FlyClient: Super-Light Clients for Cryptocurrencies

Benedikt Bünz[1*†], Lucianna Kiffer[2*†], Loi Luu[3*], and Mahdi Zamani[4]

[1] *Stanford University,* [2] *Northeastern University,* [3] *Kyber Network,* [4] *Visa Research*

Abstract

To ensure the validity of transactions, cryptocurrencies such as Bitcoin and Ethereum require nodes to verify that a proof-of-work blockchain is valid. Unfortunately, this often entails downloading and verifying all transaction blocks, taking days and gigabytes of bandwidth and storage to verify the blockchain. As a result, clients with limited resources such as mobile phones cannot verify transactions independently without trusting full nodes. As a solution to this, Bitcoin and Ethereum offer light clients known *simplified payment verification (SPV)* clients, that can verify the chain by downloading only the block headers which have significantly-smaller size than the full blocks. Unfortunately, the storage and bandwidth of SPV clients still increase linearly with the chain length. In Ethereum, for example, an SPV client needs to download and store more than 3.6 GB of data.

Recently, Kiayias *et al.* proposed a solution, known as the non-interactive proof of proof-of-work (NIPoPoW), that requires a light client to download and store only a polylogarithmic number of block headers. Unfortunately, NIPoPoWs suffer from several drawbacks: they are succinct only as long as no adversary influences the honest chain. Furthermore, they can only be used for proof-of-work chains that have a fixed block difficulty, which is not the case in most cryptocurrencies, including Bitcoin and Ethereum, that require adjusting block difficulty frequently according to the network hashrate.

In this paper, we introduce FlyClient, a novel transaction verification protocol for light clients that is efficient both asymptotically and practically. Our protocol requires to download only a logarithmic number of block headers to synchronize and verify transactions while storing only a single block header between executions. We formally prove that FlyClient is optimal for this class of protocols. For Ethereum, our protocol achieves a proof size of only less than 500 KB. This is achieved by utilizing a simple design based on Merkle Mountain Range (MMR) commitments and a probabilistic block sampling protocol. FlyClient overcomes the limitations of NIPoPoWs and generates shorter proofs over all measured parameters. We also discuss how FlyClient can be implemented via a soft fork in Bitcoin/Ethereum.

1 Introduction

Today, many financial service providers can deliver digital payment services on low-capacity clients such as phones, wearable devices (*e.g.*, smartwatches and fitness trackers), and Internet-of-things (IoT) devices for added convenience and security [10, 9, 13]. Such clients are characterized by their limited access to storage, bandwidth, and computation while still requiring instant confirmation latency.

While traditional financial services provide efficiency and strong security guarantees for mobile clients, they use centralized payment systems, and are generally inapplicable to decentralized cryptocurrency networks such as Bitcoin [42] and Ethereum [19] due to their entirely-different designs. On the other hand, current solutions for verifying transactions made over most cryptocurrency networks do not suit low-capacity devices. This is typically due to the large amounts of storage, computation, and bandwidth required by decentralized protocols to verify transactions on decentralized ledgers. These ledgers rely on state-machine replication across many untrusted nodes in the network. One option for low-capacity clients is to rely on a powerful trusted party who can verify payments on behalf of the client. The existence of such a trusted entity, however, greatly opposes the decentralized nature of cryptocurrency networks.

Proof-of-Work Blockchains. Most cryptocurrencies, including Bitcoin and Ethereum, maintain an

[*] This work was done by the author as part of an internship at Visa Research.

[†] Both authors contributed equally.

append-only ledger, known as a *blockchain*, which stores a sequence of blocks of transactions chained together using cryptographic hashes. These blocks are created and appended to the blockchain via a mining process, where the participating nodes, known as *miners*, compete to become the next block proposer by typically solving a computationally-intensive puzzle, known as a *proof-of-work (PoW)* [24], with sufficient difficulty. Through a gossip protocol (*e.g.*, [34]) initiated by the block proposer, every miner receives each block including a PoW solution and appends the block to their local copies of the blockchain if the solution is valid. Since this process is not coordinated by any central party (nor by any traditional consensus protocol such as [20]), the blockchain may *fork* into multiple chains; *e.g.*, due to multiple solutions found for the same puzzle by different miners, or even due to malicious behaviors. To agree on the same chain consistently with other miners, each miner downloads all blocks in every chain and picks and follows the one with the highest total difficulty. Using this strategy, it is shown that, in the long run, the network will agree on a single chain [27, 44, 39], known as the *honest (valid) chain*. This chain consists of n *valid blocks*, where each block contains a cryptographic proof ensuring that the block creator has spent (or locked) a certain amount of a resource (*e.g.*, computation or space) uniquely for this block.

To verify that a blockchain is valid without participating in the mining process, a client may choose to download every block from a miner or a *full node* who holds a complete copy of the entire chain. Currently, downloading and verifying all blocks in Bitcoin and Ethereum requires a node to download more than 190 GB and 1.8TB of data respectively [4, 5], taking days to synchronize the node's local blockchain [3, 1]. Such a requirement causes a long delay for regular clients and makes it nearly impossible for storage-limited clients to quickly verify transactions.

Light Clients. The original Bitcoin design [42] describes a faster synchronization mechanism, known as the *simplified payment verification (SPV)* that allows lightweight verification of transactions on the blockchain by what is typically referred to as an SPV client (also known as a *light* [35] or *thin* client [12]).

Instead of downloading all blocks from a full node, an SPV client downloads only the *headers* of each block that amounts to a much smaller synchronization overhead than the full blocks (80 bytes versus 1 MB per block in Bitcoin). Block headers contain enough information to ensure that (1) valid PoW was done for the block, (2) a transaction is included in the block and (3) a block is included at a certain position on the most difficult chain mined so far. The transaction validation utilizes a Merkle tree commitment to all transactions in a block which is stored in each block header. A *SPV proof* gives a Merkle inclusion proof for a transaction of interest to the light client. Notably, a light client does not verify all transactions and could in theory receive an invalid blockchain. It essentially relies on the assumption that the chain with most proof-of-work contains only valid transactions and follows the rules of the system.

Besides being useful for lightweight verification of transactions, light clients also enable various applications to a broad class of users who need to verify a log of past events recorded on a blockchain. For example, SPV proofs can be used for efficient verification of cross-chain transactions that rely on funds or states recorded on another chain. Such transactions happen when, for example, exchanging cryptocurrencies [33, 6] or transferring assets to sidechains [17, 36, 38]. For example, blockchain-based notary services have been recently developed [8, 11] that allow lightweight verification of the correctness and the integrity of documents uploaded on a blockchain.

Although relying only on block headers significantly reduces the verification overhead of SPV clients, it still incurs a large overhead on resource-limited clients, especially when considering the fact that this overhead increases linearly with the number of blocks. This has already become a major concern for Ethereum due to its significantly shorter block interval than Bitcoin (∼15 seconds vs ∼10 minutes) and significantly larger block headers (528 bytes vs 80 bytes). Given that the Ethereum blockchain has more than 7 million blocks (as of late 2018 [5]), an SPV client wishing to verify Ethereum transactions would have to download and store more than 3.6 GB of data. The client has to either download a fresh copy of the data every time it wants to verify a transaction or keep a local copy in its storage and only download the changes since the last synchronization. In either case, this puts a large burden on the client. The problem is further amplified for users that run clients for many blockchains or systems that use sidechains [36].[1]

Sublinear Light Clients. One may wonder if it is possible for a client to verify any event on a blockchain of length n by downloading and/or storing only a sublinear (in n) amount of information. In fact, such a performance gain comes with an important security challenge: Since such a client cannot verify every PoW in the received blockchain, it can be tricked into accepting an invalid chain by a malicious prover who can precompute a sufficiently-long chain using its limited computational power. Recently, Kiayias *et al.* [35] introduce an interactive proof mechanism, known as *proofs of proof of work (PoPoW)*, through which a prover can convince a verifier with high probability in sublinear time and communication that a sufficient number of PoWs have been solved to create a chain.

In a later work [36], Kiayias *et al.* provide an attack against PoPoW and propose a non-interactive, yet-sublinear alternative solution known as *non-interactive PoPoW (NIPoPoW)*. Unfortunately, both PoPoW and NIPoPoW have several drawbacks as follows: (1) Both solutions are complex and difficult to understand, and work only if a fixed difficulty is assumed for all blocks. Unfortunately, this is not a realistic assumption in practice due the variable combined hashing power of miners in most PoW-based cryptocurrency networks. For example, the block difficulty in Bitcoin has shown exponential growth over the network's lifetime in the past decade [14]. Moreover, the practicality of these solutions is yet to be shown as the constant factors in the protocol's overhead seems to be large.

Our Contribution. We propose FlyClient, a new blockchain verification protocol for light clients in cryptocurrencies such as Bitcoin and Ethereum. Unlike regular SPV clients that use a linear amount of bandwidth and storage in the length of the chain, a FlyClient is required to download only a logarithmic number of block headers to verify the validity of the chain. Once the chain is verified, the client needs to store only a single block to efficiently verify the inclusion of any transaction on the chain. The FlyClient protocol overcomes the limitations of NIPoPoW. It provides succinct proofs even in the face of an adversary. The protocol is parameterized by $c \in [0, 1)$ and $\lambda \in \mathbb{N}$ such that an adversary controlling at most a c fraction of the honest mining power succeeds with probability at most $2^{-\lambda}$. FlyClient's block inclusion proofs are only $\log_2(n)$ hashes for a blockchain of size n. In Ethereum this is less than 1KB vs. 1.5MB proofs for NIPoPoW. These proofs need to be provided for every single transaction validation. And finally FlyClient works for chains with variable difficulty and we demonstrate its concrete efficiency on Ethereum (see Table 1).

To achieve this, we use the following techniques:

- **Probabilistic Sampling with Variable Difficulty:** We introduce a protocol to randomly sample $O_{c,\lambda}(\log n)$ block headers from a remote blockchain with variable block difficulty. We formally prove the security of our protocol as long as the computational power of the adversary is a $c < 1$ fraction of that of honest nodes.
- **Efficient Chain Commitments:** We formalize and use the notion of a *Merkle mountain range (MMR)* [49], an efficiently-updatable commitment mechanism that allows provers to commit to an entire blockchain with a small (constant-size) commitment while offering logarithmic block inclusion proofs with position binding guarantees.
- **Non-Interactive and Transferable Proofs:** We introduce a non-interactive variant of FlyClient using the Fiat-Shamir heuristic [26] that allows both the light client and the full node to forward the proof to other light clients without the need to recalculate the proof.

1.1 Overview of FlyClient

Consider a blockchain network that is growing a *valid* (or *honest*) chain C based on the most difficult chain principle, and a client (or *verifier*) who wants to verify that a given transaction *tx* is recorded on the chain. The valid chain is characterized as the chain with the highest cumulative computational difficulty created

[1] Ethereum also has a *fastsync* synchronization flag which allows a full node to sync to the current chain via SPV including downloading the current full state [7]. From this the node can start verifying all incoming transactions. Unfortunately, even *fastsync* can take upwards of 10 hours to receive all headers from nodes in the network, likely due to throttling by individual peers.

Block Height	**10 K**	**100 K**	**1,000 K**	**7,000 K**
SPV	4,961	49,609	496,094	3,472,656
FlyClient	154	261	389	484

Table 1: Comparison of proof sizes (in KB) for SPV clients and FlyClient in the Ethereum blockchain at various block heights assuming an adversary that has a hash power of at most 1/2 of the honest hash power and succeeds with probability less than 2^{-50}.

so far by the network. Any other chain is considered an invalid chain. The light client assumes that this valid chain follows all other rules of the network, such as containing only valid state transitions. For ease of explanation, we first assume each block has the same level of difficulty. In this model, the valid chain is the one with the highest length (*i.e.*, number of blocks). We will later formulate the problem using the variable block difficulty model to be consistent with most cryptocurrencies including Bitcoin and Ethereum.

The client is connected to a set of full nodes (or *provers*) at least one of which is honest (*i.e.*, holds a copy of the valid chain), but the client does not know which one is honest. The client and the provers participate in the FlyClient protocol to convince the client that tx is included in some valid block B on the honest chain.

Two Provers and a Verifier. We consider the case where the client is connected to only two provers one of which is honest. Kiayias et al.[36] presents a generic verifier that extends the two-party case to n parties.

If both provers agree that B is located at the same height on their chains, then the client is convinced and the protocol ends. However, if the provers do not agree, then one of them holds an invalid chain. In this case, the client challenges both provers with a *probabilistic sampling protocol* to find out which one holds the honest chain. Assuming that the combined hash power of all malicious miners is a $c < 1$ fraction of the honest miners, the probability that the adversary can mine the same number of blocks as the honest miners reduces exponentially as the honest chain grows. Thus, the adversary has to insert a sufficient number of invalid (or fake) blocks to make its chain as long, or more accurately, as difficult as the honest chain.

Probabilistic Sampling. Our probabilistic sampling protocol samples a logarithmic number of block headers from both chains using a probability density function $g(x)$ that specifies, for every height x in each chain, the likelihood that the block located at height x of the chain is sampled. Using differential analysis, we find the optimal $g(x)$ that maximizes the probability of catching the invalid chain given the adversary's optimal strategy. We can give a concrete lower bound on the probability of a single query catching the adversary, independent of the adversaries forking strategy. By repeatedly sampling we can reduce the adversaries success probability to a negligible probability.

Chain Commitments. The FlyClient protocol described so far still allows a malicious prover to deceive the verifier with an invalid chain. Namely, since the verifier downloads only a small number of block headers that are not necessarily chained together, the malicious prover can choose to only (or mostly) return correctly-mined block headers from the honest chain in response to the verifier's request, thus significantly decreasing the success probability of the probabilistic sampling protocol. One way to protect against such a strategy is to have the prover "commit" to its entire chain before the protocol starts, hence ensuring that the prover always returns the exact block headers at the correct positions on the chain.

To commit to the entire chain of blocks, FlyClient requires the prover to maintain a special type of a Merkle tree known as a Merkle mountain range (MMR) over all blocks added to the blockchain so far. In addition to being a Merkle tree, an MMR allows efficient appends at the prover side and efficient block inclusion verifications at the verifier side. Further it enables efficient subchain proofs, i.e. a proof that two MMRs agree on the first k leaves. At every block height i, the prover appends the hash of the previous block, B_{i-1}, to the most recent MMR and records the new MMR root, M_i, in the header of B_i (see Figure 8). As a result, each MMR root stored at every block height can be seen as a commitment to the blockchain at that specific height.

Putting Things Together. With MMR commitments in block headers, each prover begins by sending the header of the last block in its chain, *i.e.*the header of block B_n that includes the MMR root M_n. Next, the verifier samples a number of random blocks from the prover according to the probability distribution $g(x)$.

For every sampled block, the prover provides the corresponding block header and an *MMR proof* that the block is located at the correct height of the chain committed by B_n. Further the verifier checks the MMR root stored in each sampled Block, correctly commits to the correct subchain of the chain committed to in B_n. If the PoW solution or the MMR proofs of any of the sampled blocks is invalid, then the verifier rejects the proof. Otherwise, it accepts B_n as the last block of the honest chain. To ensure that tx is included in some block in the honest chain, the client proceeds similar to a regular SPV client, *i.e.*, verifies the Merkle proof provided by the prover against the root of the transaction Merkle tree included the block header along with another MMR proof that the block is in the MMR rooted at M_n.

Non-Interactive and Transferable FlyClient. To make our probabilistic sampling protocol non-interactive, we apply the Fiat-Shamir heuristic [26]. The randomness is generated from the hash of the head of the chain. The verifier now simply receives the full proof and checks that it is correct and that the randomness was derived correctly. The non-interactiveness makes FlyClient more practical since (1) the full nodes can send the same proof to many light clients without any recalculation; (2) the client can forward the proof to other new light clients and the new clients can safely verify the correctness of the proof. This reduces both the computation and bandwidth overheads for both the provers and the verifier.

2 Existing Solutions

Nakamoto [42] proposed an SPV protocol to verify Bitcoin transactions with minimum trust on some full nodes. Specifically, a light client only needs to download all block headers which are much smaller in size (*e.g.*, 80 byte against 1 MB block size in Bitcoin). A block header contains a hash of the Merkle root that commits all transactions in the block. Thus, after downloading all block headers in the blockchain, the light clients can efficiently verify the existence of any transaction in any block, given that the prover provides the Merkle proof of size $\log s$ hashes, in which s is the number of transactions in the block. Unfortunately, the SPV client still needs to download data that is linear in the length of the blockchain, *i.e.*, 1 header per block. For example, Ethereum produces 1 block every 6 seconds and the Ethereum chain has over 7 million blocks. An SPV client, therefore, needs to download 3.56 GB and additionally sync 7.3 MB per day. airforceblue

The idea of implementing SPV clients with sub-linear complexity using skip lists was first proposed in 2012 [48]. Subsequently, Kiayias *et al.* improved and formalized the proposal [35, 36]. In [35], they propose an SPV protocol, called *proofs of proof-of-work (PoPoW)*, which reduces the required resources of any SPV client to a logarithmic number of blocks. The protocol is based on the observation that a certain number of lucky blocks, known as *superblocks*, are expected to exist in a PoW chain if it has been created honestly.[1] Informally speaking, a superblock is a rare block that its PoW output value (*i.e.*, the block ID) has more leading zeros than other valid blocks, and hence, can be used to show that enough work has been done when the chain containing that block was created. It can be shown that, by verifying the validity of only a logarithmic number of superblocks, a light client can ensure the validity of the entire chain it receives from a full node with high probability. Inspired by the skip list data structure, PoPoW changes the blockchain structure in such a way that each block, instead of having only one reference to the immediate previous block, stores multiple references to previous blocks including the superblocks.

Each transaction inclusion proof in PoPoW is increased by $m \log n \log\log n$ in size to prove that the blocks that contains the transaction belongs to the correct chain, where m is the security parameter. Furthermore, PoPoW increases the size of each proof by a $\log n$ factor, where n is the number of blocks in the blockchain. alizarin In addition, the protocol is interactive, meaning that the verifier has to communicate over multiple sequential rounds with the verifier to obtain a validity proof. This incurs a high latency and communication cost for both the light client and the full node.

In a later work, Kiayias *et al.* [36] present an attack against PoPoW, where the adversary can double-spend bitcoins even if it controls only a minority of the hashing power. They also propose a *non-interactive PoPoW (NIPoPoW)* protocol that allows succinct (*i.e.*, logarithmic-size) proofs but with the same proof complexity as in [35]. Unfortunately, both PoPoW and NIPoPoW assume a fixed difficulty for all blocks which does not hold in practice in Bitcoin and Ethereum. Moreover, both protocols are vulnerable to a *bribing attack*, where an attacker offers an incentive to those miners who will be lucky to find superblocks,

[1] This suggestion was first proposed in a post in the Bitcoin forum [48].

in exchange for not publishing their blocks to the network. Such an attack is possible in any protocol that differentiates between mined blocks in a deterministic way, because the adversary knows in advance the type of blocks that it is willing to bribe so it can advertise them before they are mined and published to the network. As a defense against such an attack NiPoPoW reverts to the standard SPV protocol if not enough super blocks exists. This means that NiPoPoW is only efficient if no adversary exist.

FlyClient is not vulnerable to such bribing attacks, because it does not differentiate between blocks in any way *before* they are mined. The set of "lucky" blocks selected to serve as a proof to the light client are determined only after those blocks are mined via a randomness chosen by the client during transaction verification. Therefore, the adversary will not be able to know ahead of time which blocks they will need to provide to the verifier.

airforceblue airforceblue

3 Model and Problem Definition

Network Model. Consider a blockchain network that is growing a chain based on the most difficult chain rule, and a client who is connected to two full nodes at least one of which is honest (*i.e.*, holds a copy of the valid chain), but the client does not know which one is honest. Given a protocol for distinguishing between the valid and the invalid chain, Kiayias et al.[36] presented a generic verifier that can determine the valid chain from a larger set of nodes. We, therefore, focus on the simpler case of two parties. **Threat Model.**

We consider an adaptive (or rushing) adversary who can choose which full nodes to corrupt and which blocks to "fake" in the blockchain. However, the adversary's mining power is always bounded by a known fraction $0 < c < 1$ of the combined mining power of honest nodes.

Our assumption that the client is connected to at least one full node is equivalent to assuming the client is not vulnerable to eclipse attacks [32]. Defending against such attacks is orthogonal to this work and has been addressed by recent papers [32, 28]. We also assume that the adversary cannot drop or tamper the messages between the client and the full nodes. The light client is not assumed to know any block or state in the chain, except the genesis(*i.e.*, the first) block.

Problem Definition. The full nodes want to convince the light client that they hold a copy of the valid chain. The light client ultimately wants to verify that a transaction is included in the valid chain.

4 Preliminaries

A key component of our protocol is a type of Merkle hash tree [41] which allows every block to commit to all previous blocks. This enables efficient inclusion proofs that a particular block is part of the blockchain. Similar to vector commitments [21], Merkle trees provide *position binding*, which is that a malicious prover cannot open a commitment to two different values at the same position of a committed sequence. In FlyClient, we use an MMR construction which is a Merkle tree with an efficient append functionality. An MMR further allows a prover to efficiently convince a verifier that two MMRs share the same subtree (see Figure 1). We will discuss these properties in Section 5.2.

Before defining the preliminaries, we establish our notation and terminology used throughout the paper.

Notation and Terminology. Let n denote the blockchain length which is the number of blocks in the blockchain at the time of proof generation and verification. Also, let $c < n/3$ denote the total computational power of the adversary. We say an event occurs *with high probability* if it occurs with probability $1-O(1/2^{\lambda})$, where λ is the security parameter.

Definition 1 (Collision resistant hash function)**.** *A family of hash functions* $\mathsf{H}_{\lambda} : \{0,1\}^* \rightarrow \{0,1\}^{\lambda}$ *is collision resistant if for all efficient adversaries* $\mathcal{A}$ *the probability that* $x, y \overset{?}{\leftarrow} \mathcal{A}(1^{\lambda})$ *and* $\mathsf{H}(x) = \mathsf{H}(y) \wedge x \neq y$ *is negligible in* λ*.*

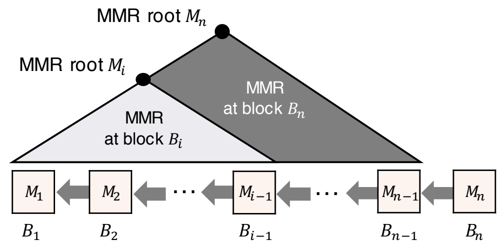

Figure 1: MMR subtree inclusion

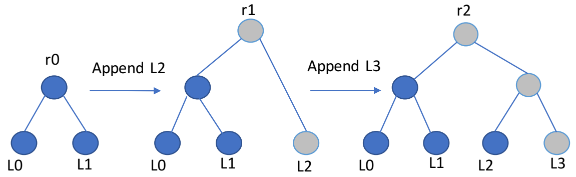

Figure 2: Example of updating a MMR tree when new data entries are appended as new leaves of the tree. The grey nodes are either new nodes or nodes that are changed due to the new data entry. MMR guarantees that for every update, only $\log n$ nodes are either created or modified.

4.1 Merkle Mountain Ranges

A Merkle tree is a balanced binary tree where the value of each non-leaf node is the hash of its children. Merkle trees allow logarithmic, in the size of the tree, size proofs that a node is contained in the tree. We refer to Appendix A for a precise definition of Merkle tress and Merkle proofs as well as their security guarantees.

Todd [49] proposed MMRs as part of a distributed time-stamping service using Bitcoin. The idea was around even before in the context of certificate transparency logs [16] to show that any particular version of an append-only log is a superset of any previous version. In FlyClient, we use MMRs to commit to previous blocks efficiently using a single hash value written in every block header. One may wonder if the original Merkle tree structure can be used to achieve the same goal. In fact, appending a Merkle tree with new block headers as the leaves is unspecified: it only specifies how a hash tree can be constructed from a sequence of hashes/elements. To append a new element to the sequence, one has to either create a new Merkle tree from scratch or suffer an "unbalanced" tree that can result in proof sizes that are much larger than logarithmic-size proofs from a perfectly-balanced Merkle tree.

An MMR is a Merkle tree variant that guarantees the tree is reasonably balanced even when new elements are appended, without rebuilding the tree from scratch. This is done by modifying only a few nodes of the existing tree and still bounding the length of the Merkle proof for any element (sitting as a leaf) by $\log n$, where n is the number of leaves. In FlyClient, this allows a much more efficient update process resulting in only a small overhead for full nodes when adding new blocks to the chain. Moreover, adding an MMR root (*e.g.*, a SHA-256 hash) to the block header requires only a small modification to the current Bitcoin and Ethereum protocol.

We give an example of updating a MMR tree by appending new data entries to the leaves in Figure 2. The append function of MMRs additionally give the ability to prove that an MMR is the previous version of another MMR with a short proof. That is, given the k-th MMR and the n-th MMR the prover can give a proof of size $O(\log(n))$ that convinces a verifier of this fact while the only previous information the verifier has is k, n, and the root of the k-th and n-th MMRs. We formalize this important property in Theorem 8 of Appendix B.

5 FlyClient Design

In cryptocurrencies like Bitcoin and Ethereum, the valid chain is the one that requires more computational work to create, and hence is the one that has highest total block difficulty. While the most difficult chain rule is the accurate way of determining which chain is the valid one, the notion of the longest chain provides a simplified way that makes protocol analysis easier. Therefore in this section, we assume that all blocks have the same difficulty. Later in Section 6.5, we extend our protocol to handle the more realistic scenario where blocks have variable difficulty throughout the chain.

5.1 Design Components

FlyClient consists of three main building blocks. First, we leverage a new data structure called a Merkle Mountain Range (MMR) to allow for verification of any previous block header with only the latest block header. Once a block is verified, we can then verify any transaction included in that block with simple SPV Merkle proofs.

Second, in order to reduce the number of block headers that light clients need to download to verify the latest block header, FlyClient employs a probabilistic verification mechanism by which it randomly samples only a logarithmic number of block headers; if these block headers are valid, then we show that the latest block belongs to the longest chain with high probability. Which block headers to sample are chosen by the light clients to prevent the adversary from avoiding sampling fake blocks. If an adversarial prover is trying to convince the verifier that they know a chain that is the same length as the honest chain, there is a maximal number of blocks in the adversary's chain which are valid (*i.e.*, have a valid PoW since the adversary has limited computing power). Our probabilistic verification guarantees that after randomly sampling enough number of blocks, we can detect at least one invalid block in the adversary's chain with high probability. In Section 6, we build up our sampling protocol by describing different straw-man approaches the verifier can use to sample blocks, until finally we calculate our optimal sampling protocol in Section 6.4. We outline our interactive protocol in Protocol 1.

Our third building block is using the Fiat-Shamir heuristic to remove the interaction between the light clients and the full nodes. Specifically, a full node can figure out locally which random blocks it should send to the clients for the verification without any initial randomness from the light client, yet the light client can verify the correctness of the proof and is guaranteed that the full node is not cheating. We discuss in detail how to make FlyClient non-interactive in Section 6.6.

Algorithm 1 FlyClient Protocol

A client (*i.e.*, the verifier) performs the following steps speaking with two provers who want to convince the verifier that they hold a valid chain of length $n + 1$. At least one of the provers is honest.

1. Both provers send to the verifier their last block headers in their chains. Each header includes a commitment value representing the root of an MMR created over the first n blocks of the corresponding chain.

2. For $j \leq O(\log n)$ rounds:

   (a) The verifier queries k random block headers from each prover based on the probabilistic sampling algorithm described in Section 6.

   (b) For each queried block, B_i, located at position i of either chain C, the prover sends to the verifier the header of B_i along with an MMR proof $\Pi_{B_i \in C}$ that B_i is the i-th block in C.

   (c) The verifier checks the PoW for each block header as well as the validity of $\Pi_{B_i \in C}$. If any of the checks fails, the verifier rejects the prover.

3. If the prover has not been rejected, the verifier accepts C as the valid chain.

5.2 Block Inclusion Verification

Algorithm 2 Prover/Verifier protocol for a single query

The verifier queries the prover for the header and MMR proof for a single block k in the prover's chain of $n+1$ blocks.

```
Verifier
Has the root of the MMR of $n$ blocks stored in the $n+1$ block's header
Queries prover for the header of block $k$ and for $\Pi_{k \in n}$
Verifies the hashes of $\Pi_{k \in n}$ hash up to the root of $\mathrm{MMR}_n$
Calculates the root of the MMR of $k-1$ blocks from $\Pi_{k \in n}$ by calling Get_Root( $\Pi_{k \in n}$, $n$, $k$)
Compares the calculated root with the root in the header of block $k$
If everything checks out, accepts block proof

Prover
Has chain of $n+1$ blocks and the MMR of the first $n$ blocks
Receives query for block $k$ from verifier
Calculates $\Pi_{k \in n}$ from $\mathrm{MMR}_n$ by calling MMR_Proof( $\mathrm{MMR}_n$, $n$, $k$)
Sends header of $k$ and $\Pi_{k \in n}$ to verifier
```

As discussed before, the need to download all block headers is in part due to the need for verifying transactions/ events in previous blocks. Assuming the longest chain has been verified and accepted with only some of the block headers downloaded, *i.e.*, the verifier knows some C is the last block header in the longest chain, verification of a transaction in some previous block requires checking if the block actually belongs to a chain ending in C. The naive approach is to download all intermediate block headers from the block to C, which inherently requires downloading a linear number of block headers from the chain. Once the block is verified to belong to the chain, the verifier needs only an SPV Merkle proof that a transaction is in that block.

Our goal is to allow the verification of any block (*i.e.*, obtain $\pi_{\mathsf{rec}(tx)}$) with only the latest block header in the chain. There exist simple solutions to achieving the same goal. For example, one can build a global Merkle tree on all transactions in the blockchain and update this Merkle tree after every block. However, such a solution requires miners to maintain all transactions on the chain, which they often do not do for performance reasons. The block verification also requires full nodes to obtain all transactions and to reconstruct the Merkle tree from scratch (to keep the tree balanced). Thus, such solutions are expensive and not practical for a real-world deployment.

Our solution leverages the recently introduced data structure called *Merkle mountain range (MMR)* [49] to commit to all previous block headers in the latest block. Having this commitment allows the verifier to efficiently validate that a previous block belongs to the longest chain based on the latest block header of the chain. **Thus, the full node can prove that a transaction was included in the longest chain by just providing an MMR proof (to prove that a block belongs to the longest chain) in addition to the current transaction proof (which shows that the transaction is included in the said block).** Protocol 2 describes how a verifier can query a prover for the validity of a single block.

New Block Header. Our new block header now contains one extra field namely the MMR root, *i.e.*, the root of the MMR tree that commits the headers of all previous blocks. This requires a minimal change to the current block structure of Bitcoin and Ethereum, and can be implemented as a soft fork in both of these networks. We discuss this in more details in Section 8.2. A full node, upon receiving a new block, will conduct only one additional check on the validity of the MMR root. This entails a negligible overhead on the full node as we report in Section 7.

6 Proof of Honest Chain

Our goal is to have a protocol that allows an honest prover to convince the verifier of the validity of its chain while a malicious prover cannot convince the verifier of a dishonest chain. In the previous section, we outlined the basic FlyClient protocol, what is left to be determined is how the verifier samples blocks from the prover. In this section, we describe some strawman approaches for our sampling protocol and build up the properties we wish to satisfy. We start with a simple sampling protocol which gives us the desired properties and show how to optimize our protocol to achieve smaller proof sizes.

6.1 Naive Approach

One approach is for the verifier to request a uniformly-random set of multiple blocks from each prover. Since the malicious prover has only a limited computation power, it can at best correctly mine a subset of all the blocks. Thus, the verifier needs to sample enough blocks to ensure that at least one of them is invalid, *i.e.*, an incorrectly-mined block. The protocol begins with each prover giving the verifier the header of the last block in its chain, where this header contains the root of an MMR tree built over all blocks in the chain. Whenever the verifier requests a block from each prover, the prover must also provide a Merkle proof that the block is a leaf in the MMR of the last block. From the MMR inclusion proof, the verifier can recreate the MMR root for that block and verify that it is the same root in the header of the block (therefore included in the proof of work for the block).

As shown in Corollary 2, once a malicious prover forks off from the honest chain, it cannot include any of the later honest blocks in its chain because the MMR root in those blocks would not match the chain. With this setup, if the verifier makes enough queries, it will eventually ask the malicious prover for a block it has not mined (*i.e.*, an invalid block).

To determine how many blocks the verifier must query to achieve a desired probability of success in catching a malicious prover, we bound the malicious computing power to a c fraction of the honest computing power. After the adversary forks from the honest chain, it can correctly mine up to only a c fraction of the blocks in the rest of the chain. So, if we know that the adversary forked at some block B_a, then for each random block the verifier requests after B_a, there is a probability of $(1-c)$ that the sampled block is invalid (*i.e.*, incorrectly mined) as the adversary has to "lengthen" its fork to have a chain of equal length to the honest chain. Thus, with k queries after the fork point, the verifier has a success probability of $(1-c^k)$ in catching the malicious prover. The value k can be chosen arbitrarily (*e.g.*, a constant) to bound the protocol's failure probability according to the security parameter λ.

Solution Limitation. Since the verifier does not know where in the chain the adversary started the fork, the verifier has to sample a large number of blocks to increase its chance of catching the malicious prover, especially if the fork point is located near the end of the chain, *i.e.*, the fork is short. Can the verifier sample a smaller number of blocks in such a way that it can find the fork point a?

6.2 Binary Search Approach

Since the verifier knows at least one of the provers is honest, it can *search* for the fork point by querying both provers at the same time to find the first block at which they disagree. The verifier can do this by performing a binary search over the entire chain as follows. The verifier starts by asking each of the two provers to send the block in the middle of its chain. If the two middle blocks are the same, then the verifier recurses on the second half (*i.e.*, the one with higher block numbers), otherwise it recurses on the first half. This is repeated until the verifier finds the fork point which is the smallest block number where the two provers disagree. Once the verifier finds the fork point, it samples blocks randomly from both provers after the fork point, with each sample having a probability of $(1-c)$ in catching the dishonest prover in a lie. Thus, the verifier must sample $2\log n$ blocks to find the fork point (n being the chain length) plus $2k$ blocks to have a probability of $(1-c^k)$ in determining which of the two provers is dishonest, if one is.

Solution Limitation. This approach is inherently interactive and requires multiple rounds of communication between the verifier and the provers, resulting in higher verification latencies. Can we find the fork point by sampling blocks from each prover in a single shot while still achieve a high probability of success?

6.3 Bounding the Fork Point

Finding the exact location of the fork point by sampling a small number of blocks in only one shot is challenging. We instead relax this requirement and allow the verifier to only "bound" the proximity where the fork point is located while still sampling in one shot. Our goal is to ensure that the verifier makes sufficient queries after the fork point. Instead of searching for the fork point, the verifier can iterate through intervals from which it sample blocks. If in at least one of the intervals the verifier has a sufficiently-high probability of catching the malicious prover, then the verifier succeeds with high probability in the whole protocol.

For each prover, the new sampling protocol first samples k random blocks from the entire chain. Then, it successively splits the chain (or the current interval) in half and queries another random k blocks from the last half, *i.e.*, the interval always ends with the tip of the chain. More precisely, for every integer $j \in [0, \log n)$, the verifier queries k blocks from the last $n/2^j$ blocks of the chain. This is repeated until the size of the interval is at most k, *i.e.*, all last k blocks are sampled.

We now show that the above strategy succeeds with high probability. To do this, we calculate the probability that the verifier samples at least one invalid block from the malicious prover, based on the observation that the adversary has to insert a sufficient number of invalid blocks into its fork to obtain an overall chain of equal length to the honest chain.

Lemma 1. *The probability that the verifier fails to sample any invalid block is at most* $\left(\frac{1+c}{2}\right)^k$.

Proof. Let n denote the length of the chain (not counting block $n+1$ which the verifier has already sampled) and c denote the fraction of the adversary's computing power relative to the honest computing power. At any interval j, the verifier samples from the interval between block $\frac{(2^j-1)n}{2^j}$ and n. Let h_j denote the number of invalid blocks the adversary has inserted in the j-th interval. The probability that the verifier fails to sample an invalid block in this interval is

$$P_j = \left(\frac{\frac{n}{2^j} - h_j}{\frac{n}{2^j}}\right)^k = \left(\frac{n - 2^j h_j}{n}\right)^k .$$

Thus, the probability that the verifier fails is $\prod_{j=0}^{\log n} P_j$. Since $P_j \leq 1$, if one P_j is sufficiently small, then the total probability of failure is also sufficiently small.

Letting a denote the forking point, there is some integer j such that $\frac{(2^j-1)n}{2^j} \leq a < \frac{(2^{j+1}-1)n}{2^{j+1}}$. In other words, there is some sampled interval of size $n' = n/2^j$ in the protocol where the fork point lies between the start and the middle of the interval. Let l denote the length from a till n, *i.e.*, the length of the fork, $l > \frac{n'}{2}$. The number of invalid blocks in the interval is $h_j = (1-c)l \geq (1-c)\frac{n'}{2}$. Thus, the probability that the verifier fails to catch the invalid chain is at most equal to the probability that the verifier fails at step j, *i.e.*,

$$\Pr[\text{fail}] \leq \Pr[\text{fail at } j] \leq \left(\frac{n' - (1-c)\frac{n'}{2}}{n'}\right)^k = \left(\frac{1+c}{2}\right)^k .$$

Note that if $l \leq k$, the verifier will sample all of the adversary's invalid blocks and $\Pr[\text{fail}] = 0$. □

Solution Limitation. In our analysis, we calculate the probability of success based on the likelihood of success in at least one of the $\log n$ intervals. However, our protocol samples other blocks that we do not consider in our analysis, but could increase the verifier's success probability. Can we achieve a better bound by further taking these blocks into account?

6.4 Distributional View

While we presented the protocol of Section 6.3 as an iterative protocol, it is important to note that all of its steps are independent. That is the verifier's samples do not depend on the prover's responses to previous queries. This implies that the order of samples can be altered to create an isomorphic protocol with the same security and efficiency properties. We can further use this to examine the probability that a given block is sampled. Clearly, the protocol samples later blocks with higher probability. In other words, the probability grows inversely with the relative distance of a block to the end of the chain (*i.e.*, the most recent block).

We can use these probabilities to determine the probability distribution of the following protocol (which is displayed in Figure 3 as $s(x)$): The verifier picks one of the intervals uniformly at random (from the protocol presented in Section 6.3) and samples a block uniformly at random from that interval.

Consider a protocol that simply repeats the sampling steps q times. If the adversary is caught with probability at least p given one sample, then he will be caught with probability at least $1-(1-p)^q$ after q independently and identically-distributed samples. This distributional approach enables a simple analysis of the protocol as we only need to bound the success probability of a single query. Furthermore, it allows us to optimize the protocol by finding a query distribution that maximizes p. As shown in Figure **??**, the distribution introduced by the protocol from Section 6.3 is not smooth. In the following, we show that a different and smoother distribution performs better.

Optimizing the Sampling Distribution. We now find the optimal sampling distribution, that is the sampling distribution over the blocks, which maximizes the probability of catching the adversary given that it chooses the optimal strategy. We do this by finding the sampling distribution that maximizes the probability of catching the adversary with only a single query. Given this probability, we can directly bound the adversary's success probability after q queries. As a simplifying assumption, we treat the number of blocks as a continuous space between 0 and 1. That is, the block header is at 1 and the genesis block is at 0. We later show that this simplified analysis still produces a good distribution for the discrete case.

As a first step, we show that the probability density function (pdf) of the optimal sampling distribution must be increasing. A pdf f defined over the continuous range $[0,1]$ is increasing if, for all $a,b \in [0,1]$, $b > a \implies f(b) \geq f(a)$. For any distribution defined by a pdf that is not increasing, there exists a distribution that results in an equal or greater probability of catching the adversary.

Lemma 2 (Non-increasing sampling distribution)**.** *A sampling distribution over the blocks defined by a non-increasing pdf f is not uniquely optimal, i.e., there exists another distribution with equal or higher probability of catching the adversary.*

Proof. We prove the statement by contradiction. We show that given f, there exists another pdf f' that with a single query succeeds in catching a given adversary with slightly higher probability.

Given that f is non-increasing, there exist numbers $x_1, x_2, d \in [0,1]$ and intervals $I_1 = [x_1, x_1+d]$ and $I_2 = [x_2, x_2+d]$ such that $x_1 + d \leq x_2 \leq 1-d$ and $f(x) > f(x')$, for all $x \in I_1 \wedge x' \in I_2$. Any adversarial strategy can be defined by a fork point $a \in [0,1]$ and by the ranges of blocks which are invalid after a. Note that given a fork point, the adversary can freely decide which blocks, *i.e.*, which intervals, to make invalid and which ones to honestly mine. For any strategy which produces an invalid block in I_1 but valid blocks in I_2 there exists a strategy which creates an additional invalid block in I_2 and one more valid block in I_1 without changing any other part of the strategy. Note that the converse is not true. If the fork point $a > x_1$ then it may not be possible to move invalid blocks to the first interval. Given that the querying probability of any point in I_2 is lower than the probability of any point in I_1 the adversary is always better off by moving all possible invalid blocks to I_2. I_2 must therefore contain no less invalid blocks than I_1 in any strategy which is optimal for the adversary.

Consider the probability distribution f' which is equal to f on all points but $x \in I_1 \cup I_2$. There exists an $\epsilon > 0$ such that for any point $x \in I_2$, $f'(x) = f(x) + \epsilon$ and for any point in I_1, $f'(x) = f(x) - \epsilon$ and the following condition holds: For all adversaries, a single query drawn from the distribution defined by f' has a slightly higher probability of querying an invalid block than a single query drawn from the distribution defined by f. This is because f' queries with higher probability in I_2 which must contain no less invalid blocks than I_1 for any optimal adversary. □

Since all non-increasing distributions yield a non-unique optimal sampling distribution, we can focus our search on sampling distributions defined by increasing pdfs that sample later blocks with higher probability than earlier blocks. For such distributions, if the adversary forks off from the main chain at some point $0 \leq a < 1$, the adversary's best strategy is to put all of its correctly-mined (*i.e.*, valid) blocks at the end of its chain so they are the most likely to be sampled. If the adversary has a c fraction of the honest mining power, and $1-a$ is the length of the adversary's fork, then the adversary can mine a $(1-a)c$ fraction of the chain. Thus, in its best strategy, the section of the adversary's chain from a to $1-(1-a)c$ does not contain valid blocks.

To catch the malicious prover, we must sample a block in this interval. Hence, the probability that we catch an adversary who forks at some point a with one sample is $\frac{\int_a^{1+ac-c} f(x)dx}{\int_0^1 f(x)dx}$, where $f(x)$ is proportional to the probability density function of the sampling distribution. Considering all points where the adversary

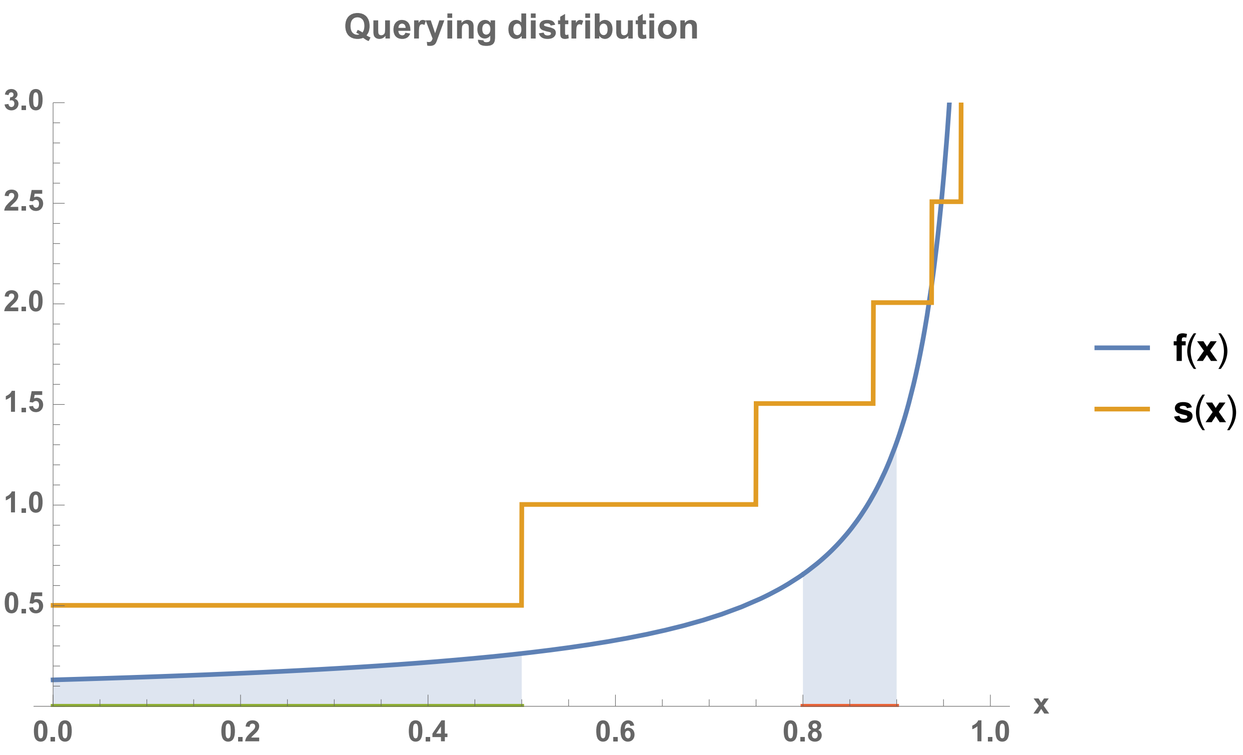

Figure 3: $s(x)$ defines the probability density function (pdf) for the protocol from Section 6.3. $g(x) := \frac{1}{(x-1)\ln(\delta)}$ is the optimized pdf. The integral $\int_a^{1+ac-c} g(x)dx$ for $c = 1/2$, $\delta = 2^{-10}$, $a = 0$ and $a = 0.8$ respectively is displayed.

could fork from, the probability we catch the him is

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

In order to find the optimal protocol, we have to find the distribution that maximizes this quantity. Intuitively, we want a sampling distribution which makes the adversary indifferent about which fork point to use. Otherwise, queries would be wasted on blocks which an optimal adversary would not make invalid anyway. Concretely, we find an $f(x)$ that satisfies

$$\int_0^{1-c} f(x)dx = \int_a^{1+ac-c} f(x)dx.$$

In other words, if the adversary forked from the beginning of the chain or any other point, we have the same probability of catching it.

Through differential analysis, we find that

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

satisfies this condition, *i.e.*, $\int_a^{1+ac-c} f(x)dx = \frac{(c-1)\ln(c)}{c}$. In Figure 3, $f(x)$ and this property is displayed visually.

We are now interested in analyzing how close our $f(x)$ is to the optimal sampling distribution. We first try to compute the normalized probability density function by normalizing $f(x)$ by an $\int_0^1 f(x)dx$ factor.

Unfortunately, $f(x)$ goes to infinity as x approaches 1 and $\int_0^1 f(x)dx = \infty$. Luckily, we can restrict the querying domain from 0 to $1-\delta$ and always check the final δ fraction of the blocks directly. We will later find the optimal value for δ. Let

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

The probability of catching the adversary is equal to

$$p = \min_{0 \le a \le \frac{c-d}{c}} \int_a^{1+ac-c} g(x)dx = \int_0^{1-c} \frac{1}{(x-1)\ln(\delta)}dx = \log_\delta(c).$$

This probability takes into account that the last δ blocks are *manually* checked by the protocol with probability 1. Any fork after $\frac{c-d}{c}$ will contain a manually-checked block, and thus be caught with probability 1.

We will now show that $g(x)$ defines an optimal sampling distribution by showing that no sampling distribution can achieve a higher p value, *i.e.*, a higher probability of catching the adversary with a single query. Note that the sampling strategy is optimal for an optimal adversary. The optimal adversary can adapt the placement of invalid blocks *after* learning the querying strategy.

Theorem 1 (Optimal sampling distribution)**.** *Given that the last* $\delta = c^k \in (0,1], c \in \mathbb{N}$ *fraction of the chain contains only valid blocks and the adversary can at most create a* c *fraction of valid blocks after the fork point* a*, the sampling distribution defined by the pdf* $g(x) = \frac{1}{(x-1)\ln(\delta)}$ *maximizes the probability of catching any adversary that optimizes the placement of invalid blocks.*

Proof. Let $\delta = c^k$, for some $k \in \mathbb{N}$, we get that $p = \frac{1}{k}$ and that as k increases the success probability decreases. Hence, the smaller δ is set, the fewer the blocks that are always checked near the tip of the chain but the worse our probability of catching the adversary with a sample anywhere else. Therefore, a smaller δ leads to more samples from the rest of the chain.

Say $g^*(x)$ is the probability density function of the *best* sampling distribution. Note that given Lemma 2, $g^*(x)$ is increasing and therefore for an optimal adversary the success probability is denoted by

$$p^* = \min_{a, 0 \le a \le \frac{c-d}{c}} \int_a^{1+ac-c} g^*(x)dx.$$

$g^*(x)$, therefore, maximizes p^*. The optimality condition implies that $\int_{1-c^i}^{1-c^{i+1}} g^*(x)dx \ge p^*$, for all integer $i \in [0,k]$, where $a = 1 - c^i$ is a possible forking point. Further we have that $\int_0^{1-c^k} g^*(x)dx = 1$ since $g^*(x)$ is a pdf. We have $\int_0^{1-c^k} g^*(x)dx = \sum_{i=0}^k \int_{1-c^i}^{1-c^{i+1}} g^*(x)dx = 1 \ge k \cdot p^*$. This implies that $p^* \le \frac{1}{k}$. Note that $g(x)$ as a candidate distribution achieves $p = \frac{1}{k}$ and is, therefore, optimal.

□

Optimizing the Proof Size. Given $g(x)$ and p, we can now define $p_m = (1-\frac{1}{k})^m$ as the probability of not catching the optimal adversary after m independent queries. Note that w.l.o.g., $k \ge 1$ as otherwise $\delta > c$, implying that a sufficient fraction of blocks are checked to catch any adversary. If we want $p_m \le 2^{-\lambda}$, then

$$m \ge \frac{\lambda}{\log_{1/2}\left(1-\frac{1}{k}\right)}$$

. Now, assume that the verifier always checks L final blocks, $\delta n = c^k n = L$, thus $k = \log_c(\frac{L}{n})$ and $m \ge \frac{\lambda}{\log_{1/2}\left(1-\frac{1}{\log_c(\frac{L}{n})}\right)}$.

This means that m approximates $\lambda \log_c(\frac{1}{2})\ln(n)$, *i.e.*, $\lim_{n->\infty} \frac{m}{\lambda \log_c(\frac{1}{2})\ln(n)} = 1$. As long as L is a constant, *e.g.*, 1, the number of queries are linear in the security parameter λ and logarithmic in the size of the blockchain n.

Despite the already-good asymptotic, we can further find the optimal L that minimizes the proof size. Let B denote the number of hashes per block header. The proof size is approximately proportional to $|\pi| = m \cdot (B + \log_2 n) + L \cdot B$ (the average inclusion proof consists of $\log_2 n$ hashes). We now want to find $L' = \arg\min_L |\pi|$, *i.e.*, the value of L for which $|\pi|$ attains its minimum. While it is difficult to analytically minimize $m \cdot (B + \log_2 n) + L \cdot B$, we can numerically find the optimal L. Alternatively, we can approximate $m \cdot (B + \log_2 n) + L \cdot B$. In particular, let

$$h(n) = \lambda(B \ln 2 + \ln n) \log_c \left(\frac{L}{n}\right) - \frac{1}{2}\lambda(B \log 2 + \log n) + BL + \frac{1}{12}\lambda \ln c.$$

Then, $\lim_{n\to\infty}(|\pi| - h(n)) = 0$, *i.e.*, in the limit $h(n)$ perfectly approximates $|\pi|$. Since $h(n)$ is analytically simpler to minimize, we can find $L' = \arg\min_L h(n) = l \log_{\frac{1}{c}}(2n^{\frac{1}{B}})$. Plugging L' into $|\pi|$, we get that $|\pi| = \Theta(-\lambda \log(n) \log_c(n))$.
For realistic Ethereum values of $\lambda = 50, n = 2^{22}, c = \frac{1}{2}, B = 16$, this leads to a proof size of 660 KB. Note that the real proof size for 4 million blocks in Ethereum is a bit smaller at less than 400KB (See Section 7). This is because not all blocks have the same difficulty and later blocks have higher difficulty leading to better values for δ.

6.5 Handling Variable Difficulty

So far, we have only considered the unrealistic case that all blocks have the same difficulty. The distributional view allows us to directly handle variable difficulty as well. We simply use the same sampling distribution

$$g(x) = \frac{1}{(x-1)\ln(\delta)},$$

but now x indicates the relative aggregate weight and δ the relative weight of the blocks which are queried with probability 1. For example $x = 1/2$ is the point in the chain where half of the difficulty has been amassed. And $g(1/2)$ is the probability density at that point. Note that $x = \frac{1}{2}$ may be a recent block for chains with high difficulty growth. Given a variable difficulty can we still provide meaningful bounds on the proof size? It turns out that we can. Note that each block contains at least $1/2^\lambda$ of the total difficulty. This follows from the total difficulty being bounded by 2^λ and each block having at least difficulty 1. Even still the proof size grows as $\lambda^2 \log_c(\frac{1}{2})(B \ln(2) + \ln(n))$ as λ goes to infinity. For the same worst case conditions and Ethereum parameters $c = 0.5, n = 2^{22}, \lambda = 50, B = 16$ the estimated proof size is still just 1.85 MB. This is an upper bound for the proof size at these parameters. This analysis shows that FlyClient is efficient even under the worst possible difficulty distribution.

Sampling Based on Difficulty. An obvious question is how FlyClient can quickly verify which block has which total amassed difficulty. To do this, we slightly amend the MMR chain commitment as depicted in Figure 4. Each node in the tree now additionally contains the aggregate weight of all leaves below it. The root, therefore contains the total difficulty. Note that it is simple to check whether an inclusion proof is internally consistent. For every node, the two children's weights should sum up to the nodes weight. In every Merkle tree inclusion proof, both children are provided for every internal node. Each header already contains the total aggregated difficulty up to that block. The verifier checks that the provided node is indeed at the x-th percentile of the total weight. The check is done as follows.

- Let Π_k be the Merkle proof for the k-th node which is claimed to be at the x-th weight percentile.
- Let d_k be the difficulty of the node k.
- Ensure that node k indeed satisfies difficulty d_k.
- Verify the Merkle proof and ensure that each node's difficulty is positive and the sum of its children's difficulty.
- Let d be the total difficulty of all left nodes in Π_k.
- Let t be the total difficulty indicated in the root
- Ensure that $d/t \leq x$ and that $\frac{d+d_k}{t} > x$.

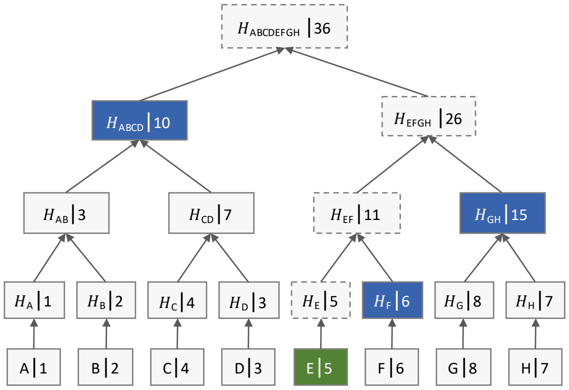

Figure 4: Merkle tree committing to aggregate difficulty

6.6 Non-Interactive FlyClient

Since our probabilistic verification relies on a randomness for the sampling, one trivial solution is to ask the light client to send the randomness to the full node. The full node then uses the randomness to sample k blocks and sends them back to the light client. This prevents the full node from biasing the sampled blocks and avoiding the detection of invalid blocks. However, this mechanism requires interaction between the light client and the full node. Moreover, the light client and the full node cannot forward the proof to other light clients as they cannot prove that the randomness is actually random. In this section, we introduce a mechanism to make FlyClient non-interactive by removing the randomness exchange step between the light client and the full node.

Our FlyClient protocol described so far is an interactive public-coin protocol [30], since the verifier's messages are chosen randomly from some known probability distribution. Concretely, in FlyClient these messages are block numbers in some predefined intervals. Fiat and Shamir [26], show that it possible to turn any interactive public-coin protocol into a non-interactive protocol which is sound in the random oracle model. To achieve this, every message of the verifier is replaced by the result of a query to a random oracle H, which in practice, is represented by a hash function such as SHA-3. H is queried at the current transcript and the oracle's answer is mapped into the verifier's message space. Concretely, the queries would be computed by applying H to the block header.

The Fiat-Shamir heuristic turns the statistical soundness of Protocol 1 into computational soundness as a prover can receive new samples by recomputing the final block header. On the other hand, recomputing the final header requires solving a new PoW puzzle, which itself requires a high number of queries to the hash function. In fact, our security assumption gives a concrete bound on the number of PoW puzzles the adversary can solve, which is $c \cdot n$. Let p_m be the soundness of Protocol 1 and $2^{-\lambda}$ be the desired failure probability (*e.g.*, 2^{-50}). Using the union bound, we conclude that the non-interactive FlyClient is secure as long as $p_m < \frac{2^{-\lambda}}{c \cdot n}$.

Transferable and Unique Proofs. A major benefit of the non-interactive proofs is that they are transferable. A single prover can produce a proof and other users can relay the proof without any additional computation. The relayed proof is still convincing to a verifier. A full node, therefore, can create a proof which many other clients can use. Moreover, by applying the Fiat-Shamir heuristic to the head of the chain we enforce that there only exists a single valid non-interactive proof for a given chain. It therefore suffices if a single party produces the proof for the valid chain and forwards it to all FlyClient nodes.

Subchain Proofs. Another benefit of the non-interactive proofs is that they allow clients to re-sync to a chain that has grown since the last time they were given a proof for it, by only needing to download a shorter proof for the section of the chain they haven't seen. Once a FlyClient has received a proof for a chain of n blocks (or D cumulative difficulty), they are convinced that at the point in time when they received the proof for that chain it was the honest chain. Suppose that at a later point in time the chain has grown to n' blocks (or D' difficulty), the FlyClient needs only to verify that this new section is honest and thus only a proof logarithmic in the size of the new section. We note that the prover must also provide a single MMR proof that block n is in the MMR of block n', meaning the previous chain is a prefix of the new chain.

Theorem 2 (Subchain proofs)**.** *A* FlyClient *that was given a valid proof for a chain of length n at a time when the honest chain had length n, and when the honest chain has length n' is given a subproof for the subchain from n to n' including a Merkle proof that block n is in the MMR of block n', would not accept another chain if they were instead given the full proof for a chain of length n'.*

Proof. We consider two strategies the adversary may choose: (1) It forks from the honest chain after block n, this is as if the genesis block were set to block n and the subproof from block n to n' is a whole proof for a chain of $n' - n$ blocks. (2) The adversary forks from the honest chain before n, by the security of the proof for the first n blocks, the FlyClient would not accept the adversary's chain up to n so their subproof from n to n' would fail because the FlyClient's block n is not in the MMR of the adversary's new chain. The FlyClient that receives the whole proof would also not accept the adversary's proof based on the security of a proof for n' blocks. □

We note that a subchain proof does not have to be created specifically for the subchain, a FlyClient can take a proof for a chain of n' blocks and only check the blocks after n. This allows for FlyClient to use only the part of a transferable n' chain proof which it has not yet verified. This is a convenient option for FlyClient that may be running on cell phones or other data-limited devices and do not want to use data re-checking sections chains they have already verified. Subchain proofs also introduce the option of select checkpoint proofs, meaning that proofs can be created for select points in the chain and a FlyClient can request the precomputed proof they need, minimizing the computation overhead for prover full nodes and proofs will be more easily reused.

7 Evaluation

Experimental Setup. In order to measure FlyClient, we implemented the protocol and evaluated it computationally in two different scenarios. Our comparisons are focused on the proof size but both creating and verifying proofs is fast. Even in our unoptimized implementation, it takes less than a second over all tested parameters.

First, we compare FlyClient with NIPoPoW in the unrealistic scenario that all blocks have the same difficulty. NIPoPoW cannot handle variable difficulty chains. We show that both NIPoPoWs and FlyClient's proofs are logarithmic in the chain length and that FlyClient outperforms NIPoPoW over all parameters. Additionally, we show the performance of FlyClient on the actual Ethereum blockchain which has widely varying difficulty. FlyClient significantly outperforms standard SPV clients especially for longer chains. All evaluations assume a block header of size 508 bytes and a hash output of 32 bytes. Additionally, the MMR nodes contain 8 bytes to store the difficulty.

Implementation and Optimizations. We implemented FlyClient as a proof-of-concept in Python. Our implementation only supports the production and verification of FlyClient proofs and does not verify state transitions. We assume a hard fork, *i.e.*, that each block header contains the MMR root of all previous blocks. We perform several optimizations to minimize the proof size. First, we optimize for the smallest proof size by trying different values of δ. The security holds for arbitrary values of δ so a prover can choose a δ which minimizes the proof size. Note that our analytical optimizations from Section 5 does not directly apply as the difficulty is variable. However, it can still provide a good starting point for a numerical optimization of proof size.

We also reduce the proof size by not duplicating overlapping MMR proof elements. Note that overlaps are fairly common as our sampling distribution samples late blocks with significantly higher probability. The verifier can easily detect which nodes in a proof are shared and therefore does not require the duplicated information. The efficiency of this optimization is displayed in Figure 5. We can see that it reduces the proof size by around 30%. Additionally the plot shows the number of manually checked blocks vs. the number of randomly sampled blocks. Note that even at a chain length of $7,000,000$ the protocol only inspects around 600 blocks. We also see that L the number of manually inspected blocks hardly grows with increased chain length.

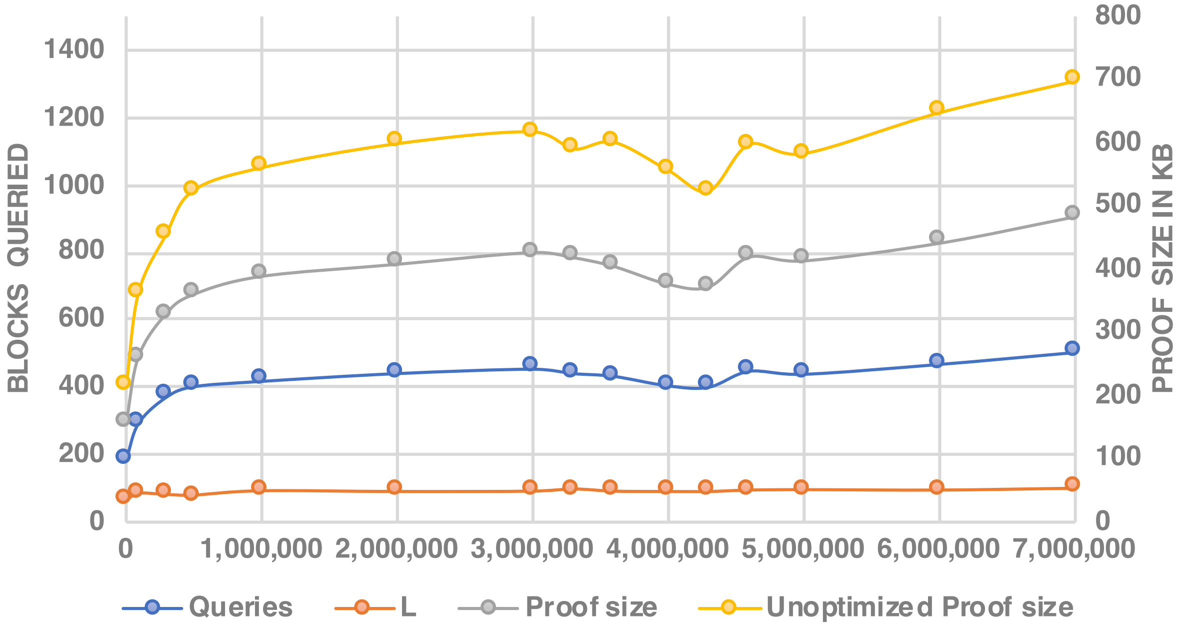

Figure 5: The plot shows the number of manually checked blocks L and the number of queried blocks for the Ethereum blockchain and $c = 0.5, \lambda = 50$. Additionally on the secondary axes the plot shows the proof size both without the MMR proof optimization and without.

7.1 Comparison with NIPoPoW

NIPoPoW like FlyClient promises short proofs of proof of work for light clients. We compare FlyClient with NIPoPoW by analytically computing NIPoPoWs proof size. We match the security level of NIPoPoW and FlyClient such that for security parameter λ an attacker who controls a c fraction of the main chain's mining power succeeds with probability $2^{-\lambda}$. Concretely, in NIPoPoW we set both the number of blocks checked at the end of the chain (k) and the length of each super-chain m to $\log_{\frac{1}{c}}(2)\lambda$. The total NIPoPoW proof size is

$$\log_{\frac{1}{c}}(2)\lambda \cdot ((\log_2(n)+1)\cdot B + \log_2(n)\cdot \lceil \log_2(\log_2(n,2),2)\rceil \cdot |H|),$$

for $B = 508$ bytes being the size of each block and $|H| = 32$bytes being the size of a hash. We compare the two light client approaches in Figure 6. The evaluation uses a security parameter of $\lambda = 50$ and 3 different parameterizations of c. c is a bound on the fraction of the honest mining power that an adversary controls. $\frac{c}{1+c}$ is the fraction of the total mining power that the adversary controls. For $c = 0.9$ this is 47.3%. We see that both proofs are very efficient producing proofs under 6MB even for the largest parameters. FlyClient outperforms NIPoPoW over all parameters but especially for large values of c, yielding an almost 40% improvement in proof size. This validates the optimization approach for finding an optimal light client design. Note that for $n = 10,000,000$, an SPV client would have required a 4.9GB proof over 1000 times more than the corresponding FlyClient proof for $c = 0.9$.

7.2 Ethereum Implementation with Variable Difficulty

We implement FlyClient for the Ethereum blockchain and measure its performance at different chain lengths,*i.e.*, at different historic data points. Ethereum's PoW difficulty is not constant but varies widely and has historically been increasing. FlyClient is the first proof of proof-of-work design that achieves succinct proof sizes for variable difficulty chains. We demonstrate the efficiency of FlyClient in Figure 7. For $c = 0.5$, *i.e.*, the adversary with less than a third of the total mining power the proofs are less than 1 MB even for 7,000,000 Ethereum blocks. This compares to a 3.4 GB SPV proof size for the same chain.

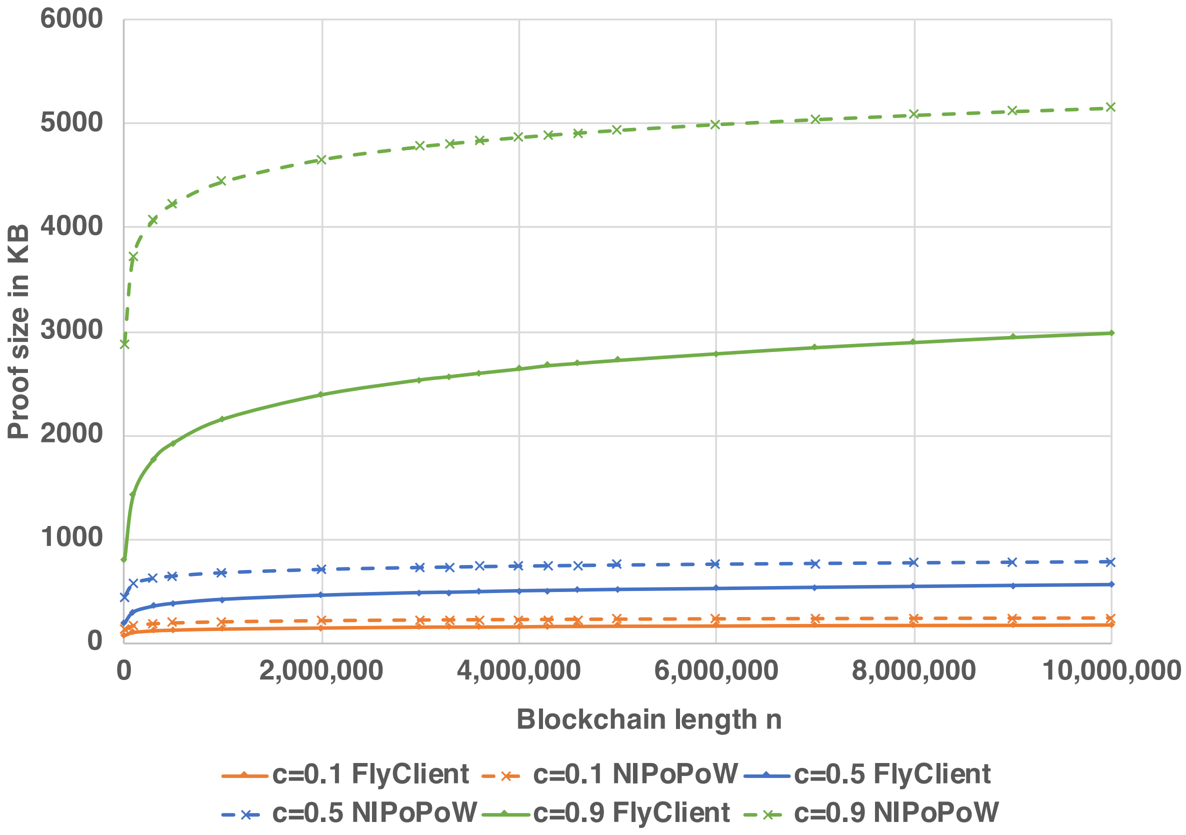

Figure 6: Comparison of FlyClient and NIPoPoW at varying difficulty levels and $\lambda = 50$.

We additionally plot the mining difficulty in the same figure. Interestingly, the proof size decreases from 3 to 4 million blocks as the difficulty rapidly grows. This is because with high difficulty growth the manually checked blocks contain a larger fraction of the overall difficulty. This reduces the number of blocks that need to be sampled from the rest of the chain. From 3 to 4 million blocks, a so-called difficulty bomb [47] resulted in a rapid increase of proof size. This "bomb" was removed at around 4.3 million blocks which led to a drastic decrease in difficulty and accordingly a slightly higher proof size.

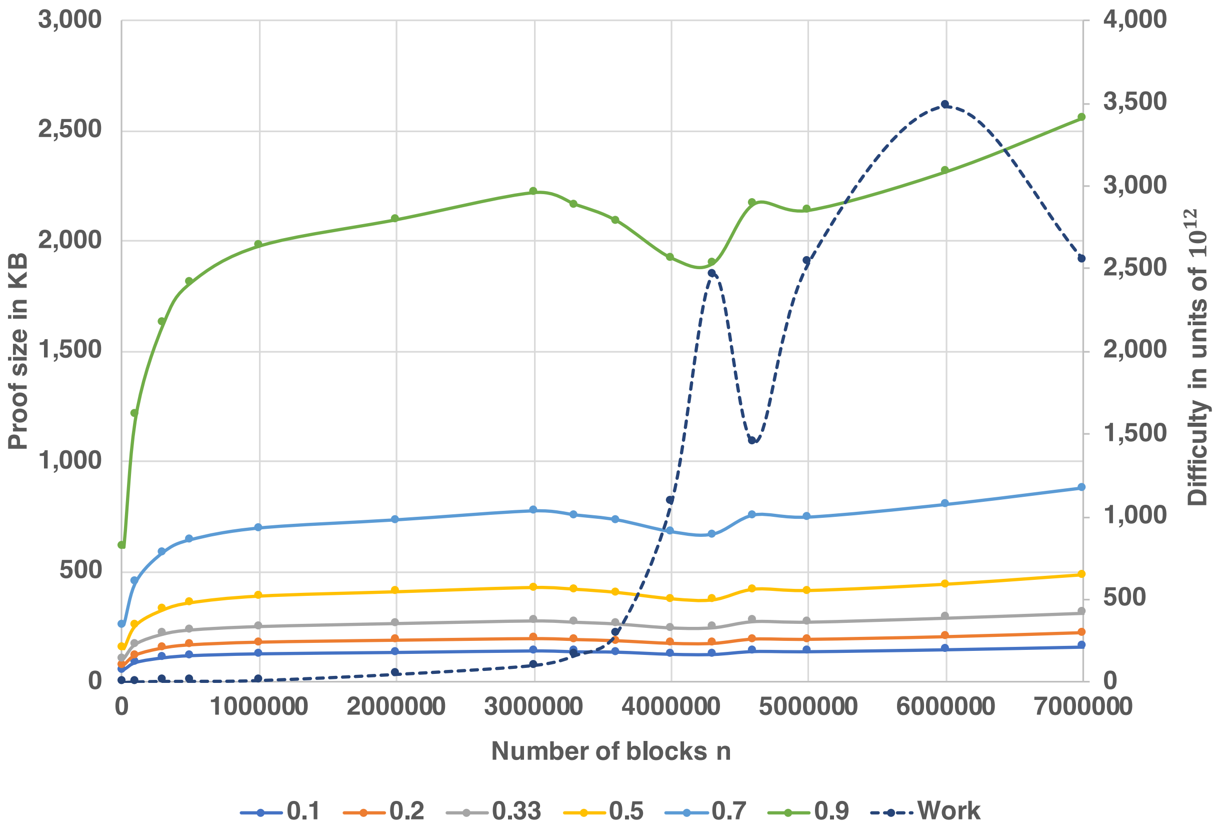

Figure 7: FlyClient for the Ethereum chain at varying chain lengths n and for different adversarial powers c. Additionally we display the difficulty on the secondary axis.

8 Discussion

8.1 Connection to Proof of Sequential Work

Cohen and Pietrzak presented a simple proof of sequential work(PoSW) construction based on Merkle Trees [22] with added edges. A PoSW convinces a verifier efficiently that a significant amount of sequential work was applied to a given input. In their construction the edges which are added to a full Merkle tree connect the left siblings of a leaf's path to the root with the leaf itself. The verifier simply queries random leafs and checks that they are a) part of the tree and b) have the correct incoming edges. The construction is almost[1] identical to an iterative MMR construction where every leaf is the root of the previous MMR. The FlyClient blockchain follows this design, storing the previous MMR root in every new block/leaf. It is easy to see that constructing a FlyClient chain of length n takes $\theta(n)$ sequential steps. [22]'s verification algorithm can be interpreted as our FlyClient protocol with a uniform querying distribution. A FlyClient blockchain is, therefore, a PoSW, albeit an inefficient one. In a PoSW a cheating prover will cheat on a constant fraction of leafs in order to save a significant amount of sequential work. FlyClient's security guarantee is stronger, ensuring that from no point on a constant fraction of leafs are corrupted.

[1] In [22]'s construction a node can have more than two incoming edges.

8.2 Deploying FlyClient

The only modification to the block structure of Bitcoin, Ethereum and similar blockchain protocols that is required to implement FlyClient is to include the MMR root in every block. The MMR root can be added to blocks in three different ways. The first way is a hard fork in which the MMR root is added to the header of all blocks (both old and new). In this case, the MMR root can even replace the current hash pointers to previous blocks. In some newer blockchain designs, such as the Mimblewimble [46]-based Grin and Beam MW [43], this is already the case. These blockchains can directly deploy the FlyClient protocol.

Alternatively, a soft fork can be used such that new blocks contain the MMR root while old blocks do not. In a soft fork, un-upgraded miners will not reject new blocks while upgraded miners may reject old blocks belonging to the un-upgraded miners. A soft fork gets "activated" when a majority of nodes have enforced the new protocol rules. Starting from the soft fork, new blocks would store the MMR root encoded in a backwards compatible way. For example, the MMR root can be stored in a special transaction. In the FlyClient protocol the miner would provide the block headers, as well as the special transaction and a proof that the transaction is part of the block. The proof size would grow by a factor that is proportional to $\log(|tx|)$, where $|tx|$ is the number of transactions. Since old nodes do not contain the MMR roots, a light client would still have to run the traditional, linear SPV verification. Alternatively the light client could hard code the block at which the soft fork was activated and use the FlyClient protocol to verify the chain since that block. A third deployment path is called a Velvet fork and was proposed by [36, 51]. In Velvet forks blocks by outdated miners are not being rejected. Velvet forks are therefore backwards compatible updates to blockchain protocols and rely on clients reinterpreting the blockchain data. For FlyClient the velvet fork would lead to a constant fraction α of blocks containing an MMR root. Blocks created by outdated nodes would not contain the root. The FlyClient protocol would simply treat multiple blocks as one. Concretely, those blocks that do not contain an MMR root are viewed as part of the next upgraded block. The miner will always download and check these joined blocks together. If in expectation $\frac{1}{\alpha}$ blocks are joined, the FlyClient proof would be at most $\frac{1}{\alpha}$ large than for an equivalent fully upgraded chain. Velvet forks, therefore, lead to less efficient proofs but provide an uncontentious deployment mechanism for FlyClient.

8.3 FlyClient for Proof-of-X Protocols

For simplicity, in this paper we describe FlyClient in the context of Bitcoin and Ethereum, where the blockchain grows based on a PoW mining process. Our protocol, however, is applicable to any *proof-of-X protocol* [18], where a more energy-efficient alternative to PoW is used to build a chain based on the longest chain rule, similar to Bitcoin and Ethereum. Examples of such alternatives are proof-of-stake [37], proof-of-space [25], or proof-of-elapsed-time [2]. Such a protocol must allow any node to verify the validity of each block individually ensuring that the block creator has spent (or burnt) a certain amount of a *resource* uniquely for this block.

Proof-of-space (PoS) protocols require a source of randomness that can reveal random strings in regular intervals to pick leaders (i.e. block proposers) randomly with respect to the stake distribution. PoS protocols typically extract this randomness from various sources such as previous blocks [23, 29] or multi-party coin tossing [37]. Some of these protocols grow their chains based on the longest chain rule that can result in forks. Some PoS protocols such as [29, 40], however, use a hybrid design to avoid forks. While FlyClient can be used in the first type of PoS protocols with minimal changes to allow lightweight transaction verification, it cannot be used as-is in the second type of PoS protocols for the reasons discussed in Section 8.4.

8.4 Light Clients for Hybrid Blockchains

Most hybrid blockchain protocols such as [45, 15, 40, 31, 50] that rely on classical Byzantine fault-tolerant (BFT) consensus protocols such as [20], including hybrid proof-of-stake protocols such as [23, 29, 37], create a special type of block, sometimes known as *identity blocks*, that store the identities of block validators, usually referred to as a *committee*. Every identity block contains the list of members of a new committee, signed by the previous committee, recording the transfer of custody from the previous committee to the new one, starting from a trusted "genesis committee". These committees are usually re-elected at a slower rate than the rate transaction blocks are added to the blockchain. Therefore, the number of identity blocks is usually much smaller than transaction blocks, possibly only a sublinear (in the length of the transaction blocks) number of identity blocks.

To verify that a block belongs to the valid chain in a hybrid protocol, a client can download and verify every identity block, and then verify the signature on the desired transaction block against the public keys of the committee members who witnessed the addition of the block to the blockchain. Without verifying every identity block, a malicious prover can deceive the client by providing a fake signature along with a fake set of public keys that match the signature. Some BFT-based protocols such as Algorand [29] that are resilient to a fully-adaptive adversary, choose a new committee for every transaction block resulting in a linear number of identity blocks required to verify transactions. To reduce this overhead by a factor of, say k, the acting committee can witness (i.e. sign) the election of k committees selected after it. This allows a light client to download only one identity block per every k identity blocks.

References

[1] Blockchain takes way too long to sync · issue #2394 · ethereum/mist. https://github.com/ethereum/mist/issues/2394, 2017. (Accessed on 11/29/2018).

[2] Intel sawtooth lake documentation, available at: https://intelledger.github.io, March 2017.

[3] Bitcoin wiki. https://en.bitcoin.it/wiki/Help:FAQ, 2018. (Accessed on 11/29/2018).

[4] Blockchain charts: Bitcoin's blockchain size, July 2018. Available at https://blockchain.info/charts/blocks-size.

[5] Ethereum blocks. https://etherscan.io/blocks, 12 2018. (Accessed on 12/21/2018).

[6] ethereum/btcrelay: Ethereum contract for bitcoin spv. https://github.com/ethereum/btcrelay, 2018. (Accessed on 12/14/2018).

[7] Getting deep into geth: Why syncing ethereum node is slow. https://hackernoon.com/getting-deep-into-geth-why-syncing-ethereum-node-is-slow-1edb04f9dc5, July 2018.

[8] Open timestamps. https://opentimestamps.org/, 2018.

[9] Over 175 million europeans ready to pay with wearable devices — global hub. https://newsroom.mastercard.com/press-releases/over-175-million-europeans-ready-to-pay-with-wearable-devices/, 2018. (Accessed on 11/27/2018).

[10] Secure payments and internet-of-things — visa. https://usa.visa.com/visa-everywhere/innovation/visa-brings-secure-payments-to-internet-of-things.html, 2018. (Accessed on 11/27/2018).

[11] Stampery. https://stampery.com/, 2018.

[12] Thin client security - bitcoin wiki. https://en.bitcoin.it/wiki/Thin_Client_Security, 2018. (Accessed on 11/29/2018).

[13] Top 6 vendors in the wearable payment market from 2016 to 2020: Technavio — business wire. https://www.businesswire.com/news/home/20161118005252/en/Top-6-Vendors-Wearable-Payment-Market-2016, 2018. (Accessed on 11/27/2018).

[14] Difficulty - blockchain. https://www.blockchain.com/en/charts/difficulty, 2019. (Accessed on 01/17/2019).

[15] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *Proceedings of the 21st International Conference on Principles of Distributed Systems*, OPODIS '17, 2017.

[16] E. Kasper B. Laurie, A. Langley. Rfc 6962 - certificate transparency. https://tools.ietf.org/html/rfc6962#section-2.1, June 2013.

[17] Adam Back and Gregory Maxwell. Transferring ledger assets between blockchains via pegged sidechains, Nov 2016. US Patent App. 15/150,032.

[18] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *CoRR*, abs/1711.03936, 2017.

[19] Vitalik Buterin. Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[20] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, 1999.

[21] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[22] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 451–467. Springer, 2018.

[23] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. https://eprint.iacr.org/2016/919.

[24] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology — CRYPTO' 92: 12th Annual International Cryptology Conference Santa Barbara, California, USA August 16–20, 1992 Proceedings*, pages 139–147. Springer Berlin Heidelberg, 1993.

[25] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. Cryptology ePrint Archive, Report 2013/796, 2013. http://eprint.iacr.org/.

[26] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[27] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, 2015.

[28] Arthur Gervais, Hubert Ritzdorf, Ghassan O. Karame, and Srdjan Capkun. Tampering with the delivery of blocks and transactions in bitcoin. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 692–705. ACM, 2015.

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

[30] S Goldwasser and M Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 59–68, New York, NY, USA, 1986. ACM.

[31] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

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

[33] Maurice Herlihy. Atomic cross-chain swaps. *arXiv preprint arXiv:1801.09515*, 2018.

[34] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 565–. IEEE Computer Society, 2000.

[35] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. *Proofs of Proofs of Work with Sublinear Complexity*, pages 61–78. Springer Berlin Heidelberg, 2016.

[36] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. 2017.

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

[38] Aggelos Kiayias and Dionysis Zindros. Proof-of-work sidechains. Cryptology ePrint Archive, Report 2018/1048, 2018. https://eprint.iacr.org/2018/1048.

[39] Lucianna Kiffer, Rajmohan Rajaraman, and abhi shelat. A better method to analyze blockchain consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 729–744. ACM, 2018.

[40] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 19–34, 2018.

[41] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378. Springer-Verlag, 1988.

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

[43] Rachel Rose O'Leary. Grin and beam: A tale of two coins being built on mimblewimble. https://www.coindesk.com/grin-and-beam-a-tale-of-two-coins-being-built-on-mimblewimble, December 2018. (Accessed on 02/05/2019).

[44] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673. Springer International Publishing, 2017.

[45] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/2016/917.

[46] Andrew Poelstra. Mimblewimble, 2016. https://scalingbitcoin.org/papers/mimblewimble.pdf.

[47] Rakesh Sharma. What is ethereum's "difficulty bomb"? https://www.investopedia.com/news/what-ethereums-difficulty-bomb/, August 2018. (Accessed on 02/05/2019).

[48] socrates1024. The high-value-hash highway. https://bitcointalk.org/index.php?topic=98986.0, 2012.

[49] Peter Todd. Merkle mountain range. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, 2016.

[50] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *2018 ACM Conference on Computer and Communications Security (CCS)*, 2018.

[51] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar Weippl, and William J. Knottenbelt. (short paper) a wild velvet fork appears! inclusive blockchain protocol changes in practice. Cryptology ePrint Archive, Report 2018/087, 2018. https://eprint.iacr.org/2018/087.

Parameter	Definition
n	Chain length
c	Fraction of malicious hash power to honest power
a	Fork point
f	Sampling probability distribution
k	Fraction of blocks/weights queried from the tip (*i.e.*, end) of the chain
L	Number of blocks queried at the tip of the chain
q	Total number of queries

Table 2: List of parameters used throughout the paper

Appendices

A Basic Merkle Tree

A Merkle tree is a balanced binary tree where the leafs hold some value, and each non-leaf node stores a hash of both of its children. Such a structure allows proving the inclusion of any value in the tree with only a logarithmic number of hashes, known as a Merkle proof. In Bitcoin, Merkle trees are used to aggregate the hashes of transactions in a particular block so that the root becomes a binding commitment to all transactions in that block. The root is then stored in the header of the block. An SPV proof of a transaction is the Merkle proof that the hash of the transaction is a leaf in the Merkle tree. Though it is a commonly-used data structure, we redefine a Merkle tree and the security of a Merkle proof in the following so we can extend the definition to MMRs later.

Definition 2 (Merkle Tree)**.** *Given a list of values, a Merkle tree is a balanced binary tree, where each leaf node stores some value, and each non-leaf node holds the value* $H(\mathsf{LeftChild}||\mathsf{RightChild})$*, where* H *is a collision-resistant hash function. Balanced binary tree here means a tree with* n *leaves that has depth less than or equal to* $\lceil \log_2 n \rceil$*.*

Definition 3. *Given a Merkle tree, MT, with root* r*, a Merkle proof that* x *is the* k*th node in MT,* $\Pi_{k\in MT}$*, are the siblings of each node on the path from* x *to* r*. Since MT has depth at most* $\lceil \log_2(n) \rceil$*, the proof length is at most* $\log_2(n)+1$ *as each node in the path can be calculated from it's two children so we only need the siblings and the 2 leaf nodes.*

Below, we define a prover-verifier model, where the verifier knows the root of a Merkle tree and the prover wants to convince the verifier that a particular node exists in the tree.

Prover-Verifier Model.

1. Verifier has access to $r =$ root of some Merkle tree, MT.
2. Prover has access to MT and generates a Merkle-Proof path of some $x \in MT = \Pi_{k\in MT}$ using Protocol 3 and sends it to the verifier.
3. Verifier uses the proof and x to build up the path to r' using Protocol 4, and checks that $r' = r$.
4. If the checks pass, the Verifier accepts the proof, otherwise it rejects the proof.

Theorem 3. *Given a Merkle tree,* MT*, a polynomial-time adversary cannot produce a valid proof* $\Pi_{k\in MT}$*, for a* k *not in* MT*. [Soundness of Merkle-proofs]*

Proof. Assume the adversary can produce a valid proof $\Pi_{k\in MT}$. Let r be the root of MT, any proof must start with r, otherwise the verifier will reject it. Since $k \notin MT$, the path the adversary gives must have some initial depth i at which it differs from any true path in MT.
Let p'_i be the node in the path at level i and s'_i be its sibling, and let p_i and s_i be the true nodes in a path in MT where $x = p_i||s_i$ or $x = s_i||p_i$ s.t. $H(x) = p_{i-1}$. In order for the verifier to accept $\Pi_{k\in MT}$, x' must equal $p'_i||s'_i$ or $s'_i||p'_i$ s.t. $p_{i-1} = H(x')$. Since the sets $\{p_i, s_i\}$ and $\{p'_i, s'_i\}$ differ by at least one value as stated above, $x \neq x'$ therefore the adversary found a collision of H $(\perp)$. □

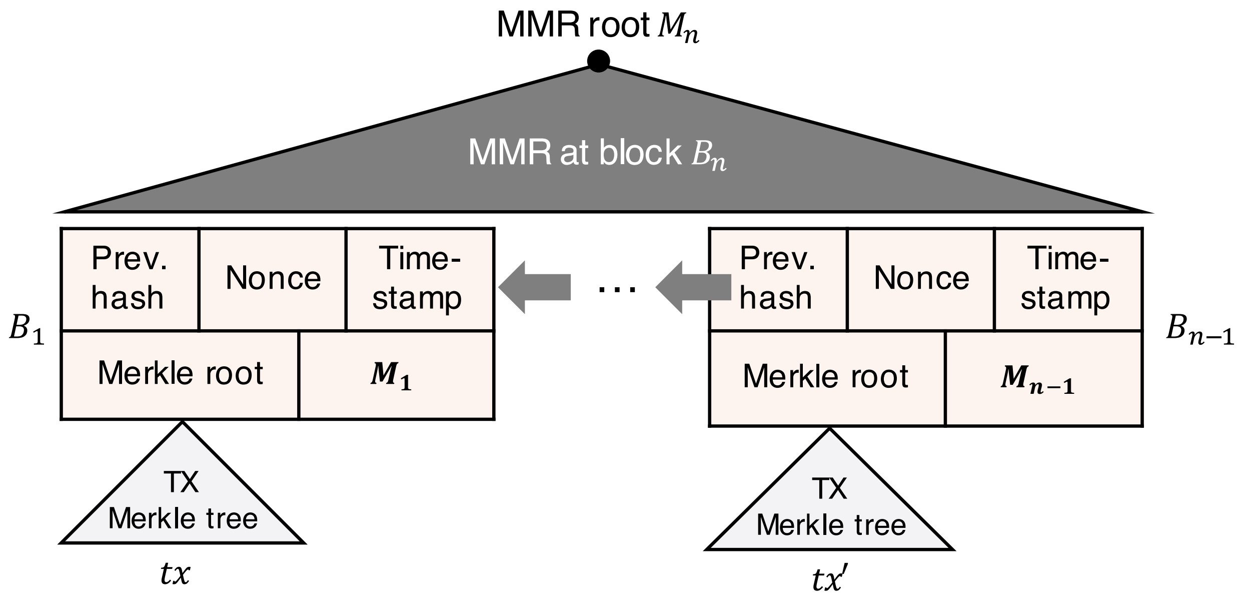

Figure 8: MMR chain commitments

Theorem 4. *Given a Merkle tree, MT, and a node $k \in MT$, a polynomial-time adversary cannot generate a proof $\Pi_{k \in MT}$ that is not a true path in MT. [Completeness of Merkle proofs].*

Proof. Same as the proof of soundness, if there is some point in the path that differs from a true path in MT, in order for it to be valid, the adversary must have found a hash collision. □

Algorithm 3 Merkle_Proof (Merkle root r, index k) → MMR Proof Π_k for leaf k

```
if $r$.leaves $= 0$ then
    return []
end if
if $k \leq r$.left.leaves then
    $\Pi \leftarrow$ Merkle_Proof( $r$.left, $k$)
    return $\Pi || r$.right.value
else
    $\Pi \leftarrow$ Merkle_Proof( $r$.right, $k - r$.left.leaves)
    return $\Pi || r$.left.value
end if
```

B Merkle Mountain Range

Definition 4. *A Merkle Mountain Range, M, is defined as a tree with n leaves, root r, and the following properties:*

1. *M is a binary hash tree*
2. *M has depth $\lceil \log_2 n \rceil$*
3. *If $n > 1$, let $n = 2^i + j$ such that $i = \lfloor \log_2(n-1) \rfloor$:*
   - r.left is an MMR with 2^i leaves
   - r.right is an MMR with j leaves

Algorithm 4 Verify_Merkle_Proof (Merkle tree root r, number of leaves in the Merkle tree n, index k, element x, Merkle proof $\Pi_{k\in n}$)

Note: This algorithm can be written recursively since every subtree of an MMR is also an MMR.

```
$y \leftarrow \mathsf{H}(x)$, $k' \leftarrow k-1$, $n' \leftarrow n-1$
if $|\Pi_{k\in n}| \neq \lceil \log_2(n') \rceil$ then
    return $reject$
end if
for $z \in \Pi_{k\in n}$ do
    if $k' \bmod 2 = 0 \wedge k'+1 \leq n'$ then
        $y \leftarrow \mathsf{H}(y||z)$
    else
        $y \leftarrow \mathsf{H}(z||y)$
    end if
    $k' \leftarrow \lfloor \frac{k'}{2} \rfloor$, $n' \leftarrow \lfloor \frac{n'}{2} \rfloor$
end for
if $y = r$ then
    return accept
else
    return reject
end if
```

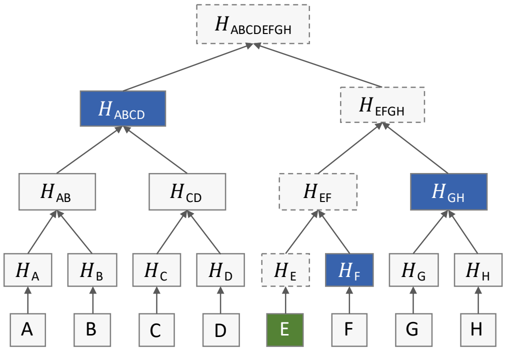

Figure 9: A Merkle tree for elements “A” through “H” and the corresponding Merkle proof for element “E”, that is $\Pi_E = (H_{\mathrm{F}}, H_{\mathrm{GH}}, H_{\mathrm{ABCD}})$ marked as dark blue nodes. H represents a cryptographic hash function, $H_x = H(x)$ for any element x, and $H_{\mathrm{XY}} = H\left(H_{\mathrm{X}}||H_{\mathrm{Y}}\right)$, for any sequence of letters X and Y.

Note: M is a balanced binary hash tree, *i.e.*, M is a Merkle tree. Therefore, for all nodes $k \in M$, $\exists\, \Pi_{k\in M}$.

We define AppendLeaf in Protocol 5, the $O(\log n)$ algorithm used to append new nodes to an existing MMR with n leaves. Below, we state a theorem that AppendLeaf returns an MMR as defined, and refer the proof to Appendix B.

Theorem 5. *Given an MMR, M, with root r and n leaves, AppendLeaf(r,x) will return an MMR, M', with n+1 leaves (the n leaves of M plus x added as the right-most leaf).*

We now define a set of MMRs $M = \{M_1, M_2, ..., M_n\}$ created from some list $[x_1, x_2, ..., x_n]$, where M_1 is a single node with value x_1 and r_i is the root node of an i leaf MMR, $M_i = \text{AppendLeaf}(r_{i-1}, x_i)$. A key feature of the way MMRs are constructed is that, assuming all x_i's are unique, each M_i has an unique root (otherwise there would be a hash collision), and given the Merkle proof $\Pi_{x_k \in M_n}$ that some x_k is in M_n for $k \leq n$, a verifier can regenerate r_k and therefore verify that M_k is an *ancestor* of M_n (*i.e.*, M_n was created from $n-k$ appends to M_k). We state this in the following theorem and refer the proof to Appendix B, and Protocol 6 describes the algorithm used to regenerate the root M_k.

Algorithm 5 AppendLeaf(MMR root r, new leaf node x): Returns new MMR root r'

```
if $r.\mathsf{leaves} = 2^i$ for $i \geq 0 \in \mathbb{Z}$ then
    Node $r'$
    $r'.\mathsf{left} \leftarrow r$
    $r'.\mathsf{right} \leftarrow x$
    $r'.\mathsf{value} \leftarrow \mathsf{H}(r||x)$
    $r'.\mathsf{leaves} \leftarrow r.\mathsf{leaves} + 1$
    return $r'$
else
    $r.\mathsf{right} \leftarrow \mathsf{AppendLeaf}(r.\mathsf{right}, x)$
    $r.\mathsf{value} \leftarrow \mathsf{H}(r.\mathsf{left}||r.\mathsf{right})$
    $r.\mathsf{leaves} \leftarrow r.\mathsf{leaves} + 1$
    return $r$
end if
```

Algorithm 6 Get_Root(number of leaves in the MMR n, proof for block k $\Pi_{k \in n}$): Given $\Pi_{k \in n}$, the algorithm returns the root for the MMR of the tree with $k-1$ blocks, *i.e.*, the root stored in the header of block k

```
$k' \leftarrow k-1$, $n' \leftarrow n-1$, $r = \perp$
for $y \in \Pi_{k \in n}$ do
    if $k' \bmod 2 = 1 \vee k'+1 > n'$ then
        if $r = \perp$ then
            $r = \Pi[i]$
        else
            $r = \mathsf{H}(y||r)$
        end if
    end if
    $k' \leftarrow \lfloor \frac{k'}{2} \rfloor$, $n' \leftarrow \lfloor \frac{n'}{2} \rfloor$
end for
if $y = r$ then
    return 1
else
    return 0
end if
```

Theorem 6. *For $k \leq n$, given $\Pi_{x_k \in M_n}$, i.e., the Merkle proof that leaf x_k is in M_n, a verifier can regenerate r_k, the root of M_k.*

Corollary 1. *If $x_1, ..., x_n$ are the hashes of blocks 1 through n of chain C_n, r_n commits the first n blocks to x_n, and $\Pi_{k \in M_n}$ for any k commits $x_1, ..., x_k$ as the blocks of the chain C_k, where chain C_k is a prefix of chain C_n.*

Corollary 2. *If an adversary changes any block i in the chain in any way, then it's hash x_i will also change, so any MMR M_k for $k \geq i$ with root r'_k that contains the new block x'_i will have that $r'_k \neq r_k$.*

Using the above constructions, we can now define terminology which we will use for the remainder of the paper.

Definition 5. *A **valid** block B_x for a chain ending in block B_n with MMR root M_{n-1}, is a header with POW and for which a $\Pi_{x \in M_{n-1}}$ exists.*

Definition 6. *An **honest** chain $B_0, B_1, ..., B_n$ of length n, is an ordered list such that each B_i is valid.*

Theorem 7. *Given an MMR, M, with root r and n leaves, AppendLeaf(r,x) will return an MMR, M', with n+1 leaves (the n leaves of M plus x added as the right-most leaf).*

Proof. Induction on n
Base case: $(n = 1)$ M is a single node r with depth 0. r.children $= 0$, so AppendLeaf returns a new node with left $= r$ and right $= x$, and value $= H(x||r)$. This is a balanced binary hash tree with 2 leaves and depth $1 = \log_2 2$.
Induction step: Assume theorem holds for all M with $< n$ leaves. Let M be an MMR with n leaves and root r, AppendLeaf(r,x) will return the following:

(i) if $n = 2^i$ for some $i \in \mathbb{N}$, AppendLeaf returns a new node, r', with left $= r$, right $= x$ and value $= H(r||x)$. M' is the new tree with the 3 properties of an MMR.

1. Since M is a hash tree, so is M'.
2. Since the depth of $M = \log_2 n$, the depth of $M' = \log_2 n + 1 = \lceil \log_2(n+1) \rceil$
3. $n' = 2^i + 1$
   r'.left $= M$, a MMR with $n = 2^i$ leaves
   r'.right $= x$, a MMR with 1 leaf

The leaves of M' are the leaves of M plus x added as the new right-most leaf.

(ii) Otherwise, $\exists i, j \in \mathbb{N}$ s.t. $n = \max_i 2^i + j$, AppendLeaf returns r with r.left the same, and r.right $=$ AppendLeaf(r.right, x), and value $= H(r.\text{left}||r.\text{right})$. M' is the new tree with the following MMR conditions satisfied.
(1,3) r'.left is an MMR by definition with 2^i leaves, r'.right is an MMR by the induction hypothesis with $j + 1$ leaves, thus M' is a hash tree.
(2) M has depth $\log_2 2^i = i \geq j$, thus M' has depth $i + 1 = \lceil \log_2(n+1) \rceil$.
The leaves of M' are the leaves of r'.left $=r$.left, then the leaves of r'.right which by the induction hypothesis will be the original leaves of r.right plus x on the right-most side.

□

Theorem 8. *For $k \leq n$, given $\Pi_{x_k \in M_n}$, i.e., the Merkle proof that leaf x_k is in M_n, a verifier can regenerate r_k, the root of M_k.*

Proof. Induction on n.
Base case: $(n = 1)$ $M_1 =$Node(x_1), $\Pi_{x_1 \in M_1} = [r_1]$.
Induction step: Assume the theorem holds for all M_m, $m < n$ and $k \leq m$. Given M_n, any k and $\Pi_{k \in M_n} = [r_n, r_n.\text{left}, r_n.\text{right}, ...]$, if $k = n$ then $r_k = r_n$. Otherwise, let i be the maximum integer s.t. $n = 2^i + j$ where $j > 0$. We have 3 possibilities:

(i) $k = 2^i$, $r_k = r_n$.left

(ii) $k < 2^i$, thus x_k is in the left subtree of M_n. Let $n' = 2^i$ and $r_{n'} = r_n$.left, we get that $\Pi_{x_k \in M_{n'}} = \Pi_{x_k \in M_n} - [r_n, r_n.\text{right}]$. Since $n' < n$, by the induction hypothesis we can get r_k from $\Pi_{x_k \in M_{n'}}$.

(iii) $k > 2^i$, thus x_k is in the right subtree of M_n. Since $k < n$ and i is the maximum integer s.t. $n = 2^i + j$ for some $j > 0$, i is also the maximum integer s.t. $k = 2^i + j'$ for some $j' > 0$. Thus r_k.left $= r_n$.left. Note r_n.right is the MMR M_j where k is the $k' = k - 2^i = j'$th leaf. Thus, r_k.right $= M_{k'}$ and $\Pi_{x_{k'} \in M_j} = \Pi_{x_k \in M_n} - [r_n, r_n.\text{left}]$. By the induction hypothesis we can extract $r_{k'}$ from $\Pi_{x_{k'} \in M_j}$. The verifier hashes the left and right roots to get the value of r_k.

□