is

$$p = \min_{0 \le a < 1} \frac{\int_a^{1-(1-a)c} f(x)dx}{\int_0^1 f(x)dx}$$

We want to find a distribution that maximizes p. Ideally, we would like to make p independent from the forking point a. [Bün+19] uses differential analysis to get to the distribution

$$f(x) = \frac{1-c}{c(1-x)}$$

which verifies

$$\int_a^{1-(1-a)c} f(x)dx = \frac{(c-1)\log(c)}{c}$$

unfortunately, f is not normalized and can't be normalized since it tends to infinite when x approaches 1. In order to fix this, we will restrict f to the interval $[0, 1-\delta]$ for some $\delta \in (0,1)$ and manually verify all the blocks in the last δ fraction of the chain. By doing that, we get the following normalized distribution

$$g(x) = \frac{f(x)}{\int_0^{1-\delta} f(x)dx} = \frac{1}{(x-1)\log(\delta)}$$

The probability of catching the adversary is

$$p = \int_a^{1-(1-a)c} \frac{1}{(x-1)\log(\delta)}dx = \frac{\log|x-1|}{\log(\delta)}\Big|_a^{1-(1-a)c} = \log_\delta((1-a)c) - \log_\delta(1-a) = \log_\delta\left(\frac{(1-a)c}{1-a}\right) = \log_\delta(c)$$

which is independent from the forking point a too. We can see that, if $\delta > c$, then $\log_\delta(c) > 1$. This makes the probability to catch a malicious block not being well-defined. This happens because, when calculating this probability, we assumed the optimal interval of invalid blocks $[a, 1-(1-a)c]$ would be contained in the sampling distribution domain $[0, 1-\delta]$. However, if $\delta > c$, then the amount of blocks we will manually check at the end of the chain will be bigger than the amount of valid blocks the adversary managed to produce in its fork, meaning that the last δ fraction of the chain will always contain an invalid block. This won't be a problem, though. In such a situation, any adversary would be trivially caught with probability 1. We will therefore assume from now on that $c \ge \delta$, ensuring that $\log_\delta(c) \le 1$.

Finally, we prove that g is indeed an optimal sampling distribution:

Lemma 24. *Assuming a fraction $\delta = c^k, k \in \mathbb{N}$ of the last blocks in the chain only contains valid blocks, and no adversary can create more than a c rate of valid blocks, the sampling distribution $g(x) = \frac{1}{(x-1)\log(\delta)}$ maximizes the probability of catching an adversary that optimizes the placement of invalid blocks.*

Proof. Given that $\delta = c^k$ for some $k \in \mathbb{N}$, the probability of catching the adversary with a single query to g is $p = \log_\delta(c) = 1/k$.

Let g^* be a different sampling distribution such that the probability of catching the adversary in a single query is

$$p^* = \min_{0 \le a \le \frac{c-\delta}{c}} \int_a^{1-(1-a)c} g^*(x)dx$$

if we take a fork point of the form $1 - c^i, i \in \{0, \dots, k-1\}$ it must hold

$$p^* \leq \int_{1-c^i}^{1-c^{i+1}} g^*(x)dx$$

from which it follows

$$1 = \int_0^{1-c^k} g^*(x)dx = \sum_{i=0}^{k-1} \int_{1-c^i}^{1-c^{i+1}} g^*(x)dx \geq kp^* \implies p^* \leq \frac{1}{k} = p$$

□

Observation 7. *Lemma 24 introduces a restriction on the choice of δ. We can see that the smaller the k we choose, the bigger p will be. This increase in the probability of catching an adversary will however come at the cost of bigger proof sizes, since we will have to check a bigger fraction of blocks at the end of the chain.*

3.2. The interactive FlyClient protocol in static difficulty

We start to describe the simplest possible form of the FlyClient protocol. This is, that one in which a constant difficulty target T is assumed in the whole committed chain and interaction between the verifier and the prover is needed to complete the protocol.

The protocol will be parameterized by the number t of queries the verifier will make the prover and the suffix δ of the chain that will be manually queried. Additionally, we will consider the block difficulty target T as a parameter. This won't be the case in the variable difficulty versions of the protocol where each block will have an assigned difficulty target. Finally, we reflect the fact a prover is supposed to be holding a blockchain $\mathcal{C}$ by representing it as a prover input. The description of the protocol is shown in Figure 12.

Prover $(\mathcal{P} = \mathcal{P}_0, \mathcal{P}_1)$

Stage P_0: On input $\mathcal{C}$:

1. parse $\mathcal{C} = (B_n, G_{n-1})$
2. send B_n to $\mathcal{V}_0$

Stage P_1: On input $\tau = (\tau_i)_{i=1}^t$:

1. $\forall i \in \{1, \dots, t\}$ do
   $y_i := (B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))$
2. $\forall i \in \{1, \dots, \delta\}$ do
   $y_{t+i} := (B_{n-\delta+i}, \mathsf{mp}(B_{n-\delta+i}, G_{n-1}), \mathsf{parents}(B_{n-\delta+i}, G_{n-1}))$
3. send $y := (y_1, \dots, y_{t+\delta})$ to $\mathcal{V}_1$

Verifier $(\mathcal{V} = \mathcal{V}_0, \mathcal{V}_1)$

Stage V_0: On input B_n:

1. parse $B_n = \langle r_{n-1}, x, ctr \rangle$
2. $b_0 := 1$ iff $H(ctr, G(r_{n-1}, x)) < T$
3. $\forall i \in \{1, \dots, t\}$ sample τ_i according to $f(\delta)$
4. send $\tau := (\tau_1, \dots, \tau_t)$ to $\mathcal{P}_1$

Stage V_1: On input $y = (B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))_{i=1}^{t+\delta}$:

1. $\forall i \in \{1, \dots, t+\delta\}$ do
   (a) parse $B_{\tau_i} = \langle r_{\tau_{i-1}}, x, ctr \rangle$
   (b) $b_i := 1$ iff
   (-) $\mathsf{parents}(B_{\tau_i}, G_{n-1}) \subseteq \mathsf{mp}(B_{\tau_i}, G_{n-1})$
   (-) $H(ctr, G(r_{\tau_{i-1}}, x)) < T$
   (-) $\mathsf{vmp}(B_{\tau_i}, \tau_i, r_{n-1}, \mathsf{mp}(B_{\tau_i}, G_{n-1})) = 1$
   (-) $\mathsf{vroot}(r_{\tau_{i-1}}, \mathsf{parents}(B_{\tau_i}, G_{n-1})) = 1$
2. output $\bigwedge_{i=0}^{t+\delta} b_i$

Figure 12: Description of the FlyClient protocol. $\mathcal{C}$ represents a FlyClient blockchain. $T, t \in \mathbb{N}, \delta \in (0, 1]$ are parameters of the protocol, which represent the block difficulty target and the number of queries made to the prover, respectively. δ represents the fraction of blocks in the end of the chain that are manually checked.

We proceed to explain the steps taken during the protocol:

1. Prover $\mathcal{P}$ commits a chain $\mathcal{C} = B_0, \dots, B_n$ by sending its last block B_n to the verifier $\mathcal{V}$. Recall that, as shown in Section 2.7, blockchains will be represented as a tuple consisting on the B_n and its associated MMG.

2. $\mathcal{V}$ checks B_n's Proof of Work and generates a query $\{\tau_i\}_{i=1}^{t}$ of size t using the sampling distribution f. If B_n's checking fails, $\mathcal{V}$ could abort the protocol and reject the committed chain, saving some computations. We didn't reflect this in Figure 5 in order to make the protocol more readable.

3. For every τ_i, $\mathcal{P}$ provides $\mathcal{V}$ with the header of block $B_{\tau_i}, \mathsf{parents}(B_{\tau_i}, G_{n-1})$ and $\mathsf{mp}(B_{\tau_i}, G_{n-1})$. Additionally, $\mathcal{P}$ will send $\mathcal{V}$ all the blocks in the last δ fraction in the chain, together with their respective Merkle Proof.

4. For every received block, $\mathcal{V}$ will

   - Check the block's Proof of Work.
   - Apply algorithm vmp to check the block's Merkle Proof.
   - Apply algorithm vroot to check the block was correctly computed with respect to the graph labeling function we defined.

   If all the checks were correct, $\mathcal{V}$ will accept the chain. Otherwise, it will reject it.

It is left to explain the mentioned vmp and vroot algorithms. We show them here:

Algorithm $\mathsf{vmp}(B, i, r, mp)$

```
//B is a block, i is B's index, r is an MMG root, mp is a Merkle Proof
y ← B, k ← i
for x in mp:
    if k  mod 2 = 0 then:
        y ← H(y, x)
    else:
        y ← H(x, y)
    k ← ⌊k/2⌋, i ← ⌊i/2⌋
if y = r then:
     return 1
else:
     return 0
```

Figure 13: Implementation of the vmp algorithm. vmp checks wether mp is a valid Merkle Proof for a block B with index i in an MMG with root r.

Algorithm $\mathsf{vroot}(r, p)$

```
//r in an MMG root, p is the parent set of an MMG block
y ← H_r(p)
if y = r then:
    return 1
else:
    return 0
```

Figure 14: Implementation of the vroot algorithm. Given a block B with parent set p and stored root r, vroot checks wether r was correctly computed from p.

Where H_r is the function

$$H_r(p) = \begin{cases} p[0] & \text{if } |p| = 1 \\ H(p[0], H_r(p[1..])) & \text{otherwise} \end{cases}$$

that was previously mentioned in Section 2.8.

3.3. FlyClient security guarantees

We proceed to prove the security claims that were made at the end of [Bün+19]. In particular, we will prove that any adversary that works under the rules of the (c, L)-assumption will get rejected with overwhelming probability. Also, we will take into account the possibility for an adversary not computing the whole blockchain sequentially, saving some computational power in order to maximize the amount of blocks with a valid Proof of Work. As we will see, the adversarial success chances will decrease exponentially with the rate of nodes that weren't sequentially computed.

We will first introduce some concepts that will help us in this section.

Definition 24 ([CP18])**.** *Given an MMG G_n and a set $S \subset G_n$ of nodes, we denote $\hat{S} = \{x \in \mathsf{leaves}(G_n) | \mathsf{path}(x, G_n) \cap S \neq \emptyset\}$. This is, $\hat{S}$ represents the set of leaves of G_n that are below S. We denote by S^* the minimal set of nodes such that $\hat{S^*} = S^*$. We denote by D_S the set of nodes that are in S or below some node in S.*

Observation 8. *Given any MMG node v, D_v is always an MMG. Given a node set $S = \{v_1, \dots, v_n\}$, it holds $D_S = \cup_{i=1}^{n} D_{v_i}$. Therefore, D_S will equal the union of some MMG.*

To illustrate this, we refer to Figure 7. Given the node set $S = \{H_{0,1}, B_2, B_4, B_5\}$, S^* will consist on the set $\{H_{0,1}, B_2, H_{4,5}\}$. D_S will consist on the set $\{H_{0,1}, B_0, B_1, B_2, B_4, B_5\}$. Recall that for any node set S, D_S will always be the union of some MMG.

Lemma 25 ([CP18])**.** *Given an MMG $G_n = (M_n, E)$ of n leaves and a node set S, the subgraph of G_n on vertex set $M_n \setminus D_{S^*}$ has a path going through all its $|M_n| - |D_{S^*}| = 2n - 1 - |D_{S^*}|$ nodes.*

Proof. We prove the result by induction on n. If $n = 1$, the result is trivial.

If $n > 1$, then G_n will have a left child L and a right child R. If G_n's root belong to S^*, then $D_{S^*} = G_n$, and the result is trivial. Otherwise, if L's root belongs to S^*, then $L \subseteq D_{S^*}$. By induction hypothesis, R has a path going trough all its nodes not contained in D_{S^*}. Since the MMG graph structure induces a total order in the nodes, we conclude this path must end in R's root. The result is obtained by adding the edge that joins R's root with G_n's root to our path.

If R's root belongs to S^*, the result is obtained in an analogous way. If none of L or R's roots belongs to S^*, then we apply induction hypothesis to get two paths ℓ_1, ℓ_2 going through the vertices of L and R not contained in D_{S^*}. By MMG construction, L's root is connected to every leaf in R. Additionally, we know that both paths must start in a certain leaf and finish in L and R's roots, respectively. Therefore, we will obtain the path we are searching for by joining the last element in ℓ_1 with the first element in ℓ_2, as well as joining the last element in ℓ_2 with G_n's root.

□

Lemma 26 ([CP18])**.** *For any set of nodes S in an MMG G_n, D_{S^*} contains $\frac{|D_{S^*}|+|S^*|}{2}$ many leaves.*

Proof. Let $S^* = \{v_1, \dots, v_k\}$. The minimality of S^* implies that $D_{v_i} \cap D_{v_j} = \emptyset$ for any $i \neq j$. Therefore, we can write

$$|\mathsf{leaves}(D_{S^*})| = \sum_{i=1}^{k} |\mathsf{leaves}(D_{v_i})|$$

Additionally, each binary tree D_{v_i} will have $(|D_{v_i}| + 1)/2$ many leaves, so

$$\sum_{i=1}^{k} |\mathsf{leaves}(D_{v_i})| = \sum_{i=1}^{k} \frac{|D_{v_i}| + 1}{2} = \frac{|D_{S^*}| + |S^*|}{2}$$

which gives us the desired result.

□

Lemma 27. *Let G_n be an MMG of n leaves that was computed making at most q queries to a hash function $H : \{0,1\}^* \to \{0,1\}^w$ that is instantiated in the random oracle model. The probability that either H's sequentiality or collision resistance are violated is upper bounded by $2q^2nw/2^w$.*

Proof. Lemmas 10 and 12 prove that G_n can be seen as an H-sequence of length $2n-1$. Additionally, Lemma 3 tells us that the probability to compute an H-sequence $x_0, \ldots, x_s$ of length s by making at most q queries to H of length Q in at most $s-1$ rounds is upper bounded by

$$q\frac{Q+\sum_{i=0}^{s}|x_i|}{2^w}$$

We observe that in an MMG no node has more than $\lceil \log_2(n) \rceil$ parents. Therefore, we can upper bound the former probability by

$$q\frac{q(2n-1)\lceil \log_2(n) \rceil w + q(2n-1)w}{2^w}$$

On the other hand, Lemma 2 tells us the probability to find a collision in H is upper bounded by

$$\frac{q^2}{2^{w+1}}$$

Finally, by using the union bound we state the probability of some of the two mentioned events happening is upper bounded by the sum of the two calculated probabilities. In turn this amount will be upper bounded by

$$\frac{2q^2nw}{2^w}$$

which gives us the desired result.

□

Now we can finally prove that the FlyClient protocol is indeed secure. In particular, we will prove that the inability of an adversary to fully compute a blockchain will cause an exponential decrease in its success probabilities. Our result will be instantiated by the protocol parameters t, δ. $t \in \mathbb{N}$ represents the number of queries made to the prover, while $\delta \in (0,1)$ represents the suffix of the chain that is manually checked.

Theorem 5 (FlyClient)**.** *Assume an adversary $\mathcal{A}$ forks an honest chain and creates a chain $\mathcal{C} = (B_n, G_{n-1})$ of $n+1$ blocks such that:*

- *Given some $\alpha \in [0,1]$, $\mathcal{A}$ performs at most $(1-\alpha)|V(G_{n-1})| = (1-\alpha)(2n-1)$ sequential queries to the hash function H.*
- *$\mathcal{C}$ was computed under the limitations of the (c, L)-assumption. This is, the section of the chain created after the fork will contain at most a c fraction of valid blocks.*

Then, if $\mathcal{A}$ commits $\mathcal{C}$ in the FlyClient protocol described in Figure 12 with parameters t, δ, it will get rejected with probability at least

$$1-(1-p)^t - \frac{2q^2nw}{2^w}$$

where $p = \max\{\alpha, \log_\delta(c)\}$.

Proof. The exponentially small $2q^2nw/2^w$ upper bounds the probability of some of the hash security properties being violated, as seen in Lemma 27.

Let $\mathcal{C} = (B_n, G_{n-1})$, where B_n is $\mathcal{C}$'s rightmost block and G_{n-1} is its associated MMG. When querying a block B, there are two possible events that could lead a verifier to reject $\mathcal{C}$:

- B is not a valid block. This is, its Proof of Work wasn't correctly solved.
- Some element in $\mathsf{path}(B, G_{n-1})$ (including B itself) wasn't consistently computed as indicated in Section 2.8.

Let I be the set of invalid blocks and S the set of nodes that were not consistently computed. The probability that a single queried block will be rejected by the prover is

$$\Pr[B \in I \cup D_{S^*}] = \Pr[B \in I] + \Pr[B \in D_{S^*}] - \Pr[B \in I \cap D_{S^*}] \geq \max(\Pr[B \in I], \Pr[B \in D_{S^*}]) \quad (2)$$

We start by bounding $\Pr[B \in I]$. In Section 3.1, we proved that for an optimal adversary the sampling distribution f will query an invalid block with probability $\log_\delta(c)$. Recall that, since f is increasing, an optimal adversary will place all the valid blocks it can mine at the end of the chain, placing invalid blocks between the forking point and the first valid mined block. An adversary could not follow this strategy, but this would imply having invalid blocks in positions that are more likely to be queried by f. Therefore, it holds $\Pr[B \in I] \geq \log_\delta(c)$.

Next, we need to bound $\Pr[B \in D_{S^*}]$. Lemma 25 tells us the subgraph $G_{n-1} \setminus D_{S^*}$ contains a path going through all its nodes. Since all the nodes in that path were consistently computed, that path constitutes an H-sequence of length $2n - 1 - |D_{S^*}|$. If $|D_{S^*}| \leq \alpha(2n-1)$, the length of our path will be equal or greater than $(1-\alpha)(2n-1)$, so we assume $|D_{S^*}| > \alpha(2n-1)$. On the other hand, Lemma 26 tells us

$$|\mathsf{leaves}(D_{S^*})| = \frac{|D_{S^*}| + |S^*|}{2} > \alpha n$$

We claim that $\Pr[B \in D_{S^*}] \geq \alpha$. In order to check this, we observe that if every leaf in G_n had the same chances of belonging to D_{S^*} and f was a uniform distribution, then the probability would be exactly $|\mathsf{leaves}(D_{S^*})|/n$.

However, G_{n-1} was the result of adding new leaves to an honestly computed MMG. This means that the blocks placed before the fork will have some consistently computed nodes above them, which cannot belong to S. Because of this, blocks placed before the fork will have a smaller probability of belonging to D_{S^*}. Additionally, f is an increasing distribution, so the blocks after the fork will be sampled with higher probability. We conclude from this that $\Pr[B \in D_{S^*}] \geq \alpha$. Combining the two obtained bounds with (2) we get

$$\Pr[B \in I \cup D_{S^*}] \geq \max\{\alpha, \log_\delta(c)\}$$

therefore, a queried block B will get accepted by the verifier with probability

$$\Pr[B \notin I \cup D_{S^*}] \leq 1 - \max\{\alpha, \log_\delta(c)\}$$

Finally, if the verifier makes a query $\{B_{\tau_1}, \ldots, B_{\tau_t}\}$, we have that the prover will get accepted if $\{B_{\tau_1}, \ldots, B_{\tau_t}\} \cap (I \cup D_{S^*}) = \emptyset$. This will happen with probability at most $(1 - \max\{\alpha, \log_\delta(c)\})^t$.

Combining this result with the bound obtained from Lemma 27, we get that the adversary will get rejected with probability at least

$$1 - (1 - \max\{\alpha, \log_\delta(c)\})^t - \frac{2q^2nw}{2^w}$$

as stated at the beginning.

□

Observation 9. *In the proof we ignored the extra queries a verifier should do to check the last δ fraction of the chain. However, since those extra queries would further reduce the success chances of an adversary, this doesn't compromise the soundness of our result.*

It is left to analyze the meaning of our security result. Figure 15 displays the probability for a malicious prover to get rejected after getting queried 20 times by a verifier, with respect to the "soundness gap" $p = \max\{\alpha, \log_\delta(c)\}$. This is, the maximum between the amount of sequential work that was left undone by the adversary and the probability to catch an invalid block with a single query. It can be seen the adversarial success probability becomes negligible with $p > 0.2$. It must be taken into account those results correspond to a small number of queries. A better result would be obtained by increasing the number of queries.

It is also interesting to analyze the relation between the (c, L)-Assumption parameters and the probability to catch an invalid block with a single query to the prover, which equals $\log_\delta(c)$. This relation is displayed in Figure 15, from which we can extract the following conclusions:

- The probability becomes higher the larger section δ we verify at the end of the chain. This means we will have to balance between better security guarantees and smaller proof weights.

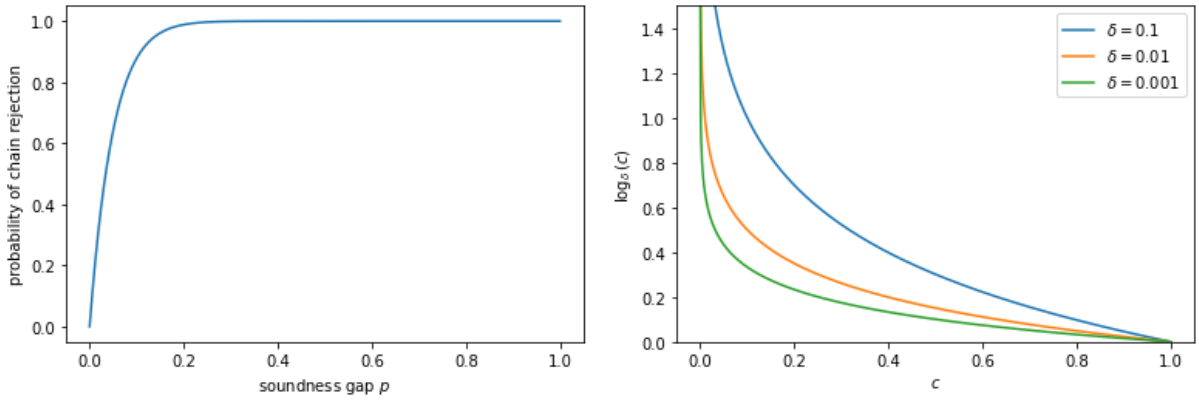

Figure 15: Left: Probability for a verifier to reject a prover with respect to a "soundness gap" $p = \max\{\alpha, \log_\delta(c)\}$ and $t = 400, w = 256, n = 10^8, q = 10^6$, as shown in Theorem 5. Right: Variation of the probability to query an invalid block with respect to the (c, L)-Assumption parameters.

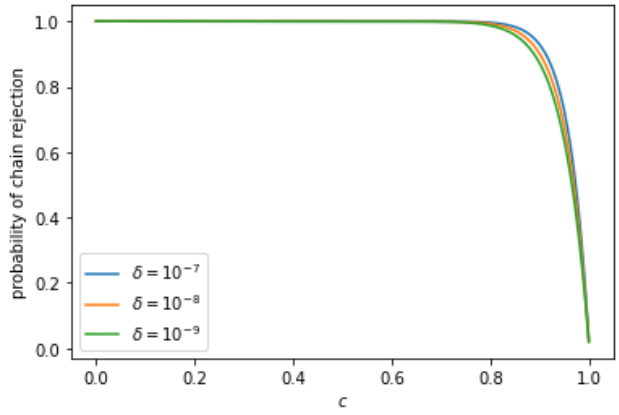

Figure 16: Probability for a malicious prover to get rejected with respect to the parameters of the (c, L)-Assumption.

- The probability to catch an invalid block is inversely proportional to the section of valid blocks c an adversary is able to create and reaches 0 when $c = 1$. This corresponds to the situation in which the adversary has managed to create a fully valid fork.
- $\log_\delta(c) \geq 1$ when $\delta \geq c$. This happens when the amount of valid blocks created after the forking point is smaller than the amount of blocks we will manually check at the end of the chain.

Finally, Figure 16 displays the relation between the (c, L)-Assumption parameters and the success probability of an adversary. In order to get more sound results, we have done $t = 400$ queries and checked a much smaller suffix of the chain. We observe that an adversary will get rejected with overwhelming probability when $c < 0.8$.

3.4. FlyClient proof size

In order to ensure protocol security, we want to choose parameters δ, t such that $(1 - \log_\delta(c))^t \leq 2^{-w}$, where t is the number of queries made in the protocol, δ is the suffix of the chain that is manually checked and w is the hash function size, which is in turn the protocol's security parameter.

In both cases, the choice of smaller parameters will mean reducing proof sizes at the expense of smaller

security guarantees. Therefore, we will seek to choose the smallest possible parameters such that the aforementioned property holds.

As for δ, [Bün+19] makes the verifier manually check the last L blocks of every received chain. This pairs well with the (c, L)-Assumption, according to which every fork longer than L blocks will contain some invalid blocks. Therefore, for a chain of length n, the verifier will manually check $\delta = L/n$ blocks at the end of the chain.

Now, it rests to find some t such that $(1 - \log_{\frac{L}{n}}(c))^t \leq 1/2^w$. This equals $t \cdot \log_{\frac{1}{2}}(1 - \log_{\frac{L}{n}}(c)) \geq w$. It follows from this that the number of queries made to the prover must satisfy

$$t \geq \frac{w}{\log_{\frac{1}{2}}(1 - \log_{\frac{L}{n}}(c))}$$

adding the L additional blocks thata re manually queried at the end of the chain, we get

$$t \geq L + \frac{w}{\log_{\frac{1}{2}}(1 - \log_{\frac{L}{n}}(c))}$$

3.5. The interactive FlyClient protocol in variable difficulty

We now extend the static difficulty protocol shown in Section 3.2 in order to support the extended blockchain model shown in Section 2.9. The main difference with the former protocol is that MMG nodes are now tuples containing information about difficulty weight, difficulty targets and timestamps, which introduces the necessity to do security checkings on those parameters.

In order to simplify checkings, [Bün+19] assumes an epoch length m that equals a power of 2. This ensures that an MMG node with less than m underlying blocks won't contain any difficulty transitions. Additionally, if a node has exactly m underlying blocks, those blocks will form an epoch. This will allow a verifier to do precise verifications on the weight of such nodes. Let $N = \langle r, x, t, T_S, T_N, w, ctr, n\rangle$. If $n < m$, then N's underlying leaves won't contain any difficulty transition. Therefore, it must hold $T_S = T_N$ and $w = n/T_S$ (since every underlying block will have difficulty $1/T_S$).

If $n = m$, then N's underlying blocks will form a full epoch. In this case, t will represent the time it took to compute the full epoch. The verifier will be able to recompute T_N using T_S and t (the bitcoin difficulty transition rules were shown in section 2.2). As in the previous case, it will hold $w = n/T_S$.

Finally, if $n > m$, some difficulty transition must have taken place in N's underlying blocks. Unlike the previous cases, the verifier won't be able to know the weights or the difficulty targets of the underlying nodes. Therefore, it will have to use the bounds we calculated in Section 2.9. Figure 17 displays the vnode algorithm, which implements the mentioned checkings.

We will use a slightly modified version of the sampling distribution. As before, we will use the distribution $f(x) = \frac{1}{(x-1)\ln(\delta)}$, where δ is the fraction of the chain we will manually check at the end of the chain. However, this time x will represent the aggregate difficulty at some point in the chain. This is, $x = \frac{1}{2}$ represents the point of the chain where half of the difficulty has been amassed and $f(\frac{1}{2})$ will represent the probability for the block at that point of the chain to be queried. This introduces the possibility for an adversary to lie a verifier about the queried blocks. This is, on a query $x \in [0, 1]$ corresponding to an invalid block, the adversary could send a valid block corresponding to a different point in the chain.

In order to defend against this, we need to prove that a verifier will be able to calculate the amassed weight of every received block. First, we will prove that it is easy to calculate the total weight of a chain if we have its last block and associated MMG.

Lemma 28. *Let $\mathcal{C} = B_0, \ldots, B_n$ be a variable difficulty chain and $R = \mathsf{root}(\mathsf{MMG}(B_0, \ldots, B_{n-1}))$ the root of its associated MMG. If we write $R = \langle r_{n-1}, \perp, t', T'_S, T'_N, w', \perp, l'\rangle$ and $B_n = \langle R_{n-1}, x_n, t_n, T_S, T_N, w_n, ctr, 1\rangle$, then $\mathcal{C}$'s cumulative weight equals $w' + w_n$.*

Proof. According to MMG definition, w' equals the cumulative weight of $B_0, \ldots, B_{n-1}$. It follows from this that $\mathcal{C}$'s cumulative weight will be obtained from summing w' to B_n's weight.

□

Algorithm $\mathsf{vnode}(N)$

$//N$ is an MMG node
parse $B = \langle r, x, t, T_S, T_N, w, ctr, l\rangle$
if $l < m$ do
    (-) $b_1 = 1$ iff $T_S = T_N$
    (-) $b_2 = 1$ iff $w = l/T_S$
    (-) return $b_1 \wedge b_2$
else if $l = m$ do
    (-) $b_1 = 1$ iff $T_N = \mathsf{difftrans}(T_S, t)$//calculates new target using difficulty transition rules
    (-) $b_2 = 1$ iff $w = l/T_S$
    (-) return $b_1 \wedge b_2$
else do
    (-) $b_1 = 1$ iff $T_N \leq \tau^{\lfloor \frac{l}{m} \rfloor} \cdot T_S$ and $T_N \geq (\frac{1}{\tau})^{\lfloor \frac{l}{m} \rfloor} \cdot T_S$//see Lemma 13
    (-) $b_2 = 1$ iff $w^L \leq w \leq w^U$//w^L, w^U are bounds as calculated in Lemma 15
    (-) return $b_1 \wedge b_2$

Figure 17: Implementation of the vnode algorithm. Given an MMG node N, vnode determines wether N's attributes are feasible or not. $m \in \mathbb{N}$ represents an epoch's length. The difftrans function, which updates difficulty targets at the end of each epoch, is left unspecified since it depends in the blockchain's difficulty transition rules.

Algorithm $\mathsf{getWeight}(B, R)$

$//B$ is a chain's last block, R is a chain's associated MMG root
parse $B = \langle r, x, t, T_S, T_N, w, ctr, l\rangle$
parse $R = \langle r', x', t', T'_S, T'_N, w', ctr', l'\rangle$
return $w + w'$

Figure 18: Implementation of the getWeight algorithm. getWeight calculates the cumulative weight of a given chain with last block B and associated MMG root R.

Using this result, we can easily implement the getWeight algorithm to calculate the cumulative weight of a chain using its last block and associated MMG. This algorithm is depicted in Figure 18.

Now we will prove it is possible to calculate the amassed weight of a block by using its MMG parent set. We will take advantage on the previously obtained results on the MMG graph structure.

Lemma 29. *Let $\mathcal{C} = B_0, \ldots, B_n$ be a variable difficulty blockchain with associated MMG G_n and $B_i \in \mathcal{C}$. For every $x \in \mathsf{parents}(B_i, G_{n-1})$, let w_x be x's weight and w_i be B_i's weight. Then, the amassed weight at B_i can be calculated as*

$$\sum_{x \in \mathsf{parents}(B_i, G_{n-1})} w_x + w_i$$

Proof. Lemma 11 tells us that $R = \mathsf{root}(\mathsf{MMG}(B_0, \ldots, B_{i-1}))$ is obtained by hashing the elements in $\mathsf{parents}(B_i, G_{n-1})$ in reverse order. Let w be R's weight. By variable difficulty MMG definition, w equals the sum of the weights of B_0 to B_{i-1}. Additionally, by using the node combination rules described in Section 2.9, we get that

$$w = \sum_{x \in \mathsf{parents}(B_i, G_{n-1})} w_x$$

with w_x defined as before. It follos fron this that the amassed weight from B_0 to B_i equals

$$\sum_{x \in \mathsf{parents}(B_i, G_{n-1})} w_x + w_i$$

as stated.

□

Taking advantage of this, a verifier will be able to check wether a certain block is placed at the expected point of the chain by using the vindex algorithm, which is depicted in Figure 19.

Algorithm $\mathsf{vindex}(i, B, p, w_t)$

```
//B is a block, i is B's index, p is B's parent set, w_t is a chain's cumulative weight
parse B = ⟨r, x, t, T_S, T_N, w, ctr, l⟩
w_c ← w
for n in p:
    parse n = ⟨r', x', t', T'_S, T'_N, w', ctr', l'⟩
    w_c ← w_c + w'
if w_c/w_t = i then:
    return 1
else:
    return 0
```

Figure 19: Implementation of the vindex algorithm. vindex checks wether i is the correct index of a block B with parent set p. w_c represents the cumulative weight of the chain B belongs to.

The extended version of the FlyClient protocol is displayed in Figure 20. The new protocol works as follows:

1. The prover, holding a chain $\mathcal{C} = (B_n, G_{n-1})$ sends B_n and $\mathsf{root}(G_{n-1})$ to the verifier, where B_n and G_{n-1} are $\mathcal{C}$'s head and associated MMG, respectively.

2. On input $(B_n, R_{n-1)}$, the verifier checks the following:

   - R_{n-1}'s hash is contained in B_n.
   - B_n's parameters are feasible with respect to MMG design rules.
   - B_n's Proof of Work is valid.

   Additionally, the verifier generates a sampling of cumulative weights $o_1, \ldots, o_t)$ according to the sampling distribution f and will send it to the prover.

3. For every $i \in \{1, \ldots, t\}$, the verifier will do the following:

   - Use the algorithm getIndex to obtain the block index τ_i that corresponds to the cumulative weight o_i. The getIndex algorithm is displayed in Figure 21.
   - Obtain block B_{τ_i}, together with its Merkle Proof opening and MMG parent set.

   Moreover, the prover will use the suffixLength algorithm to obtain the length d of the chain suffix that represents a δ fraction of the total weight. Then, it will obtain the last d blocks in $\mathcal{C}$, together with their Merkle Proofs and MMG parent sets.

   Finally, the prover will send the verifier all the obtained blocks, together with their Merkle Proofs and parent sets.

4. On input tuples $(\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))_{i=1}^{t+d}$, for every $i \in \{1, \ldots, t+d\}$, the verifier will:

   - Check that $\mathsf{parents}(B_{\tau_i}, G_{n-1})) \subseteq \mathsf{mp}(B_{\tau_i}, G_{n-1})$.
   - Check that B_{τ_i}'s Proof of Work is valid.
   - Check that B_{τ_i}'s are feasible with respect to MMG design rules.

- Apply algorithm vmp' to verify B_{τ_i}'s proof. Algorithm vmp' is a modification of the previously shown vmp algorithm that supports variable difficulty MMG nodes. It is shown in Figure 24.
- Apply algorithm vroot' to verify the MMG root stored in B_{τ_i}'s is correctly computed. Algorithm vroot' is a modification of the previously shown vroot algorithm that supports variable difficulty MMG nodes. It is shown in Figure 25.
- For every $i \in \{1, \ldots, t\}$, the vindex algorithm will be applied to ensure that every queried block B_{τ_i} corresponds to the amassed weight o_i. The implementation of the vindex algorithm is shown in Figure 19.

Finally, the verifier will use the vsuffix algorithm to check that the last d received blocks represent a δ fraction of the cumulative weight of the chain. The vsuffix algorithm implementation is shown in Figure 23.

Prover $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1)$

Stage P_0: On input $\mathcal{C}$:

1. parse $\mathcal{C} = (B_n, G_{n-1})$
2. get $R_{n-1} := \mathsf{root}(G_{n-1})$
3. send (B_n, R_{n-1}) to $\mathcal{V}_0$

Stage P_1: On input $o = (o_i)_{i=1}^t$:

1. $w_t \leftarrow \mathsf{getWeight}(B_n, R_{n-1})$
2. $\forall i \in \{1, \ldots, t\}$ do
   (-) $\tau_i \leftarrow \mathsf{getIndex}(o_i, G_{n-1}, w_t)$
   (-) $y_i := (\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))$
3. $d \leftarrow \mathsf{suffixLength}(\mathcal{C}, \delta)$
4. $\forall i \in \{1, \ldots, d\}$ do
   $y_{t+i} := (n - d + i, B_{n-d+i}, \mathsf{mp}(B_{n-d+i}, G_{n-1}), \mathsf{parents}(B_{n-d+i}, G_{n-1}))$
5. send $y := (y_1, \ldots, y_{t+d})$ to $\mathcal{V}_1$

Verifier $\mathcal{V} = (\mathcal{V}_0, \mathcal{V}_1)$

Stage V_0: On input (B_n, R_{n-1}):

1. parse $B_n = \langle r_{n-1}, x, t, T_S, T_N, w, ctr, l \rangle$
2. $b_0 := 1$ iff (-) $r_{n-1} = H(R_{n-1})$
   (-) $l = 1$ and $w = 1/T_S$
   (-) $H(ctr, G(r_{n-1}, x, t, T_S, T_N)) < T_S$
3. $\forall i \in \{1, \ldots, t\}$ sample o_i according to $f(\delta)$
4. send $o := (o_1, \ldots, o_t)$ to $\mathcal{P}_1$

Stage V_1: On input $y = (\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))_{i=1}^{t+d}$:

1. $w_t \leftarrow \mathsf{getWeight}(B_n, R_{n-1})$
2. $\forall i \in \{1, \ldots, t+d\}$ do
   (a) parse $B_{\tau_i} = \langle r_{\tau_{i-1}}, x, t, T_S, T_N, w, ctr, l \rangle$
   (b) $b_i^{(1)} := 1$ iff
   (-) $\mathsf{parents}(B_{\tau_i}, G_{n-1}) \subseteq \mathsf{mp}(B_{\tau_i}, G_{n-1})$
   (-) $H(ctr, G(r_{\tau_{i-1}}, x, t, T_S, T_N, w)) < T_S$
   (-) $l = 1$ and $w = 1/T_S$
   (-) $\mathsf{vmp'}(B_{\tau_i}, \tau_i, r_{n-1}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1})) = 1$
   (-) $\mathsf{vroot'}(r_{\tau_{i-1}}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1})) = 1$
3. $\forall i \in \{1, \ldots, t\}$ do
   $b_i^{(2)} = 1$ iff
   $\mathsf{vindex}(o_i, B_{\tau_i}, \mathsf{parents}(B_{\tau_i}, G_{n-1}), w_t) = 1$
4. $b_s = 1$ iff $\mathsf{vsuffix}(\delta, w_t, ((B_{t+1}, \ldots, B_{t+d})) = 1$
5. output $b_0 \wedge b_s \wedge (\bigwedge_{i=1}^{t+d} b_i^{(1)} \wedge b_i^{(2)})$

Figure 20: Description of the FlyClient protocol. $\mathcal{C}$ represents a FlyClient blockchain. $t \in \mathbb{N}, \delta \in (0, 1]$ are parameters of the protocol. t represents the number of queries made to the prover. δ represents the fraction of blocks in the end of the chain that are manually checked.

Note the block difficulty target is no longer considered a parameter. This is because it no longer is constant and needs to be recalculated while computing a chain. Difficulty targets will be included in every block header.

We now have to prove that an adversary won't get any advantage from manipulating the difficulty transitions. As seen in Section 2.10, the (c, L)-Assumption limits adversaries who respects the difficulty transitions. Therefore, an adversary could get a higher rate of valid blocks by artificially rising the difficulty targets and mining heavier blocks. However, it is easy to see such a strategy won't give an adversary a significative advantage.

Algorithm $\mathsf{getIndex}(o, G, w)$

```
//o ∈ [0,1], w ∈ ℝ, G is an MMG
W ← o · w
r ← root(G)
parse r = ⟨h, x, t, T_S, T_N, w, ctr, l⟩
if l = 1 then
    return 0
r' ← root(lc(G))
parse r' = ⟨h', x', t', T'_S, T'_N, w', ctr', l'⟩
if w' ≤ W then
    return getIndex(o, lc(G))
else
    return l' + getIndex(o − w', rc(G))
```

Figure 21: Implementation of the $\mathsf{getIndex}$ algorithm. Given a weight $o \in [0,1]$ and an MMG G, $\mathsf{getIndex}$ returns the block index that amasses weight o in G. w represents the cumulative weight of a chain.

Algorithm $\mathsf{suffixLength}(\mathcal{C}, \delta)$

```
//C is a blockchain, δ ∈ [0,1]
parse C = (B_n, G_{n−1})
w_t ← getWeight(B_n, G_{n−1})
d ← 1, w_s ← 0
while w_s/w_t < δ do
    parse C[−d] = ⟨r, x, t, T_S, T_N, w, ctr, l⟩
    w_s ← w_s + w
    d ← d + 1
return d
```

Figure 22: Implementation of the $\mathsf{suffixLength}$ algorithm. $\mathsf{suffixLength}$ calculates the suffix length that represents a δ fraction of a chain $\mathcal{C}$.

Algorithm $\mathsf{vsuffix}(\delta, w_t, L)$

```
//δ, w_t ∈ [0,1], L is a sequence of blocks
w_s ← 0
for B ∈ L do
    parse B = ⟨r, x, t, T_S, T_N, w, ctr, l⟩
    w_s ← w_s + w
if w_s/w_t ≥ δ do
    return 1
else do
    return 0
return d
```

Figure 23: Implementation of the $\mathsf{vsuffix}$ algorithm. On input $\delta \in [0,1]$, a chain's cumulative weight w_t and a list of blocks L, $\mathsf{vsuffix}$ determines wether L's total weight represents at least a δ fraction of w_t.

Algorithm $\mathsf{vmp}'(B, i, r, mp)$

```
//B is a block, i is B's index, r is an MMG root, mp is a Merkle Proof
y ← B, k ← i
for n in mp:
    parse y = ⟨r, x, t, T_S, T_N, w, ctr, l⟩
    parse n = ⟨r', x', t', T'_S, T'_N, w', ctr', l'⟩
    b_n^(1) = 1 iff vnode(n) = 1
    if k mod 2 = 0 then:
        b_n^(2) = 1 iff T_N = T'_S
        y ← ⟨H(y, n), ⊥, t + t', T_S, T'_N, w + w', ⊥, l + l'⟩
    else:
        b_n^(2) = 1 iff T'_N = T_S
        y ← ⟨H(n, y), ⊥, t + t', T'_S, T_N, w + w', ⊥, l + l'⟩
    k ← ⌊k/2⌋, i ← ⌊i/2⌋
if y = r then:
    return ⋀_{n∈mp} b_n^(1) ∧ b_n^(2)
else:
    return 0
```

Figure 24: Implementation of the vmp' algorithm. vmp checks wether mp is a valid Merkle Proof for a block B with index i in a variable difficulty MMG with root r.

Lemma 30. *Let $\mathcal{A}$ be an adversary that produces a chain $\mathcal{C}$ with probability p such that valid blocks represent a fraction $w \in [0, 1]$ of the total chain weight. Then, there exists an adversary $\mathcal{A}'$ that respects the retargeting rules and produces a chain $\mathcal{C}'$ with the same rate of valid blocks with probability $p - \mathsf{negl}(w)$.*

Proof. Let $\mathcal{C} = (B_n, G_{n-1})$ be the chain produced by $\mathcal{A}$. Let $B_i \in \mathcal{C}$ such that $B_i \to B_{i+1}$ contains an invalid difficulty transition. This means that the blocks $B_{i-m}, \ldots, B_i$ form a full epoch which difficulty transition wasn't correctly computed. Let $R = \mathsf{root}(\mathsf{MMG}(B_{i-m}, \ldots, B_i))$, then R will contain the epoch's difficulty target, B_{i+1}'s difficulty target and the timestamp that represents the time it took to compute the full epoch. Therefore, if a verifier makes a call to $\mathsf{vnode}(R)$ it will reject $\mathcal{C}$ (check Figure 17 to see how the vnode algorithm is implemented).

Let $j \in \{i-m, \ldots, i\}$. Then, it holds that $R \in \mathsf{path}(B_j, G_{n-1})$. Therefore, if B_j is queried by the verifier, it will make a call to $\mathsf{vnode}(R)$ when opening the Merkle Proof of B_j (check Figure 24 to see how the vmp' algorithm is implemented). Therefore, B_j will be rejected by a verifier. We conclude every block in the epoch will be rejected by the verifier.

Let $\mathcal{A}'$ be the adversary that differs from $\mathcal{A}$ in manipulating the timestamps of $B_{i-m}, \ldots, B_i$ so that the difficulty transition $B_i \to B_{i+1}$ is correct. Let $\mathcal{C}'$ be the chain created by $\mathcal{A}'$. Since all the modified blocks were already invalid in $\mathcal{C}$ it won't have less valid blocks than $\mathcal{C}$.

□

Another possible adversarial strategy takes advantage on the imprecision of the weight bounds used in the vnode algorithm. If we observe the bounds calculated in Lemma 15, we can see there is a huge difference between the upper and lower bounds. Therefore, an adversary could maliciously increase the weight of a block to inflate the cumulative weight of its chain. This way, an adversary could make a verifier query a malicious chain on top of the honest chains that have more cumulative weight. However, it is easy to see that manipulating the weight of a block implies invalidating the other blocks in its epoch.

Lemma 31. *Let $\mathcal{C} = (B_n, G_{n-1})$ be a blockchain and $B \in \mathcal{C}$ a block which weight has been maliciously manipulated. Then, every block belonging to the same epoch as B will be rejected by the verifier.*

Proof. Let $B_i, \ldots, B_{i+m}$ be the set of blocks that form the epoch B belongs to and $j \in \{i, \cdots i+m\}$. Similarly to the previous proof, $R = \mathsf{root}(\mathsf{MMG}(B_i, \ldots, B_{i+m}))$ contains the sum of the weights of all the blocks in the

Algorithm $\mathsf{vroot}'(r, p)$

```
//r in an MMG root, p is the parent set of an MMG block
y ← H'_r(p)
if y = r then:
    return 1
else:
    return 0

def H'_r(p):
    if |p| = 1:
        return p[0]
    else:
        return H(p[0], H_r(p[1..]))
```

Figure 25: Implementation of the vroot' algorithm. Given a block B with parent set p and stored root r, vroot checks wether r was correctly computed from p.

epoch. Also, it will hold $R \in \mathsf{path}(B_j, G_{n-1})$, so the verifier will call $\mathsf{vnode}(R)$ when opening B_j's Merkle Proof. Let T_S be the epoch's difficulty target. By implementation of the vnode algorithm, R will only be accepted if its weight equals m/T_S. However, since B's weight was altered, this won't be possible.

□

3.6. Non-interactive FlyClient protocol

Finally, we will use the Fiat-Shamir Heuristic[FS86] to make the protocol shown in Figure 20 non-interactive. This will allow verifiers to check a received proof without any further interaction with the prover. Additionally, proofs will be transferable, *i.e,* a single generated proof will be usable by many verifiers.

To do so, the block sampling will be chosen according to a hash of the chain's last block, instead of a sampling distribution. An adversary holding a chain might try to manipulate the sampling by modifying the last block in the chain. However, this would imply solving a new Proof of Work. As we said in Section 2.10, the adversarial ability to produce Proofs of Work is assumed to be limited.

Figure 26 displays the non-interactive FlyClient protocol. A prover, holding a chain $\mathcal{C} = (B_n, G_{n-1})$ will:

1. Extract $B_n = \mathsf{head}(\mathcal{C})$ and $R_{n-1} = \mathsf{root}(G_{n-1})$.
2. Obtain a sampling of amassed weights $o = (o_1, \ldots, o_t)$ from a hash of B_n.
3. For all $i \in \{1, \ldots, t\}$, the prover will
   - Use the getIndex algorithm to obtain the block index τ_i to which corresponds the amassed weight o_i. The getIndex algorithm is shown in Figure 21.
   - Obtain B_{τ_i}'s Merkle Proof and MMG parent set.
4. Use the suffixLength algorithm to calculate the suffix length d of the chain that represents a δ fraction of the total chain weight. The suffixLength algorithm is shown in Figure 22.
5. Get the last d blocks in $\mathcal{C}$, together with their respective Merkle Proofs and MMG parent sets.
6. Send B_n, R_{n-1}, o and all the sampled blocks, together with their Merkle Proofs and parent sets, to the verifier.

On input $(R_{n-1}, B_n, o, (\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))_{i=1}^{t+d})$, the verifier will

1. With respect to B_n, the following checks will be done:
   - R_{n-1} is stored in B_n's header.

- B_n's parameters are feasible with respect to MMG design rules.
- B_n's Proof of Work is valid.

2. o is correctly computed from a hash of B_n.
3. For all $i \in \{1, \ldots, t+d\}$, the following checks will be done:
   - B_{τ_i}'s MMG parent set is contained into its Merkle Proof.
   - B_{τ_i}'s Proof of Work is valid.
   - B_{τ_i}'s parameters are feasible with respect to MMG design rules.
   - Apply algorithm vmp' to verify B_{τ_i}'s proof. Algorithm vmp' is shown in Figure 24.
   - Apply algorithm vroot' to verify the MMG root stored in B_{τ_i}'s is correctly computed. Algorithm vroot' is shown in Figure 25.
4. For every $i \in \{1, \ldots, t\}$, the vindex algorithm will be applied to ensure that every queried block B_{τ_i} corresponds to the amassed weight o_i. The implementation of the vindex algorithm is shown in Figure 19.
5. The vsuffix algorithm will be applied to check the last d blocks in the chain represents a δ fraction of the total chain weight.

Prover $\mathcal{P}$: On input $\mathcal{C}$:
1. parse $\mathcal{C} = (B_n, G_{n-1})$
2. get $R_{n-1} := \mathsf{root}(G_{n-1})$
3. get $o := (o_1, \ldots, o_t) \leftarrow H(B_n)$
4. $\forall i \in \{1, \ldots, t\}$ do
   (-) $\tau_i \leftarrow \mathsf{getIndex}(o_i, G_{n-1})$
   (-) $y_i := (\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))$
5. $d \leftarrow \mathsf{suffixLength}(\mathcal{C}, \delta)$
6. $\forall i \in \{1, \ldots, d\}$ do
   $y_{t+i} := (n-d+i, B_{n-d+i}, \mathsf{mp}(B_{n-d+i}, G_{n-1}), \mathsf{parents}(B_{n-d+i}, G_{n-1}))$
7. send $(R_{n-1}, B_n, o, (y_1, \ldots, y_{t+d}))$ to V

Verifier $\mathcal{V}$: On input $(R_{n-1}, B_n, o, (\tau_i, B_{\tau_i}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1}))_{i=1}^{t+d})$:
1. parse $B_n = \langle r_{n-1}, x, t, T_S, T_N, w, ctr, l \rangle$
2. $b_0 := 1$ iff (-) $r_{n-1} = H(R_{n-1})$
   (-) $l = 1$ and $w = 1/T_S$
   (-) $H(ctr, G(r_{n-1}, x, t, T_S, T_N)) < T_S$
   (-) $o = H(B_n)$
3. $w_t \leftarrow \mathsf{getWeight}(B_n, R_{n-1})$
4. $\forall i \in \{1, \ldots, t+d\}$ do
   (a) parse $B_{\tau_i} = \langle r_{\tau_{i-1}}, x, t, T_S, T_N, w, ctr, l \rangle$
   (b) $b_i^{(1)} := 1$ iff
   (-) $\mathsf{parents}(B_{\tau_i}, G_{n-1}) \subseteq \mathsf{mp}(B_{\tau_i}, G_{n-1})$
   (-) $H(ctr, G(r_{\tau_{i-1}}, x, t, T_S, T_N, w)) < T_S$
   (-) $l = 1$ and $w = 1/T_S$
   (-) $\mathsf{vmp'}(B_{\tau_i}, \tau_i, r_{n-1}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1})) = 1$
   (-) $\mathsf{vroot'}(r_{\tau_{i-1}}, \mathsf{mp}(B_{\tau_i}, G_{n-1}), \mathsf{parents}(B_{\tau_i}, G_{n-1})) = 1$
5. $\forall i \in \{1, \ldots, t\}$ do
   $b_i^{(2)} = 1$ iff
   $\mathsf{vindex}(o_i, B_{\tau_i}, \mathsf{parents}(B_{\tau_i}, G_{n-1}), w_t) = 1$
6. $b_s = 1$ iff $\mathsf{vsuffix}(\delta, w_t, ((B_{t+1}, \ldots, B_{t+d})) = 1$
7. output $b_0 \wedge b_s \wedge (\bigwedge_{i=1}^{t+d} b_i^{(1)} \wedge b_i^{(2)})$

Figure 26: Description of the non-interactive FlyClient protocol. $\mathcal{C}$ represents a FlyClient blockchain. $t \in \mathbb{N}, \delta \in (0, 1]$ are parameters of the protocol. t represents the number of queries made to the prover. δ represents the fraction of blocks in the end of the chain that are manually checked.

4. Alternative solutions

In this section, we will review the main alternative light-client blockchain protocols in existence: SPV, SNACKs and NIPoPoWs. Additionally, we will compare them with the FlyClient protocol, mainly in terms of soundness and succinctness guarantees.

When it comes to SPV, comparison is largely predictable: SPV offers stronger security guarantees at the expense of non-succinct proofs. As for the SNACKs, we have concluded that both protocols offer almost identical guarantees, with SNACKs providing a more efficient blockchain model. Finally, we have found some advantages of the FlyClient protocol over NIPoPoWs.

4.1. SPV

Simplified Payment Verification (SPV, from now on) was the first proposed blockchain light-client protocol. Indeed, it was mentioned in the Bitcoin white paper [Nak08]. SPV operates in a simple manner: instead of sending the whole blockchain, a prover will only send the blocks' headers. Then, a verifier will have to check that all the blocks are chained up together and that every Proof of Work is valid. The protocol works as follows:

- The prover, who is holding a chain $\mathcal{C} = B_0, \ldots, B_n$ sends the headers of all the blocks to the verifier.
- The verifier, receiving the set of headers $(\mathsf{header}(B_i))_{i=0}^n$ checks the following:
  - B_0 equals the expected genesis block.
  - For all $B_i = \langle h_i, x_i, ctr_i \rangle, i \in \{1, \ldots, n\}$, it is checked that $h_i = H(B_{i-1})$.
  - For all $B_i = \langle h_i, x_i, ctr_i \rangle, i \in \{1, \ldots, n\}$, it is checked that $H(ctr, G(x_i, h_i)) < T$, where T is the corresponding difficulty target.

Prover P: On input $\mathcal{C}$	Verifier V(ψ): On input $\pi = (\pi_i)_{i=0}^n$
parse $\mathcal{C} = B_0, \ldots, B_n$	$b_0 := 1$ iff $\pi_0 = \psi$
for $i = 0$ to n:	for $i = 1$ to n:
$\quad \pi_i \leftarrow \mathsf{header}(B_i)$	$\quad$ parse $\pi_i := \mathsf{header}(B_i) = \langle h_i, x_i, ctr_i \rangle$
send $\pi := (\pi_i)_{i=0}^n$ to V	$\quad b_i^0 := 1$ iff $(h_i = H(\pi_{i-1}))$
	$\quad b_i^1 := 1$ iff $H(ctr_i, G(h_i, ctr_i)) < T$
	output $b_0 \wedge \bigwedge_{i=1}^n b_i^0 \wedge b_i^1$

Figure 27: Description of the SPV protocol.

When connecting a set of provers, a verifier will query them by descending order of chain length. A verifier will only accept a chain if it's fully valid. Therefore, ensuring the protocol is secure equals ensuring the longest valid chain in the network is held by an honest party. We formalize this notion here:

Assumption 3 (SPV assumption)**.** *The chain with the most PoW solutions follows the rules of the network and will eventually be accepted by the majority of miners.*

This assumption can be inferred from the security properties of the Bitcoin backbone protocol. Therefore, the SPV can be considered a fully secure protocol. This provides more sound guarantees than those of the FlyClient protocol, which depends on the unproven (c, L)-Assumption. However, SPV is not a succinct protocol. Since the verifier has to download every block header in the chain, proof sizes will always be linear in the chain length.

4.2. Comparison between FlyClient and SNACKs

The Succinct Non-Interactive Arguments of Chain Knowledge were introduced in [Abu+22] by Abusalah *et.al.* in 2022. After defining a SNACK primitive, it is shown how to build a SNACK on top of Graph Labeling

Proofs of Sequential Work (GL-PoSW), a particular type of PoSW. Additionally, concrete instantiations are provided based on skiplists and CP-based Directed Acyclic Graphs ($DAGs$). Blockchains are modeled as $DAGs$, in which block dependencies are represented as edges. The security guarantees rely on the adversarial inability to compute a heavy path. Similarly to the FlyClient protocol, the (c, L)-Assumption is used to formalize the adversarial limitations. A more in-depth review of the SNACK protocol is provided in Appendix A.1.

We will proceed to compare the FlyClient protocol with SNACKs. As we said before, both protocols have many similarities. Both protocols' security guarantees rely on the assumed adversarial inability to sequentially generate a certain amount of valid Proofs of Work. In order to catch a malicious chain with a sublinear amount of block queries, both protocols introduce a more complex blockchain model in which every block contains a commitment to all its preceding blocks, instead of the traditional blockchain models in which every block contains a commitment to its preceding block.

Remarkably, both protocols rely on the (c, L)-Assumption to achieve security guarantees. This captures the mentioned adversarial limitation: an adversary forking an honest chain won't be able to produce more than a c rate of valid blocks after the forking point. However, each protocol uses a slightly different version of the assumption. While the assumption we exposed in Section 2.10 limits the rate of valid blocks in the whole fork, [Abu+22] limits the rate of valid blocks in a single blockchain path.

However, there is a remarkable difference in the way each protocol is implemented. [Abu+22] creates a cryptographic primitive, then instantiates it with a GL-PoSW protocol, a Vector Commitment scheme and a graph model that captures blockchain dependencies. On the other hand, FlyClient creates an *ad-hoc* protocol based on Merkle Trees and a suitable sampling distribution, as we have explained throughout this work. It must be noted that the blockchain model used in the FlyClient protocol introduces the blocks in the leaves of a Merkle Tree, forcing full-node clients to store additional nodes not containing PoW solutions or transaction data. On the other hand, the two SNACK blockchain constructions don't need any additional nodes on top of those representing blocks.

We will now compare the security guarantees and efficiency of each protocol:

Security guarantees. As we saw in Theorem 5, an adversary submitting a malicious chain in the FlyClient protocol will get rejected with probability at least

$$1 - (1 - p)^t - \frac{2q^2nw}{2^w}$$

where $p = \max\{\alpha, \log_\delta(c)\}$. In order to make the comparison easier, we will assume $\log_\delta(c) \geq \alpha$ and $\delta = L/n$, so we will get

$$1 - (1 - \log_{\frac{L}{n}}(c))^t - \frac{2q^2nw}{2^w}$$

On the other hand, Corollary 1 tells us the SNACK protocol displayed in Figure 29 is an (α_{n-L}, ϵ)-knowledge-sound protocol, with

$$\alpha_{n-L} = 1 - \left(\log_c\left(\frac{L-1}{n}\right)\right)^{-1}, \epsilon = \alpha_{n-L}^t + \frac{3 \cdot q^2}{2^w}$$

Additionally, if a verifier bootstraps a chain using the protocol displayed in Figure 32, it will securely bootstrap with probability $\epsilon + \epsilon_C + \epsilon_F$, where ϵ_C relates to the security of the used Vector Commitment scheme and ϵ_F is an upper bound over the adversarial probability to generate a (c, L)-fork. This means that an adversary submitting a chain in the bootstrapping protocol will get rejected with probability at least $1 - (\epsilon + \epsilon_C + \epsilon_F)$. In order to compare this bound with the FlyClient one, we will ignore the ϵ_F and ϵ_C bounds. This won't make a big difference, since we are assuming the underlying Vector Commitment scheme is secure and our FlyClient theorem assumes the adversary works under the (c, L)-Assumption restrictions. Therefore, we will compare the functions

$$1 - (1 - \log_{\frac{L}{n}}(c))^t - \frac{2q^2nw}{2^w} \text{ and } 1 - \left(1 - \left(\log_c\left(\frac{L-1}{n}\right)\right)^{-1}\right)^t - \frac{3 \cdot q^2}{2^w}$$

It is easy to see those two functions will take very similar values. Indeed, it holds

$$\log_{\frac{L}{n}}(c) = \frac{\log_c(c)}{\log_c(\frac{L}{n})} = \frac{1}{\log_c(\frac{L}{n})}$$

Figure 28 displays the security guarantees of both protocols with respect to c. We have assumed $\frac{L}{n} \approx 10^{-8}$ and challenge sizes $t = 400$ and $t = 800$. As we can see, both protocols have identical security guarantees.

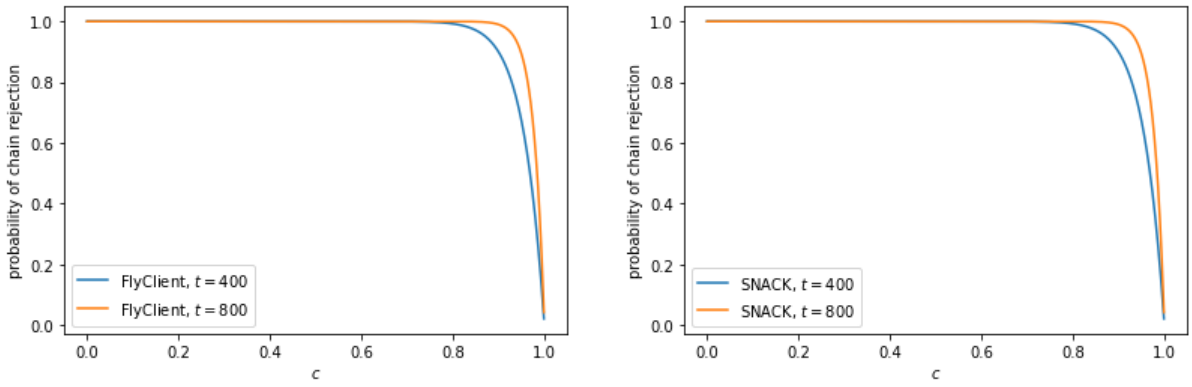

Figure 28: Left: FlyClient security guarantees. Right: SNACK security guarantees. t represents the number of challenges.

Protocol efficiency. Both protocols seek to keep proof sizes sublinear (*i.e.* polylogarithmic) in the chain size. This size will be given by the number of challenges a verifier will send the prover.

In order to defend against forks shorter than L blocks, both protocols manually check the last L blocks in the chain. After that, both protocols sample t additional blocks according to a given sampling distribution. The amount t of sampled blocks will be chosen in a way the success probabilities of any adversary are bounded by 2^{-w}. In the case of the FlyClient protocol, the number of challenges is

$$t = L + \frac{w}{\log_{\frac{1}{2}}(1 - (\log_c(L/n)^{-1})}$$

In the case of SNACKs, we have

$$t = L + \frac{w}{\log_{\frac{1}{2}}(1 - (\log_c(\frac{L-1}{m+L}))^{-1})}$$

Both amounts are very similar, so there won't be a significative difference in the amount of challenges a verifier is required to check. However, the different implementation of the protocols will provoke a significant difference in favor of the SNACK protocol: Since the FlyClient blockchain model places blocks in a Merkle Tree's leaves, it will require full-node clients to store many additional nodes. In fact, Lemma 5 tells us an MMR on n leaves will have $2n - 1$ nodes. This is, in order to store a chain of length n, a full-node will have to store $2n - 1$ nodes. On the other hand, the two SNACK blockchain constructions that are shown in [Abu+22] store a block in every node, meaning that storing a SNACK blockchain will require to store exactly n nodes.

In conclusion, both protocols are fairly similar, to the point of having identical security guarantees. However, SNACKs seem to offer some advantage in the form of more efficient blockchains, saving some space for full node miners. SNACKs are also more versatile, being able to be instantiated with different blockchain models and GL-PoSW schemes.

4.3. Comparison between FlyClient and NIPoPoWs

The Non-Interactive Proofs of Proof-of-Work (NIPoPoWs, from now on), were introduced by Kiayias *et.al.* in [KMZ17]. To the extent of our knowledge, it was the first proposed blockchain light-client protocol

with a sublinear proof size. Unlike the FlyClient and SNACK protocols, NIPoPoWs are based on the concept of superblocks, *i.e.* blocks whose Proof of Work solution contains more zeros than required, being suitable for a smaller difficulty target. For simplicity, we denote superblocks of level i as i-Superblocks. Superblocks are classified in levels, where a level denotes the extra amount of zeros a superblock has. This induces the new blockchain model: for every possible superblock level, a block is connected to the last preceding superblock of the given level.

The security guarantees of NIPoPoWs rely on the assumption that, for every possible superblock level i, a blockchain will contain approximately half i-Superblocks as $(i-1)$-Superblocks. This ensures the amount of superblock levels will be logarithmic in the chain's size. A NIPoPoW proof will be generated by picking a certain amount of superblocks of each possible level. This proof is expected to show the superblock distribution in the chain is the expected one. A more detailed review on NIPoPoWs is provided in Appendix A.2.

We proceed to compare the FlyClient protocol with the NIPoPoWs. Unlike the previous case, these two protocols rely on completely different ideas. Instead of considering the computational hardness of Proofs of Work, NIPoPoWs take advantage on the superblock concept and their assumed constant distribution along a chain. Therefore, we will have to restrict ourselves to a more heuristic comparison of the protocols' features guarantees.

One of the biggest issues with NIPoPoWs is their limitation to static difficulty blockchains. It is unclear how could the protocol be adapted to support dynamic difficulty chains. Moreover, Assumption 4 seems to be very reliant on the static difficulty setting. A variation in the difficulty targets would also change the probability to compute a superblock, seemingly altering the superblock distribution in the chain.

When it comes to security guarantees, it is difficult to compare Theorem 9 with the FlyClient security Theorem 5, since NIPoPoWs don't take into account sequential work or the (c, L)-assumption. However, the enunciation of Theorem 9 is extremely similar to the original security theorem in [Bün+19]. More importantly, NIPoPoWs offer security guarantees in the honest majority setting. We therefore don't think any of the protocols offers significative advantages with respect to the other in this aspect.

On the other hand, the two protocols show big differences when it comes to succinctness. While FlyClient proof succinctness is inherently guaranteed by protocol design, NIPoPoW succinctness is compromised by adversarial attacks. Theorem 10 only offers guarantees when the network is fully integrated by honest parties. In a deeper analysis, [KLZ21] guarantees NIPoPoW succinctness when adversarial hashing power is restricted to $\frac{1}{3}$ of the total hashing power. This seems to be a heavy burden on the NIPoPoW guarantees.

Another problem within the NIPoPoW protocol is its overreliance on the Proof of Work concept. This would be a problem if we wanted to adapt the protocol to support more resource-efficient consensus algorithms like Proof-of-Space or Proof-of-Stake. It must be said the FlyClient protocol has neither been adapted to support any of those algorithms, though.

In conclusion, we can see a handful of problems within the NIPoPoW protocol that put it in disadvantage with respect to the FlyClient protocol. The main ones are the inability to define NIPoPoWs in the dynamic diffuculty setting and the inconsistent succinctness guarantees.

5. Open problems

5.1. (c, L)-Assumption parameters

As we said before, the security guarantees of the FlyClient protocol rely on the assumed adversarial limitations to produce proofs of work. Remind the (c, L)-Assumption was discussed in Section 2.10.

However, it remains an open problem to determine suitable parameters $c \in [0, 1], L \in \mathbb{N}$ under which the assumption holds. [Bün+19] claims it is possible to infer such parameters as a function of the backbone protocol parameters and the adversarial computational power. Indeed, [Bün+19] provides a solution to the problem in the static difficulty model:

Lemma 32 ([Bün+19])**.** *For $L = \theta(w)$, where w is a blockchains' security parameter, and for every adversarial power rate μ, there exists a $c < 1$ such that the (c, L)-Assumption holds in the constant difficulty backbone.*

Nevertheless, there is a caveat with this result: In the proof, they only consider an adversary that mines a fork privately. It seems reasonable to consider a more general adversary that could deceive honest miners into working on its malicious fork. More importantly, it is left to provide a solution in the variable difficulty model.

5.2. Alternative consensus algorithms compatibility

Throughout this work, we have assumed a Proof of Work-based blockchain protocol. This is the most well known consensus algorithm in blockchain technology, but, since it bases security on computational hardness, it suffers from inefficiency. This makes PoW blockchains expensive to maintain and strongly energy consuming. As an alternative, other consensus algorithms have been proposed, mainly Proofs of Stake ([Kia+17]) and proofs of space ([Dzi+15]).

Proofs of stake require miners to hold a certain amount of blockchain tokens instead of solving a cryptographic puzzle. Because of this, generating a sequence of valid proofs of stake is computationally easy.

On the other hand, proofs of space require miners to allocate a large amount of space in their hard drives to mine blocks.

Checking the compatibility of FlyClient with the mentioned algorithms is an interesting problem, since it would make its practical adoption more attractive. We believe FlyClient is unlikely to be compatible with proofs of stake, since the computational ease of sequentially generating proofs makes the (c, L)-Assumption weak. On the other hand, adapting FlyClient to the space-heavy proofs of space seems more feasible. In fact, the already mentioned SNACKs, which security guarantees rely on the (c, L)-Assumption, have already been adapted to proofs of space in [Abu23].

References

[FS86] Amos Fiat and Adi Shamir. «How to Prove Yourself: Practical Solutions to Identification and Signature Problems». In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194.

[Nak08] Satoshi Nakamoto. «Bitcoin: A Peer-to-Peer Electronic Cash System». In: (2008).

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. «Publicly verifiable proofs of sequential work». In: *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*. Ed. by Robert D. Kleinberg. ACM, 2013, pp. 373–388.

[Dzi+15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. «Proofs of Space». In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*. Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9216. Lecture Notes in Computer Science. Springer, 2015, pp. 585–605.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. «The Bitcoin Backbone Protocol: Analysis and Applications». In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 281–310.

[GKL16] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. «The Bitcoin Backbone Protocol with Chains of Variable Difficulty». In: *IACR Cryptol. ePrint Arch.* (2016), p. 1048.

[KMZ17] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. «Non-Interactive Proofs of Proof-of-Work». In: *IACR Cryptol. ePrint Arch.* (2017), p. 963.

[Kia+17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. «Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol». In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Springer, 2017, pp. 357–388.

[Bon+18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. «Verifiable Delay Functions». In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 757–788.

[CP18] Bram Cohen and Krzysztof Pietrzak. «Simple Proofs of Sequential Work». In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 451–467.

[Bün+19] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. «Flyclient: Super-Light Clients for Cryptocurrencies». In: *IACR Cryptol. ePrint Arch.* (2019), p. 226.

[CP19] Bram Cohen and Krzysztof Pietrzak. *The Chia Network blockchain*. 2019. URL: `https://docs.chia.net/files/ChiaGreenPaper_20241008.pdf`.

[KLZ21] Aggelos Kiayias, Nikos Leonardos, and Dionysis Zindros. «Mining in Logarithmic Space». In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 3487–3501.

[Abu+22] Hamza Abusalah, Georg Fuchsbauer, Peter Gazi, and Karen Klein. «SNACKs: Leveraging Proofs of Sequential Work for Blockchain Light Clients». In: *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13791. Lecture Notes in Computer Science. Springer, 2022, pp. 806–836.

[Abu23] Hamza Abusalah. «SNACKs for Proof-of-Space Blockchains». In: *Financial Cryptography and Data Security - 27th International Conference, FC 2023, Bol, Brač, Croatia, May 1-5, 2023, Revised Selected Papers, Part II*. Ed. by Foteini Baldimtsi and Christian Cachin. Vol. 13951. Lecture Notes in Computer Science. Springer, 2023, pp. 3–17.

[AC23] Hamza Abusalah and Valerio Cini. «An Incremental PoSW for General Weight Distributions». In: *IACR Cryptol. ePrint Arch.* (2023), p. 1836.

A. Alternative solutions review

A.1. SNACKs review

The Succinct Non-Interactive Arguments of Chain Knowledge were introduced in [Abu+22] by Abusalah *et al.* in 2022. After defining a SNACK primitive, it is shown how to build a SNACK on top of Graph Labeling Proofs of Sequential Work (GL-PoSW). Additionally, concrete instantiations are provided based on skiplists and CP-based Directed Acyclic Graphs (*DAGs*). Blockchains are modeled as *DAGs*, in which block dependencies are represented as edges. The security guarantees rely on the adversarial inability to compute a heavy path. Similarly to the FlyClient protocol, the (c, L)-Assumption is used to formalize the adversarial limitations.

We will assume we can induce a topological order in the vertex set of our graph, hence we can identify it with the set $[n]_0 = \{0, \dots, n\}$. Given a *DAG* $G_n = ([n]_0, E_n)$, we define a graph labeling as a mapping $L : [n]_0 \to \{0,1\}^*$. If we introduce a weight distribution $\Omega_n : [n]_0 \to [0,1]$ over G_n's vertex set, we can consider weighted *DAGs* $\Gamma_n = (G_n, \Omega_n)$.

We will capture blockchain dependencies through a polynomial time relation R_ψ, where ψ equals the label $L(0)$ of the genesis block. This relation will take into account the block's index and label together with its parents' labels. We will consider a block $i \in [n]_0$ to be valid if $R_\psi(i, L(i), \{L(j)\}_{j \in \mathsf{parents}(i, G_n)}) = 1$. Block validity can be extended to paths by checking the validity of each block in the path. However, this requires involving the parent set of each block. We formalize this notion here:

Definition 25 (Valid paths)**.** *Let $G_n = ([n]_0, E_n)$ be a DAG, and R a relation. Furthermore, let P be a path in G_n and L_P a labeling of P, and $(p_v)_{v \in P}$ a $|P|$-tuple of bitstrings such that, for every $v \in P$, $p_v = (p_v[1], \dots, p_v[\deg(v)])$ contains the labels of $\mathsf{parents}(v, G_n)$. We say $(P, L_P, (p_v)_{v \in P})$ is R-valid if for all $v \in P$ with $\mathsf{parents}(v, G_n) = \{v_1, \dots, v_{\deg(v)}\}$, we have*

$$R(v, L_P(v), p_v) = 1 \text{ and } \forall i \in [\deg(v)] \text{ if } v_i \in P \text{ then } p_v[i] = L_P(v_i)$$

For a weighted DAG $\Gamma_n = (([n]_0, E_n), \Omega_n)$, we say $(P, L_P, (p_v)_{v \in P})$ is (α, R)-valid if in addition $\Omega_n(P) \geq \alpha$.

Graph labelings will be committed trough Vector Commitment schemes, which consist on the following algorithms:

- $cp \leftarrow \mathsf{setup}()$, computes parameters cp.
- $(\phi, aux) \leftarrow \mathsf{commit}(cp, (m_1, \dots, m_n))$, creates a commitment ϕ for a message vector $(m_1, \dots, m_n)$ and creates auxiliary information aux.
- $\rho \leftarrow \mathsf{open}(cp, \phi, aux, m_i, i)$, creates an opening ρ of ϕ to m_i at position i. Can be generalized to a set of indexes I by doing $\rho \leftarrow \mathsf{open}(cp, \phi, aux, \{m_i\}_{i \in I}, I)$.
- $\mathsf{ver}(cp, \phi, m_i, i, \rho)$, verifies an opening and returns a bit. Can be generalized to a set of indexes I by doing $\mathsf{ver}(cp, \phi, \{m_i\}_{i \in I}, I, \rho)$.

Vector commitments are position binding. This is, if an adversary generates openings $\rho \leftarrow \mathsf{open}(cp, \phi, aux, m_i, i)$, $\rho' \leftarrow \mathsf{open}(cp, \phi, aux, m_i', i)$ with $m_i \neq m_i'$, then $\mathsf{ver}(cp, \phi, m_i, i, \rho) \wedge \mathsf{ver}(cp, \phi, m_i', i, \rho') = 1$ will happen with negligible probability.

Prior to introducing the SNACK primitive, we will introduce the NP language it works on. This language consists on some setup parameters (including some vector parameters and a genesis block), an extended vector commitment of a graph labeling and a witness of the graph labeling of some path P.

Definition 26 (Chain commitment language)**.** *Let $\Gamma = (\Gamma_n)_{n \geq 0}$ be a family of weighted DAGs and Com a vector commitment scheme. We define*

$$\mathcal{R}^{(\alpha)}_{\Gamma, R, \mathsf{Com}} = \left\{ \begin{array}{l} (prm = (\sigma, cp), \eta = (\phi, n), \\ w = (P, L_P, (p_v)_{v \in P}, \rho)) \end{array} \; : \; \begin{array}{l} (P, L_P, (p_v)_{v \in P}) \text{ is } (\alpha, R)\text{-valid} \\ \wedge \mathsf{Com.ver}(cp, \phi, L_P, P, \rho) = 1 \end{array} \right\}$$

where R is a PT relation that depends on σ. The tuple $\eta = (\phi, n)$ contains a Com-Commitment ϕ to an R-valid labeling of G_n. w is a witness containing a Com-opening ρ of ϕ to L_P. We let $\mathcal{R}_{\Gamma, R, \mathsf{Com}} := \mathcal{R}^{(1)}_{\Gamma, R, \mathsf{Com}}$ and $\mathcal{L}_{\Gamma, R, \mathsf{Com}}$ denote the language defined by $\mathcal{R}_{\Gamma, R, \mathsf{Com}}$.

Now we can finally introduce the SNACK primitive as described in [Abu+22].

Definition 27 (SNACK)**.** *A tuple of PPT algorithms (P, V) is a succinct non-interactive argument of chain knowledge (SNACK) for the language $\mathcal{L}_{\Gamma,R,\mathsf{Com}}$ with parameter generator G if the following properties hold:*

Completeness: *For all $\lambda \in \mathbb{N}, prm \leftarrow \mathsf{G}(1^\lambda), \eta, w \in \{0,1\}^*$ with $(prm, \eta, w) \in \mathcal{R}_{\Gamma,R,\mathsf{Com}}$:*

$$\Pr[\pi \leftarrow \mathsf{P}(prm, \eta, w) : \mathsf{V}(prm, \eta\pi) = 1] = 1$$

*(α, ϵ)-**Knowledge soundness:** For every PPT prover $\tilde{\mathsf{P}}$ there exists a PPT extractor E such that*

$$\Pr\left[\begin{array}{c} prm \leftarrow \mathsf{G}(1^k); r \xleftarrow{\$} \{0,1\}^{poly(\lambda)} \\ (\eta, \pi) := \tilde{\mathsf{P}}(prm, r) \\ w' \leftarrow \mathsf{E}(prm, r) \end{array} : \begin{array}{c} \mathsf{V}(prm, \eta, \pi) = 1 \wedge \\ \mathcal{R}^{(\alpha)}_{\Gamma,R,\mathsf{Com}}(prm, \eta, w') = 0 \end{array}\right] \leq \epsilon(\lambda) \tag{3}$$

with $\mathcal{R}^{(\alpha)}_{\Gamma,R,\mathsf{Com}}$ from Definition 26. We say that (P, V) has universal (α, ϵ)-knowledge soundness if there exists a single extractor $\mathsf{E}^{\tilde{P}}$ with oracle access to $\tilde{P}$ that satisfies (3) for all PPT provers $\tilde{P}$.

Succintness: *For all $prm \leftarrow \mathsf{G}(1^\lambda), (prm, \eta, w) \in \mathcal{R}_{\Gamma,R,\mathsf{Com}}$ and $\pi \leftarrow \mathsf{P}(n, \eta, w)$, we have $|\pi| \leq poly(\lambda, \log n)$, P runs in time $poly(\lambda, n)$ and V runs in time $poly(\lambda, \log n)$.*

In order to instantiate the SNACK primitive with an actual protocol, [Abu+22] shows how to construct it from a GL-PoSW. Blockchains will be intertwined with GL-PoSW schemes, making miners simultaneously compute PoSW labels and PoW solutions. A template on how it is done is provided on Figure 29. It must be noted that the displayed protocol is an ACK (Argument of Chain Knowledge). This is because it is interactive and not necessarily succinct. Succinctness depends on the underlying GL-PoSW, while non-interactivity can be achieved by applying the Fiat-Shamir heuristic.

Theorem 6 provides a relation between the SNACK soundness properties and its underlying GL-PoSW protocol properties.

Verifier V = (V_0,V_1)

Stage V_0: On input η :

1. $\forall i \in [t]$ do $\iota_i \xleftarrow{\$} \Omega_n$
2. $\iota_0 = 0$
3. send $\iota := (\iota_i)_{i=0}^t$ to P

Stage V_1: On input $\gamma = (o_i, \rho_i)_{i=0}^t$:

1. $\forall i \in \{1, \ldots, t\}$ do:
   (a) $b_i^{(1)} := \mathsf{PoSW.ver}(\chi, \iota_i, o_i)$
   (b) $b_i^{(2)} := R_\sigma(\iota_i, L_K(\iota_i), p_i)$
   (c) $b_i^{(3)} := \mathsf{Com.ver}(cp, \phi, L_K(\iota_i), \iota_i, \rho)$
2. output $\bigwedge_{i=0}^t b_i^{(1)} \wedge b_i^{(2)} \wedge b_i^{(3)}$

Prover P:

On input $(\eta, (L_K(j))_{j\in[n]_0}, aux_n)$ and ι:

1. parse η as $\eta = (cp, \phi, n)$
2. $\forall i \in [t]_0$ do:
   (a) $o_i \leftarrow \mathsf{PoSW.open}(\chi, cp, \phi, aux_n, (L_K(j))_{j\in[n]_0}, \iota_i)$
   (b) $\rho_i \leftarrow \mathsf{Com.open}(cp, \phi, aux, L_K(\iota_i), \iota_i)$
   (c) $\gamma_i := (o_i, \rho_i)$
3. send $\gamma := (\gamma_i)_{i=0}^t$ to V_1

Figure 29: Construction of an ACK from a GL-PoSW.

Theorem 6. *Let (P,V) be the non-interactive version of the protocol shown in Figure 29, in the the random oracle model. If its associated PoSW scheme is (α, ϵ)-knowledge sound, then (P,V) is an (α, ϵ)-knowledge sound SNACK.*

We now indicate the two *DAG* constructions that were shown in [Abu+22]. The first one consists on a chain with extra edges.

Construction 1 (Skiplists)**.** *We define G_n as a DAG with vertex set $[n]_0$ and edge set*

$$E = \{(i, j) \in [n]_0^2 : \exists k \geq 0 \ s.t \ (j - i) = 2^k \wedge 2^k | i$$

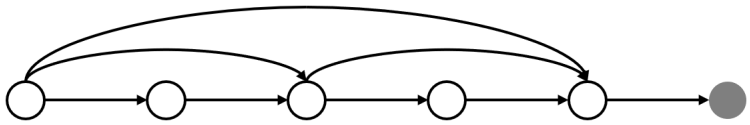

Figure 30: Illustration of a skiplist on 5 nodes. The last node, marked in gray, is introduced as a token containing a commitment to the rest of the graph.

The second construction is very similar to the CP-Graphs used in [CP18], therefore is similar to the MMG construction we introduced in Definition 15. The main difference with our construction is that every node will have additional incoming edges. Remind that in the SNACK setting every graph node is meant to represent a block with a PoW solution while the FlyClient MMG where only the leaves contain PoW solutions.

Construction 2 (Modified CP-Graphs)**.** *We define $G_n = (V, E)$ as a graph where V is the vertex set of an MMR M on n leaves and $E = E' \cup E''$, where:*

$$E' = \{(x, y) : y \in V \text{ and } x = \mathsf{lc}(y) \text{ or } x = \mathsf{rc}(y)\}$$

$$E'' = \{(x, y) : x \in \mathsf{lp}(y, M)\}$$

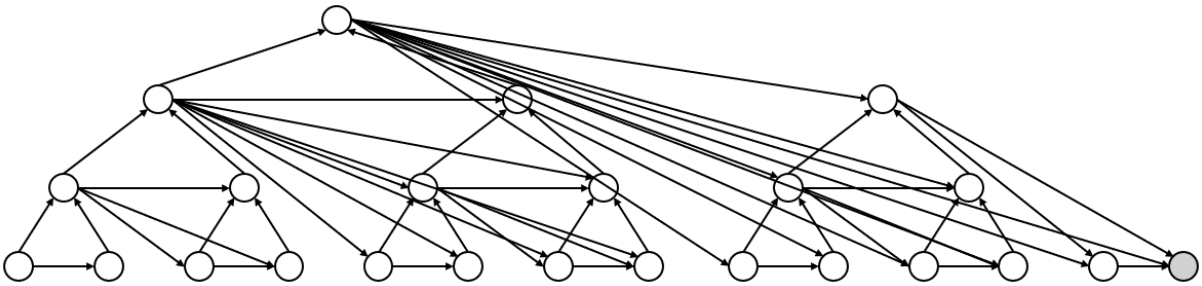

Figure 31: Illustration of a modified CP-Graph on 23 nodes. The last node, marked in gray, is introduced as a token containing a commitment to the rest of the graph.

In particular, [CP18] uses a GL-PoSW scheme based on skiplists and shortest path commitment (*i.e.* the prover will create an opening for a block i in a graph G_n by sending the labels in the shortest path in G_n containing both i and n). In this setting, we have the following result:

Theorem 7. *Let $\alpha \in (0, 1]$. The skiplist-based GL-PoSW construction in [CP18] with parameter t and arbitrary weight function Ω_n is an (α, ϵ)-knowledge-sound augmented GL-PoSW with $\epsilon = \alpha^t + 3 \cdot q^2/2^w$, where q is an upper bound on the number of the adversary's oracle queries.*

Combining Theorems 6 and 7, we get that the non-interactive version of the protocol shown in Figure 29 is an $(\alpha, \alpha^t + 3 \cdot q^2/2^w)$-knowledge-sound SNACK. The construction based on CP-Graphs has the same soundness properties.

Finally, we will show how the SNACK protocol is applied to light client protocols. First of all, we remind the (c, L)-Assumption we introduced in Section 2.10 will be used to ensure security against forking adversaries. However, we will be using a slightly different version in which, after forking a chain at a certain point, a prover will only be required to have a c fraction of valid blocks in a certain path. We will define as ϵ_F the probability for an adversary to compute a (c, L)-fork. We will define a multiple prover bootstrapping protocol, in which each prover submits a SNACK proof, which gets to be checked by the verifier. Similarly to the FlyClient protocol, we will require at least one of the provers to be honest. The bootstrapping protocol is shown in Figure 32.

Let Π be a blockchain protocol with validity relation R, chain graph $(K_n)_{n\geq 0}$ and parameters prm. On input N tuples of the form

$$(\phi, n, \pi, (k_i)_{i=n-L+1}^{n}, (\iota_j, k_{\iota_j}, \rho_{\iota_j})_{j=1}^{q}) \text{ for some } q \leq L \cdot \deg(K_n)$$

for all tuples, ordered by decreasing values of n, check the following:
(a) Let $m := n - L$; check whether ϕ is contained in k_{m+1}
(b) $\mathsf{SNACK.V}(prm, (\phi, m), \pi) \stackrel{?}{=} 1$, where SNACK is (α, ϵ)-knowledge sound
(c) For all $i \in [m+1 : n]$, verify $R(i, k_i, (k_j)_{j \in \mathsf{parents}(i, K_n)}) \stackrel{?}{=} 1$
(d) For all $j \in [1 : q]$: $\mathsf{Com.ver}(cp, \phi, k_{\iota_j}, \iota_j, \rho_{\iota_j}) \stackrel{?}{=} 1$
(e) If all checks verify, return (ϕ, n)
Return $\perp$

Figure 32: Multi-Prover Bootstraping Common Commitment

Theorem 8. *Let Π be a blockchain protocol with underlying graph $(K_n)_{n\geq 0}$ satisfying k-common prefix. Let $c \in (0,1)$ and $L \in \mathbb{N}$ with $L > k$. For $m \in \mathbb{N}$ define $\alpha_m, \Omega_m : [m]_0 \to [0,1]$ as*

$$\alpha_m := 1 - (\log_c(\frac{L-1}{m+L}))^{-1}$$

$$\Omega_m := S \cdot \frac{1}{m+L-i} \text{ for } 0 \leq i \leq m \text{ where } S := (\sum_{j=0}^{m} \frac{1}{m+L-j})^{-1}$$

Then, under the premises of the (c, L)-Assumption, an adversary will trying to create a (c, L)-fork in K_{n^*}, such that, with $m^* = n^* - L$:

- *P contains the last L blocks, i.e., $[m^* + 1 : n^*] \subseteq P$*
- *P has weight at least α_{m^*} w.r.t. Ω_{m^*}. This is, $\Omega_{m^*}(P \cap [m^*]_0) \geq \alpha_{m^*}$*

will succeed with probability at most ϵ_F.

Corollary 1. *Let $k, L \in \mathbb{N}$ with $L > k$ and let Π be a blockchain protocol with validity relation R and graph family $(K_n)_{n\geq 0}$ that satisfies k-common prefix. Assume the honest length is $n_h > L$ and consider a light client running the bootstrapping protocol displayed in Figure 32 with N full nodes, at least one of which is honest, at least one of which is honest and synchronized. Let (ϕ^*, n^*) be the output of the light client. For $m \geq 0$, let α_m and Ω_m be as in Theorem 8. If*

- *Com is ϵ_C-position binding.*
- *SNACK is $(\alpha_{n^*-L}, \epsilon)$-knowledge-sound for language $\mathcal{L}_{\Gamma,R,\mathsf{Com}}$ where $\Gamma := (\Gamma_m = (K_m, \Omega_m))_{m\geq 0}$*

then the client securely L-CP bootstraps except with security $\epsilon + \epsilon_C + \epsilon_F$.

A.2. NIPoPoWs review

The Non-Interactive Proofs of Proof-of-Work (NIPoPoWs, from now on), were introduced by Kiayias *et.al.* in [KMZ17]. To the extent of our knowledge, it was the first proposed blockchain light-client protocol with a sublinear proof size. Unlike the FlyClient and SNACK protocols, NIPoPoWs are based on the concept of superblocks, *i.e.* blocks whose Proof of Work solution would suit a smaller difficulty target than the required one. We formalize the concept here:

Definition 28. *Given a block $B = \langle h, x, ctr \rangle, \mu \in \mathbb{N}$ and a difficulty target T, we say B is a μ-superblock with respect to T if $H(ctr, G(h, x)) < \frac{T}{2^\mu}$.*

We will refer to the identity of a block B as $\mathsf{id}(B) = H(ctr, G(h, x))$. Intuitively, a μ-superblock represents a block whose PoW solution provides with μ extra 0's on top of those required by T. This holds because, if B is a μ-superblock, then $\mu = \lfloor \log_2(T) - \log_2(\mathsf{id}(B)) \rfloor$. It is easy to see that, given a μ-superblock B, B is also a $(\mu - 1)$-superblock. Given a block B, we will refer to its level as the biggest possible natural μ such that B is a μ-superblock. We will denote it by $\mathsf{level}(B)$.

NIPoPoWs use a custom blockchain model based on the superblock concept. Instead of containing the hash of the previous block, each block will contain a sequence of hashes representing the most recent μ-superblock for each possible level μ. Recall that, since any μ-superblock is also a μ'-superblock for $\mu' < \mu$, this sequence could contain repeated references to the same block.

Definition 29 (NIPoPoW block)**.** *A NIPoPoW block is a tuple $B = \langle \mathsf{interlink}, x, ctr \rangle$ is a tuple such that $\mathsf{interlink}$ is a sequence of hashes verifying that, for every possible level i, $\mathsf{interlink}[i]$ contains $\mathsf{id}(B_i)$, with B_i being the most recent i-superblock in the chain. We will say B is valid with respect to a difficulty target if $H(ctr, G(x, \mathsf{interlink})) < T$. We define $\mathsf{id}(B) := H(ctr, G(x, \mathsf{interlink}))$.*

Given $\mu \in \mathbb{N}$, we say B is a μ-superblock if $\mathsf{id}(B) < \frac{T}{2^\mu}$.

Now we can define a NIPoPoW blockchain. We will assume a fixed difficulty target T since the NIPoPoW protocol is not defined in variable difficulty. Figure 33 displays a NIPoPoW blockchain on 11 blocks.

Definition 30. *A NIPoPoW blockchain is a sequence of valid blocks with respect to a fixed difficulty target T, as indicated in Definition 29. The genesis block, denoted by gen is considered a ∞-superblock by convention, and will therefore be contained in every other block's $\mathsf{interlink}$.*

For chain addressing, we will use Python-like indexes. Given a chain $\mathcal{C}$ and $i \geq 0$, $\mathcal{C}[i]$ represents the i-th block in the chain. $\mathcal{C}[-1]$ represents the rightmost block in the chain. A range $\mathcal{C}[i:j]$ is a subchain starting from i (inclusive) to j (exclusive).

Given two chains $\mathcal{C}_1, \mathcal{C}_2$, we define $\mathcal{C}_1 \cup \mathcal{C}_2$ as the result of orderly merging the blocks in both chains. However, we will only be able to define $\mathcal{C}_1 \cup \mathcal{C}_2$ as a chain if, for every $0 < i < |\mathcal{C}_1 \cup \mathcal{C}_2|$, $\mathsf{id}((\mathcal{C}_1 \cup \mathcal{C}_2)[i-1]) \in \mathsf{interlink}((\mathcal{C}_1 \cup \mathcal{C}_2)[i])$. Analogously, we will define $\mathcal{C}_1 \cap \mathcal{C}_2 = \{B : B \in \mathcal{C}_1 \wedge B \in \mathcal{C}_2\}$.

We define the *last common ancestor* of two chains $\mathcal{C}_1, \mathcal{C}_2$ as $\mathsf{LCA}(\mathcal{C}_1, \mathcal{C}_2) = (\mathcal{C}_1 \cap \mathcal{C}_2)[-1]$. This is, LCA represents the oldest block that belongs to both chains.

Given a chain $\mathcal{C}$ and a level μ, we define a μ-superchain $\mathcal{C} \uparrow^\mu := \{B \in \mathcal{C} : \mathsf{level}(B) \geq \mu\}$. This is, a chain that only contains μ-superblocks.

Given chains $\mathcal{C}' \subseteq \mathcal{C}$ we define a downchain as $\mathcal{C}' \downarrow_{\mathcal{C}} = \mathcal{C}[\mathcal{C}'[0] : \mathcal{C}'[-1]]$. When $\mathcal{C}$ is implicit in the context, we can simply write $\mathcal{C}' \downarrow$.

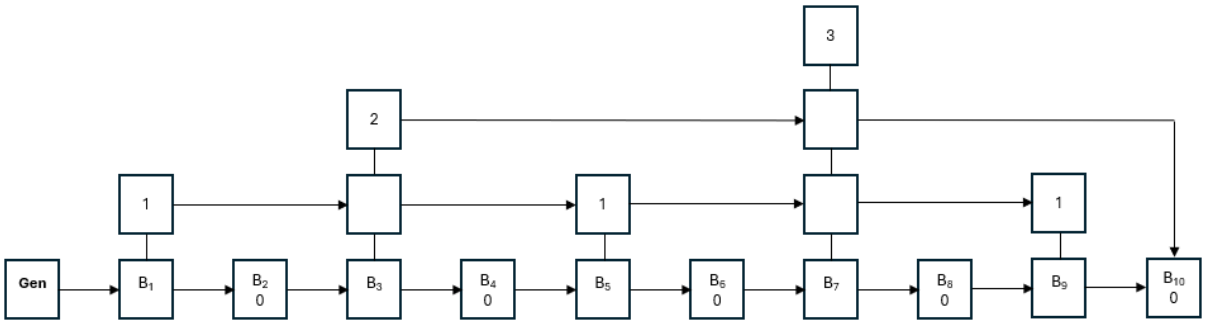

Figure 33: Illustration of a NIPoPoW blockchain on 11 blocks. Vertically stacked blocks denote superblock level, which is indicated at the top of every block. Incoming edges denote $\mathsf{interlink}$ inclusion. The incoming edges from the genesis block gen to every other block have been omitted for simplicity.

NIPoPoW blockchains are expanded through the $\mathsf{UpdateInterlink}$ algorithm. This is, in order to calculate the interlink of a chain's incoming block, we will take its rightmost block B and suitably update $\mathsf{interlink}(B)$. More specifically, for every $i \leq \mathsf{level}(B)$, we set $\mathsf{interlink}(B)[i]$ to B. Figure 34 displays the UpdateInterlink algorithm.

Algorithm $\mathsf{UpdateInterlink}(B)$

```
//B is the rightmost block in a NIPoPoW chain
interlink ← interlink(B)
for μ = 0 to level(B) do:
    interlink[μ] ← id(B)
return interlink
```

Figure 34: Description of the UpdateInterlink algorithm

We now have to describe the NIPoPoW security concept. Intuitively, we will consider a predicate to be secure if, after an honest party's chain $\mathcal{C}$ starts to verify Q, the rest of the honest parties in the network take at most ηk rounds to verify it. The choice ηk is not casual: they represent the chain growth and common prefix parameters in the backbone protocol, respectively. Therefore, ηk is meant to represent the average number of rounds a chain block takes to get into the stable prefix.

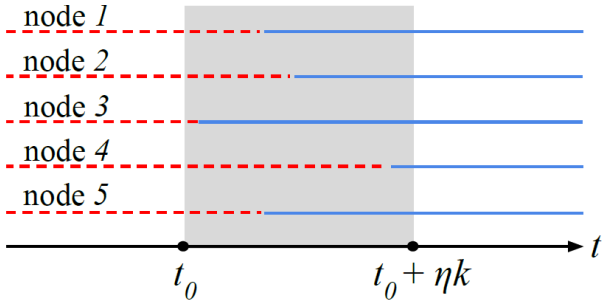

Figure 35: Illustration of the NIPoPoW security concept. A red dashed line indicates that the corresponding node doesn't verify a certain predicate. A continuous blue line indicates the node verifies it. As we can see, a secure predicate is required to be simultaneously validated by all honest nodes in a window of at most ηk rounds.

An example of the predicates we will seek to prove secure is $Q \equiv$"the transaction tx is inserted and confirmed in the chain $\mathcal{C}$".

Definition 31 (Security)**.** *A blockchain proof protocol (P, V) about a predicate Q is secure if for all environments and for all PPT adversaries $\mathcal{A}$ and for all rounds $r \geq \eta k$, if V receives a set of proofs $\mathcal{P}$ at the beginning of round r, at least one of which has been generated by the honest prover P, then the output of V at the end of round r will have the following constraints:*

- *If the output of V is false, then the evaluation of $Q(\mathcal{C})$ for all honest parties must be false at the end of round $r - \eta k$.*
- *If the output of V is true, then the evaluation of $Q(\mathcal{C})$ for all honest parties must be true at the end of round $r + \eta k$.*

NIPoPoWs offer security guarantees over suffix-sensitive predicates. Intuitively, if a predicate is suffix-sensitive, then it's univocally determined by the k-suffix of the chain. We formalize their definition here:

Definition 32 (Suffix sensitivity)**.** *A chain predicate Q is called k-suffix sensitive if for all chains $\mathcal{C}, \mathcal{C}'$ with $|\mathcal{C}| \geq k$ and $|\mathcal{C}'| \geq k$ such that $\mathcal{C}[-k:] = \mathcal{C}'[-k:]$ we have that $Q(\mathcal{C}) = Q(\mathcal{C}')$.*

The concrete predicate NIPoPoW will prove security over is the good superchain one. We introduce it here:

Definition 33 (Locally good superchain)**.** *A superchain $\mathcal{C}'$ of level μ with underlying chain $\mathcal{C}$ is said to be μ-locally-good with respect to security parameter δ, written $\mathsf{local\text{-}good}_\delta(\mathcal{C}', \mathcal{C}, \mu)$, if $|\mathcal{C}'| > (1-\delta)2^{-\mu}|\mathcal{C}|$.*

Definition 34 (Superchain quality)**.** *The (δ, m) superquality property Q^{μ}_{scm} of a chain $\mathcal{C}$ pertaining to level μ with security parameters $\delta \in \mathbb{R}$ and $m \in \mathbb{N}$ states that for all $m' \geq m$, it holds that* $\mathsf{local\text{-}good}_{\delta}(\mathcal{C} \uparrow^{\mu} [-m' :], \mathcal{C} \uparrow^{\mu} [-m' :] \downarrow, \mu)$*. That is, all sufficiently large suffixes of the chain are μ-locally good.*

Definition 35 (Multilevel quality)**.** *A μ-superchain $\mathcal{C}'$ is said to have multilevel quality, written* $\mathsf{multi\text{-}good}_{\delta,k_1}(\mathcal{C}, \mathcal{C}'\mu)$ *with respect to an underlying chain $\mathcal{C} = \mathcal{C}' \downarrow$ with security parameters k_1, δ if for all $\mu' < \mu$ it holds that for any $\mathcal{C}^* \subseteq \mathcal{C}$, if $|\mathcal{C}^* \uparrow^{\mu'}| \geq k_1$, then $|\mathcal{C}^* \uparrow^{\mu}| \geq (1-\delta)2^{\mu'-\mu}|\mathcal{C}^* \uparrow^{\mu'}|$.*

Definition 36 (Good superchain)**.** *A μ-superchain $\mathcal{C}'$ is said to be good, written* $\mathsf{good}_{\delta,k_1}(\mathcal{C}, \mathcal{C}', \mu)$*, with respect to an underlying chain $\mathcal{C} = \mathcal{C}' \downarrow$ if it has both superquality and multilevel quality with parameters (δ, m).*

Intuitively, a *good* superchain is one in which each μ-superchain contains approximately half of the elements of the $(\mu-1)$-superchain. This is assumed in [KMZ17] to be a property holding for every honestly produced chain. We leave the formal statement here:

Assumption 4 (NIPoPoW)**.** *Given a static difficulty blockchain protocol Π, let $\mathcal{C}$ be a chain held by an honest party taking part in Π. Then, for every possible level μ, it will hold that $|\mathcal{C} \uparrow^{\mu}| \approx |\mathcal{C}|/2^{\mu}$.*

Now it's the time to introduce the NIPoPoW protocol for suffix-sensitive predicates as it is described in [KMZ17]. In it, they introduce a simplified version which they later refine to prevent adversarial attacks. We will directly introduce the refined version.

We describe the NIPoPoW protocol as a tuple of algorithms (P,V) that is instantiated by security parameters $k, m \in \mathbb{N}$ and $\delta \in [0,1]$. k represents the suffix size we require to prove a certain suffix-sensitive predicate on a chain and will be generically instantiated with the common prefix parameter in our blockchain protocol. m will determine the sections of the chain that will be sampled. δ will represent the local goodness we will be requiring on the sampled chain sections. The hash function size w will also be a security parameter. Figure 36 displays the NIPoPoW protocol.

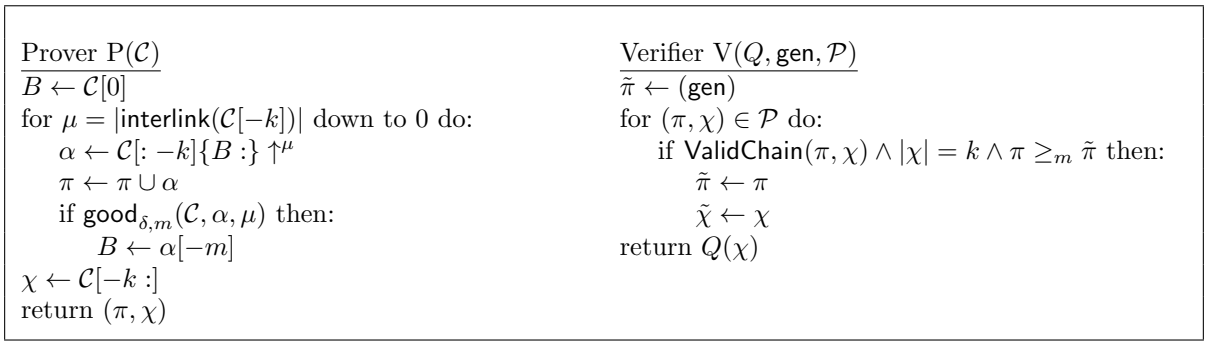

Figure 36: NIPoPoW protocol instantiated by security parameters $k, m \in \mathbb{N}$ and $\delta \in [0,1]$

On input a chain $\mathcal{C}$, a prover will provide the k-suffix of $\mathcal{C}$, which will be denoted by χ and a subchain of the k-prefix of $\mathcal{C}$, denoted by π. In order to obtain π, we will extract a suitable section of each possible μ-superchain of $\mathcal{C}$. The local-goodness check is a refinement introduced to prevent attacks.

As for the verifier, it will be holding the predicate Q it wants to evaluate, the chain genesis block gen and a set of NIPoPoW proofs $\mathcal{P}$ that were sent by a set of full nodes. For each proof $(\pi, \chi) \in \mathcal{P}$, the verifier will check the following:

- The chain suffix χ has length k.
- The ValidChain predicate is evaluated on (π, χ). This is, for all $0 < i < |\pi|$, $\mathsf{interlink}(\pi[i])$ contains $\mathsf{id}(\pi[i-1])$, $\mathsf{interlink}(\chi[0])$ contains $\mathsf{id}(\pi[-1])$ and for all $0 < i < k$, $\mathsf{interlink}(\chi[i])$ contains $\mathsf{id}(\chi[i-1])$.
- Each proof (π, χ) in $\mathcal{P}$ is compared with the stored proof $(\tilde{\pi}, \tilde{\chi})$ through the operator $\geq_m$. If $(\pi, \chi) \geq_m (\tilde{\pi}, \tilde{\chi})$, then the new proof is stored. Operator $\geq_m$ is displayed in Figure 37.

function $\mathsf{best\text{-}arg}_m(\pi, b)$:
    $M \leftarrow \{\mu : |\pi \uparrow^\mu \{b:\}| \geq m\} \cup 0$
    return $\max_{m \in M}\{2^\mu \cdot |\pi \uparrow^\mu \{b:\}|\}$
operator $\geq_m (\pi_A, \pi, B)$:
    $b \leftarrow \mathsf{LCA}(\pi_A, \pi_B)$
    return $\mathsf{best\text{-}arg}_m(\pi_A, b) \geq \mathsf{best\text{-}arg}_m(\pi_B, b)$

Figure 37: Operator $\geq_m$ implementation

Finally, the verifier will evaluate the predicate Q on the obtained proof suffix χ. Recall that Q is meant to be a suffix-sensitive predicate.

The following theorem describes the security guarantees of the protocol described in Figure 36. Recall the security parameter w is the size of the hash function that is used to compute the blockchain.

Theorem 9 (security)**.** *Assuming honest majority, the non-interactive proofs-of-proof-of-work construction for computable k-stable monotonic suffix-sensitive predicates is secure with overwhelming probability in w.*

Regarding succinctness, there is a caveat: An adversary could attack succinctness by attacking a chain's superquality at a certain level μ. If we observe the prover algorithm in Figure 36, this would imply including the whole $(\mu - 1)$-superchain in the generated proof, making it longer. The succinctness result is therefore enunciated for an optimistic environment, in which only honest parties take part in the blockchain protocol.

Theorem 10 (Optimistic succinctness)**.** *If all players are honest and the network scheduling is random, non-interactive proofs-of-proof-of-work produced by honest provers are succinct with the number of blocks bounded by $4m \log(|\mathcal{C}|)$, with overwhelming probability in m.*