JACOBS
UNIVERSITY

Superlight Clients for Proof of Stake Ethereum

by

Shresth Agrawal

Bachelor Thesis in Computer Science

Submission: May 16, 2022
Research Supervisor: Dionysis Zindros
Internal Supervisor: Sören Petrat

Jacobs University Bremen | Department of Computer Science and Electrical Engineering

Statutory Declaration

Family Name, Given/First Name	Agrawal, Shresth
Matriculation number	30002701
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

16\05\2022 [illegible]

....

Date, Signature

Abstract

Blockchains are decentralized, trust-less, and cryptographically secure ledgers. Ethereum is one of the most adopted Blockchains, allowing smart-contract execution. The most common way to interact with Ethereum is through wallet applications. These wallet applications run on consumer hardware like PC, Browsers, or smartphones. Such devices are constrained in storage and network bandwidth; hence, wallets do not actively verify the blockchain. Instead, they rely on an external full-node service provider to get the latest state of the blockchain. This is counterintuitive to the principle of decentralization as these full-node service providers can behave maliciously or get hacked. Wallets can run Light Clients (SPV), which only needs to verify the block headers. However, they solve the problem partially as they require hundreds of megabytes of storage and network bandwidth to synchronize[13]. For Proof of Work blockchains, there exist many constructions and implementation[21, 20] which allow building superlight clients which can sync in poly-log complexity. For Proof of Stake blockchains, there exists an unpublished theoretical construction[37] which allows building superlight clients using interactive bisection games. This construction requires consensus level changes which would need a chain hardfork. This paper proposes a similar construction to build a superlight client tailored for Proof of Stake Ethereum. We achieve this without any hard-fork leveraging the existing Sync Protocol. We also present the first implementation of the superlight client for a Proof of Stake blockchain and benchmark our implementation w.r.t the state of the art light client.

Contents

1 Introduction

Bitcoin is the first peer-to-peer version of electronic cash that allows for online payments to be sent from one party to another directly without going through any financial institution [27]. Bitcoin uses Blockchain technology to achieve this. In a blockchain every block consists of a batch of transactions. Every block links to a previous block forming a chain of blocks, **blockchain**. Every peer has a copy of the complete blockchain. To include a block into the blockchain, peers compete to find a nonce such that the block's hash is below the certain difficulty threshold. This is a difficult problem as hashing is a one-way function. This process is called mining, and peers who participate in mining are called miners. Bitcoin assumes that the majority of the computation power relies on honest miners. This assumption allows incrementing the chain safely.

Several other blockchains have extended the idea of Bitcoin. Ethereum is the first blockchain to introduce Smart Contracts [7]. Smart Contracts are arbitrary pieces of code that live on the blockchain, and they have a state and can hold funds. They can be programmed to mutate the state, transfer funds, or call other contracts if a specific action is made on this contract. Smart Contracts allow for building applications without any centralized trusted server as the blockchain guarantees the trust.

The method that is used to decide which block enters the blockchain is called the **consensus protocol**. The method of brute-forcing the nonce to find a good hash such that a block can be included in the blockchain is called **Proof of Work (PoW)**. Both Bitcoin and Ethereum currently use Proof of Work. Mining is highly energy-intensive. At the time of writing Bitcoin network alone uses 91 terawatt-hours of electricity annually, more than it is used by the country of Finland [17]. Proof of Work is not sustainable for the environment. Hence most modern blockchains use **Proof of Stake (PoS)**. In Proof of Stake, miners are selected randomly based on the coins they own in the system, **stake**, to propose a block to the blockchain [23]. Ethereum is currently in the process of upgrading its consensus to Proof of Stake. In this paper, we exclusively work on the upcoming Proof of Stake Ethereum.

Before a node can propose a block to the blockchain, it has to know the correct latest block and state of the blockchain. If the node starts for the first time, it might only have the initial block called **genesis**. The process of securely arriving from the current known block to the latest block is usually referred to **sync**. The traditional way to sync is to ask peers in the network for all the historical transactions and blocks and then sequentially go over every transaction and block to check the correctness to arrive at the final state. Let us denote the number of transactions in the blockchain by $|t|$ and the number of blocks as $|n|$. The time complexity to sync a **full-node** is $O(|t| + |n|)$. The current recommended requirement for running an Ethereum full-node is $> 500GB$ SSD storage and $25+$ MBit/s network bandwidth[30].

Only a signature is required from the private key to propose a transaction to the blockchain. Most popular wallets that store the private keys are browser extensions, smartphone apps, PC apps, and specialized hardware wallets. These wallets do not participate in the consensus or even verify the latest blockchain state as they are constrained in network and storage bandwidth. Instead, they blindly rely on external full-node providers to get the latest state of the chain. These providers make the wallets centralised and not secure for the end-user. This is a big issue for the blockchain industry, and multiple concerns have been raised in the past[25].

A partial solution to the problem is to build a **light client** also called as **Simple Payment Verification (SPV)**. Light clients do not participate in the consensus, but they can verify the inclusion of transactions or the correctness of the latest state of the blockchain. The data structure used to represent a block in the blockchain is called **blockheader**. A blockheader consists of the Merkle root of the transactions, nonce, previous block hash, Merkle root of the state if the blockchain has a state and other essential values depending on the consensus protocol. A light client only verifies the block headers of the chain. Merkle inclusion proofs can be used to verify a transaction inclusion or state correctness to the transaction root or the state root present in the blockheader. The time complexity to sync for a light client is $O(|n|)$. Currently, running a light client for Ethereum requires $400MB$ of storage[13]. The light client might be fine for PC wallets but not efficient enough for browser-based and mobile wallets.

To solve this problem we can use **superlight** clients. These clients sync in sublinear time complexity. There exist constructions in Proof of Work setting which selects special blockheaders called **superblocks**, which occur $log(|n|)$, to be verified [21, 20]. Building such systems is much more difficult in the Proof of Stake setting. To verify if a proof of stake blockheader is correct block proposer of the slot must be known which requires the knowledge of the stakes of all the validators in the system and the random value used for selection. There exist an unpublished construction ”Proof of Proof of Stake in sub-linear complexity” by Zindros[37] which solves this problem for Proof of Stake blockchains. It uses the interactive bisection games to achieve $polylog(|n|)$ verification. The main demerit of this construction is that it requires a hard-fork. Hard-fork is when a change in consensus requires all the nodes participating in the blockchain to upgrade their software. Hard-fork is a challenging process for a decentralized system and is avoided as much as possible.

The paper proposes a construction to build a superlight client for Proof of Stake Ethereum without any hard work. The proposed construction extends the original construction of ”Proof of Proof of Stake in sub-linear complexity”. Consider a wallet wishing to know its current balance or a smart contract state. The construction proposed in this paper allows them to sync to the latest blockheader of the chain quickly. This paper also presents the first implementation for a superlight client for proof of stake setting. This implementation can be extended to production-grade code and used in popular wallets to become trustless and secure.

1.1 Contribution

The main contribution of the paper is as follows:

1. Evaluation of the official light client specification based on the sync protocol. Clearly state the safety and liveness assumption made by the sync protocol.
2. Propose a construction for building superlight clients for Proof of Stake Ethereum, which is exponentially better than the official specification and does not require any hard-fork.
3. Presents the first implementation of superlight clients for Proof of Stake blockchains. This implementation can be easily extended to production-grade code, which wallets can use.
4. Benchmark the superlight client and light client in real network conditions to com-

pare storage, bandwidth and interactivity requirements.

1.2 Construction Overview

Currently, every block in the PoS Ethereum is signed by a set of 512 validators called **sync committee**. The sync committee is constant during a **sync committee period** (or just period for short) which is roughly one day. A block signed by $2/3$ of the sync committee can be considered finalised. To know the latest block of the chain, it is sufficient to know the correct latest sync committee and the latest block, which is signed by $2/3s$ of them. The sync committee is known one period upfront and is included in the blockheader. As the sync committee signs the blockheaders of the current period, it indirectly attests to the sync committee for the next period. This forms a chain of sync committees as every sync committee signs the following sync committee. To get to the latest sync committee, we can start from the genesis sync committee and linearly verify a single block from each period to know the following sync committee until the latest period.

Now consider a PoS Ethereum superlight client that connects to two full nodes, bob and charlie, out of which one is honest. The client wants to sync to the latest blockheader of the chain, and it asks both bob and charlie for the latest block header. If both of them answer the same, he can be assured it is the correct blockheader as one is honest. If they are not the same, the client would like to know which out of the two is correct, so it does further interrogation with bob and charlie.

To find which out of bob and charlie is honest, the superlight client would like to know if bob or charlie know the correct sync committee and have 2/3 signatures from that committee to prove their blockheaders. The client can ask for the sync committee and signatures and check if the signatures are valid. Now it only remains to check which sync committee is correct as per the sync committee chain. The client asks both bob and charlie to present a binary Merkle root of the sync committee chain. The tree leaves are concatenated sync committee addresses for each period arranged sequentially, starting with the genesis sync committee.

Both the roots are a sequential commitment to the same set of sync committee addresses; there must be at least one leaf where both parties disagree. To find the first point of disagreement, both bob and charlie go through a bisection game under the supervision of the client. The client asks both bob and charlie to open the children of the sync committee root, which are the roots of their subtrees. The client checks first the left and then the right root. One of them must be different. For the different root, the client again asks the nodes to open the children. This process is recursively continued until the leaf of the tree containing different sync committees is reached. This leaf is the first point of disagreement between bob and charlie. Before this point, both bob and charlie agree on the sync committee, and as one of them is honest, both of them should be correct. Hence the client has only to check which of the two is correct at the point of disagreement. The client can ask for the last common sync committee (the committee before the point of disagreement) from either bob or charlie and a valid signature from the last common committee to the committee of disagreement from both of them. Only one of them should be able to present the correct signatures; otherwise, at least 1/3 sync committee has attested to two different sync committees. This allows the client to know which of bob and charlie is honest. The complexity of the interaction is $log(|n|)$, which is the depth of the Merkle tree. Hence the superlight client can sync in sublinear complexity.

1.3 Structure of Paper

The paper starts by first describing a high-level construction overview. Then it introduces essential concepts in Cryptography and Blockchain that is highly relevant for the construction presented in the paper. Then the paper explains the construction of Proof of Proof of Stake(PoPoS) in sublinear complexity [37], an unpublished work by Zindros. In the construction, we first discuss a linear model to build intuition. Then we introduce bisection games to improve the linear model to polylog in the simple case of two provers. Then the paper analyzes the safety and liveness of the current official light client specification. After that, the paper introduces a PoPoS modification which can be used with PoS Ethereum without any hardfork. Then the paper also relaxes the initial assumption about two provers to multi prover model by explaining the tournament algorithm. The paper then explains the implementation of the superlight client using the above protocol. Finally, the paper compares the superlight client implementation with the existing light client implementation w.r.t to chain sizes, provers, storage requirements, interactions and network bandwidth.

1.4 Related Work

The problem of quickly and efficiently verifying the blockchain to know the latest state of the chain has been an active area of research. This problem has been rigorously explored in the Proof of Work setting [21, 20, 6] in both interactive and non-interactive methods. Also, work has been done to make such constructions efficient and feasible to implement[11, 12]. In this paper, we work on Proof of Stake blockchains. The main construction discussed in this paper is a modification of an unpublished work by Zindros[37]. The original construction proposed by Zindros works for any PoS blockchain with specific properties. The construction in this paper is specially tailored for PoS Ethereum. The PoPoS construction uses a handover period between epochs; this concept was first introduced in [15]. The construction proposed in this paper is interactive, but non-interactive approaches exist to building superlight clients. These constructions use progressive SNARKs to build a constant time verifier[4, 5, 35]. The main limitation is that these constructions require a consensus design which is SNARK friendly.

On the Ethereum front, currently, there exists the sync protocol described in the official PoS Ethereum consensus specification [14]. The sync protocol describes the implementation specification to build light client for PoS Ethereum. The specification does not formally discuss about the safety and liveness of the sync protocol. Many details about the specifications are at the time of writing work in progress. The most up to date implementation of a light client based on the sync protocol is the Lodestar light client [32]. The implementation presented in the paper will reuse the data structure, signature verification and standard function from the Lodestar implementation. Reusing the implementation allows for building a superlight client close to the official light client specifications. There exist some ongoing efforts by Nimbus Team to build Fluffy using Portal network[33]. Fluffy is an ultralight client that relies on the Portal Network, a DHT(Decentralised Hash Table) based on Kademila. So far, the project does not look into the consensus verification but instead fetches the consensus data over a DHT. Using DHT indeed makes the client decentralised, but there are no security guarantees on the correctness of the data available on the DHT.

The construction used in this paper uses interactive bisection games to find the honest prover. Bisection games were first introduced in the setting of delegating computation to untrusted servers [10]. Later it was used by Arbitrum to build Ethereum scaling solution using optimistic rollups [19]. More recently, Bisection Games was used in the ledger verification to build light clients for lazy blockchains[31].

2 Preliminary

Hashing. It is a method for converting arbitrary length data to fixed-size data using a one-way function. A hash function is assumed to have pre-image resistance and collision resistance. Hashing is often used in commit-reveal settings where the prover wants to commit to a specific data without revealing any information about the data and then later revealing the actual data. A prover can do this by presenting the hash of an arbitrary data as the commitment to the data. The prover can then provide the actual data, and the verifier can check if the hash of the presented data matches the previously committed hash. If the hash matches, the verifier can be confident that the prover committed to this data initially. Ethereum currently uses keccak256[1] from the SHA-3 family[36] as the default hash function. The paper and implementation will also use keccak256 as the hash function.

Merkle Tree. A natural extension to the commit-reveal scheme is when a prover would like to commit to a set of data and in future would like to prove the commitment of only some of the data without needing to reveal the complete data. This can be achieved using Merkle Trees [26, 24]. A Merkle tree is a balanced tree where every node has n children where n is the dimension of the tree. Each leaf of the tree is the hash of the data from an ordered data set. Each node of the tree is the hash of the concatenation of the hash of its children. The root of the tree is the commitment to the whole data set. A proof of inclusion of certain data in the original commitment is an array of all the siblings from the leaf to the root. The code snippet below [2] can be used to construct Merkle tree, construct proof for a node and verify if the node belongs to the tree. Verifying Merkle proof takes $O(log|data|)$ steps.

```
type Node = {
  hash: Hash;
  children: Node[];
  parent: Node;
}

// n is the dimension of the tree
function constructMerkleTree(nodes: Node[], n: number = 2): Node {
  if(nodes.length === 1)
    return nodes[0];
  const parents: Node[] = [];
  for (let i = 0; i < nodes.length / n; i++) {
    const children = nodes.slice(i * n, (i + 1) * n);
    const hash = hash(concat(children));
    const n: Node = {
      hash,
      children,
    };
    parents.push(n);
    children.forEach(c => c.parent = n);
  }
  constructMerkleTree(parents, n);
}
```

```
// index is the index of the node in the data
function generateProof(node: Node, index: number, n: number = 2): Hash[][]
    {
  let result = [];
  while (node.parent) {
    const pos = index % n;
    const siblings = node.parent.children!.filter((_, i) => i !== pos);
    result.push(siblings.map(s => s.hash));
    node = node.parent;
    index = Math.floor(index / n);
  }
  return result;
}

// root is the initially committed root of the tree
// leaf is the hash of the data being verified
function verify(
  root: Hash,
  leaf: Hash,
  index: number,
  proof: Hash[][],
  n: number = 2
): boolean {
  let value = leaf;
  for (let i = 0; i < proof.length; i++) {
    const pos = Math.floor(index / n ** i) % n;
    // insert the value at the correct position
    proof[i].splice(pos, 0, value);
    value = hash(concat(proof[i]));
  }
  return isHashEqual(value, root);
}
```

Merkle Mountain Range (MMR). Merkle Trees can only be constructed for data sets which are exact power of n (tree dimension). A more general construction can be made using Merkle Mountain Range. Any natural number can be written as the sum of powers of n; data of any size can be broken into multiple powers of n. For each of those subsets, a Merkle Tree can be constructed. The roots of all Trees can be concatenated and hashed. This structure is called Merkle Mountain Range[34, 16]. The code snippet below [2] can be used to construct Merkle Mountain Range.

```
function constructMMR(nodes: Node[], n: number = 2) {
  const length = nodes.length;
  let leavesLeft = length;
  const merkleRoots = [];
  while (leavesLeft > 0) {
    const treeSize = n ** Math.floor(log(leavesLeft, n));
    const root = constructMerkleTree(
      nodes.slice(length - leavesLeft, length - leavesLeft + treeSize)
    );
    merkleRoots.push(root.hash);
    leavesLeft -= treeSize;
  }
  return hash(concat(merkleRoots));
}
```

Signing. It is referred to as generating a unique signature for a given data that a private key can only generate. Anyone using the public key, signature, and original data can efficiently verify this signature. Ethereum uses ECDSA (Elliptic Curve Digital Signature Algorithm)[18] of the SECP-256k1 curve[7, 36]. The PoS Ethereum on the consensus level uses bilinear pairing based BLS (Boneh–Lynn–Shacham) signature[2]. The application layer still uses the ECDSA. BLS signatures allow for multiple signatures generated using multiple public keys for multiple messages to be aggregated into a single signature, and verifying this aggregated signature is equivalent to verifying all the signatures[3, 2]. PoS Ethereum heavily uses this to compress signatures from multiple validators into a single signature verification.

Consensus Protocol. It is a set of rules which decide if a block enters a blockchain. The two mainly used classes of consensus protocols are PoW(Proof of Work) and PoS(Proof of Stake). In PoW, all the nodes are mining the next block by finding a nonce such that the hash of the block obeys specific difficulty criteria. PoW was first introduced in Bitcoin and later used by several blockchains like Ethereum[27, 7]. In PoS, a validator is selected randomly to propose a block. The selection is made such that the probability is proportional to the coins (stake) that you own in the system.

Blockheader. It is the data structure used to represent a block of the blockchain. A blockheader consists of the previous block's hash and the Merkle root of all the transactions included in the block. If the blockchain has a state like Ethereum, the blockheader would also have the Merkle root of the state[36, 7]. Several other values can be stored in the blockheader based on the consensus protocol design.

Smart Contracts. They are pieces of code deployed on the blockchain. They can have a balance and state associated with them. They have a set of defined functions which can be triggered through a transaction on the blockchain. The function can accept parameters, and tokens do an arbitrary computation, transfer funds, or call another contract.

Blockchain State. It refers to the state associated with smart contracts and address balances. In Ethereum, the complete state is saved as a Merkle Petria tree, and the tree's root is included in the header. In every block, the state is updated according to the transactions included in the block, and the state root is recalculated based on the updated tree.

Genesis Data. All the data used to bootstrap a blockchain and is available to all the nodes from the start is called Genesis Data.

Node. A blockchain network has different kinds of participants. The participants who verify the complete ledger and actively participate in the consensus are called Nodes or Validators. When a node bootstraps, it only has access to the genesis data. To propose a block to the blockchain, it has to first know the latest block in the chain. The process of arriving at the latest block is referred to as **sync**. The traditional way to sync is to verify every block in the blockchain and every transaction in that block. This has a time/storage complexity of $O(|t| + |n|)$ where $|t|$ is the number of transactions and $|n|$ is the number of blocks. The nodes which sync using this method are called full nodes.

Light Client. Network participants who do not actively propose blocks but would like to verify the latest state of the chain can sync in a more optimised way. They can only verify the chain's block headers, which reduces the time/storage complexity to $O(|n|)$. Such clients are called Light Clients or SPV (Simple Payment Verification) client.

Superlight Client. With increasing chain sizes, light clients can get insufficient to run on low-end devices such as smartphones and browser extinctions. Clients who can verify the chain in better than linear complexity are referred to Superlight clients.

Prover-Verifier Model. We are interested in a light client (verifier) that boots up with only the genesis data. The client connects to multiple full nodes (provers), which are fully synced to the latest state of the chain. The light client assumes that one out of all the full nodes it connects to is honest (non-eclipsing assumption) rest of the nodes can be controlled by an adversary.

Consensus-fork. It refers to the situation when some nodes decide to change the rules of the consensus protocol of the blockchain. This can lead to a situation where some nodes do not accept blocks mined by a set of other nodes in the network and vice versa. The network now has two separate blockchains with a common prefix in such a case. Consensus-fork happens when a new feature or patch has to be added to the blockchain. There are three main types of Consensus Forks: Hard-fork, Soft-fork, and Velvet-fork.

Hard-fork. It is a type of Consensus Fork where blocks produced by the updated node is not valid for the old nodes, and the blocks produced by the old nodes are not valid for the updated nodes. Hence every node that wants to participate in the new consensus has to update the new consensus rules. The blockchains highly avoid this as it requires every node in the network to update at the same time if they would like to continue participating in the upgraded chain. The protocol discussed in the paper is explicitly designed such that its implementation would not require any hard-fork of PoS Ethereum.

Ouroboros. It is the first provably secure construction for PoS blockchain [23]. The chain proceeds in terms of epochs, and every epoch has $16k$ slots. We denote S_j as the sequence of slots in an epoch. The block proposers for each epoch are chosen based on two parameters: the stake distribution of the validators SD_j and a random value η_j. The randomness is used to sample validators. The probability for a validator to get elected as a block proposer is proportional to the stakes owned in the system. The stakes distribution snapshot SD_j is taken at slot $S_{j-1}[8k]$. The randomness value η_j is generated based on slots $S_{j-1}[:10k]$. Hence, during every epoch's last $6k$ slots, all the block proposer of the next epoch is known. Two results from the Ouroboros Protocol will be used to show the working of Proofs of Proof of Stake.

Lemma 1 (Chain Growth)**.** *Consider any $2k$ consecutive slots. $k+1$ blocks will be generated, except with negligible probability in k.*

Lemma 2 (Honest Subsequence)**.** *Consider any $2k$ consecutive slots if any $k+1$ keys of the block proposers are taken; it is guaranteed to have at least one honest proposer except with negligible probability in k.*

Ethereum. It is the most used blockchain. It is the first blockchain to introduce Smart Contracts with a Turing complete programming language [7]. Smart Contracts have opened the gates to many decentralised applications called DAPPs. Smart contracts' most common use cases include creating fungible non-fungible tokens, decentralised organizations, decentralised finance, and name registry. Due to the popularity and usage of Ethereum, there is massive congestion in the network. Every transaction has to specify a bribe (maxPriorityFeePerGas) that it is willing to pay the miner to prioritize its entry into the blockchain. Due to the congestion in the network, the bribes have increased so much to make Ethereum unusable to ordinary users. Also, currently, Ethereum uses PoW, which is not sustainable for the environment. Ethereum is going through a major update to solve both of these problems. This update would switch the consensus to PoS consensus and then introduce Sharding, increasing the transactions per second (TPS) on the network. The upgrade is done in three steps: **Release of the Beacon Chain, Merge of PoW to Beacon Chain, Support of Sharding**. The first step has already been done successfully, and the Beacon Chain has been live since Dec 2020. In the rest of the paper, the term PoW Ethereum will refer to the current proof of work Ethereum chain, and PoS Ethereum will refer to the upgraded proof of stake Ethereum.

Consensus Layer. It refers to the part of the Ethereum Protocol that deals with the consensus protocol. The Ethereum upgrade changes are only made to the consensus layer of the protocol.

Application Layer. It refers to the part of the Ethereum Protocol that deals with Transactions, Balances, and Smart Contract. An ordinary user only interacts with the application layer. The Ethereum upgrade is done such that the application layer remains unchanged. This backward compatibility is maintained by merging the existing PoW chain to PoS. After the merge, the blockheader of the PoW chain will be included in the blockheader of the beacon chain.

Beacon Chain. It is a separate chain which is launched as the first step to lay the groundwork for the Ethereum upgrade. Beacon Chain uses PoS, and it will act like a mother chai. In future, it will include the block headers from the existing proof of work chain and the shard chains. The proof of stake protocol used by Beacon Chain is **Gasper**.

Time. Time in Beacon Chain is broken into chunks of 12 seconds called **slot**. Every 32 slots (6.4 minutes) is called an **epoch**. We assume that time is synchronous and clocks are fully synchronized.

Validators. They are referred to as nodes which are responsible for progressing the beacon chain. Anyone can become a validator by depositing 32 Ethers into the deposit contract on the PoW chain. Before the start of an epoch, 64 validators are chosen randomly out of the complete validator set as **block proposers** for each slot. Each block proposer is responsible for batching valid transactions and proposing a valid block in its slot time. Also, in every epoch, the complete validator set is divided into 64 disjoint sets of almost equal sizes called **committee**. The committee is responsible for providing attestations required for the progress of **Gasper** consensus at each slot.

Gasper. The Gasper protocol is a complex amalgamation of two protocols, Casper FFG and LMD Ghost. Casper FFG is a finality gadget. It allows for transactions to be optimistically committed to the chain and later finalised when sufficient attestations have been made. Casper FFG only works on the epoch level. It assumes that the validators use a fork choice rule to decide the tip of the chain during the epoch. The fork choice rule used by Gasper is Latest Message Driven Greediest Heaviest Observed SubTree (LMD GHOST)[9]. LMD GHOST is a greedy algorithm. It works by recursively choosing the sub-tree with the most unique validator attestations. In this paper, we will not use the complex machinery of the Gasper protocol itself. We will only use the theorems, assumptions, and slashability conditions made by Casper FFG to analyze the Sync Protocol.

Casper FFG. Butterin and Griffith introduced Casper the Friendly Finality Gadget in [8] and was used in Gasper[9]. The tool introduces the concept of justification and finalization.

- Every block has a $height$ starting from the genesis. Certain blocks in the chain are considered checkpoint blocks. We can assume that blocks with heights that are multiple of some constant k are checkpoint blocks. We define the checkpoint height h as the number of checkpoint blocks starting from genesis. The genesis block G has $h(G) = 0$.
- Attestation is signed messages containing $A \to B$ where A and B are checkpoint blocks such that the checkpoint height of A is less than the checkpoint height of B. Each attestation has a weight based on the validator's stake that signed it. The attestation can be considered as a signal to move from checkpoint A to checkpoint B
- A checkpoint block can have three states: Valid, Justified and Finalised. Initially, every checkpoint in the network is valid. It gets justified after the checkpoint has received certain attestations. Only checkpoints that are justified and have a certain property become finalised.
- Consider $J(G)$ as the set of all justified checkpoints and $F(G)$ as the set of all finalized checkpoints. All the finalised blocks must be justified hence $F(G) \subseteq J(G)$. Initially, only the genesis block G is finalised and justified.
- A checkpoints block B is considered justified if there exist attestations of the form $A \to B$ where $A \in J(G)$ (A is justified) and the total weight of such attestations is at least $2/3$ of the total validator stake. Attestation of the form $A \to B$ with at least $2/3$ of the total validator stakes is called supermajority link $A \xrightarrow{J} B$
- A checkpoints block B is considered finalised if $B \in J(G)$ and there exist a supermajority link $B \xrightarrow{J} C$ such that $h(C) = h(B) + 1$. Alternatively, a checkpoints block is finalized if it is justified and has attestation with $2/3$ weights from itself to the next checkpoints block.

Finally, Casper introduces **slashing conditions**. These are assumptions about honest validator, and if a validator W breaks these conditions, then its stake can be slashed by some other validator V by providing some proof of violation [9]. Following are the slashing conditions taken as it from the Gasper[9]:

- (S1) No validator makes two distinct attestations α_1 and α_2 corresponding to $s_1 \to t_1$ and $s_2 \to t_2$ respectively with $h(t_1) = h(t_2)$.

- (S2) No validator makes two distinct attestations α_1 and α_2 corresponding to $s_1 \to t_1$ and $s_2 \to t_2$, such that

$$h(s_1) < h(s_2) < h(t_2) < h(t_1)$$

Casper itself is just a finality gadget and not a complete protocol[9]. It assumes that there exists a fork choice rule that the validators use to choose the fork during the epoch and perform one and only attestation during the epoch. It assumes that a honest validator running the correct protocol will never break the slashing conditions and hence never get slashed. The main theorem of Casper (paraphrased by Gasper) are as follows:

Theorem 1 (Accountable Safety)**.** *"Two checkpoints on different branches cannot both be finalized, unless a set of validators owning stake above some total provably violated the protocol (and thus can be held accountable)".[9]*

Theorem 2 (Plausible Liveness)**.** *"It is always possible for new checkpoints to become finalized, provided that new blocks can be created by the underlying blockchain."[9]*

If the safety assumption is broken, then there must be $> 1/3$ validator stake that has broken (S1), and hence they can be slashed. The slashing conditions and main theorems of Gasper will be used in this paper to compare and analyze the safety and liveness of the sync protocol.

3 Construction

Let us define the problem statement of this paper for a general proof of stake blockchain setting. A client boots up and has access to only the genesis data of some PoS blockchain. The client would like to know the latest block-header of that chain. The client connects and interrogates with multiple full-nodes (provers). And one of the full-nodes is certainly honest. An honest prover has access to the complete blockchain until the latest block and answers honestly to the client's interrogations. The protocol used by the client should be secure and succinct. The security and succinctness of the protocol can be defined as follows:

Definition 1 (Secure)**.** A superlight sync protocol is secure if, after the execution of the protocol by a client, the client has the latest block-header that matches that of the honest full-node (prover)

Definition 2 (Succinctness)**.** A superlight sync protocol is succinct if the client can execute the protocol in $0(log|n|)$ storage/ time complexity.

3.1 Proofs of Proof of Stake

Let us assume an Ouroboros based PoS blockchain. The PoPoS construction requires certain modifications of the consensus protocol. In the Ouroboros protocol, all the block proposers for the next epoch are known in the last $6k$ blocks of the current epoch. For an epoch j lets consider slots $S_j[-6k : -4k]$. Lets call the block proposers of $S_j[-6k : -4k]$ as **handover leaders** and their sequence of public keys as pk_j. The handover leaders of epoch j already know the public key of all the block proposers of the next epoch; hence they also know pk_{j+1}. The PoPoS construction requires every block in $S_j[-6k : -4k]$ to include a **handover signature** σ. Handover signatures are defined as follows:

Definition 3 (Handover Signatures)**.** The handover signature for epoch j is a set of signatures by each block proposers of slots in $S_j[-6k : -4k]$ (handover leaders) to the same plain-text set of keys pk_{j+1} concatenated with the epoch index j.

It is called handover signatures, as the leaders from the epoch j hand over to the leaders of the epoch $j + 1$ by signing their public keys. The signatures form a chain of pk_j. The handover signatures σ_j act as a certificate for the correctness between leaders of epoch j and $j + 1$. The stake distribution SD_j and the random value η_j uniquely determines the block proposers for the slots S_j. All the handover signatures should be signing the same set of public keys pk_j. A malicious handover leader can choose not to propose a block, sign arbitrary data or even sign multiple conflicting signatures. An honest handover leader will only accept a chain with correct signatures so far and will sign the correct pk_j. To use handover signatures as a certificate to determine the leaders of the next epoch, we would like to have two properties to hold:

1. There exists a protocol to verify the correctness of pk_{j+1} based on σ_j all of which sign the same pk_{j+1}
2. For each epoch, there will exist one and only one pk_{j+1} for which property one holds.

Claim 1. *Given a set of $k+1$ correct handover signatures from block proposers of $S_j[-6k : -4k]$ signing the same pk_{j+1} we can be sure that pk_{j+1} is indeed correct.*

Proof. Using the Honest Subsequence[2] lemma we know that in $k + 1$ validators out of $2k$ slots at-least one of them is honest. As one of the signatures out of $k + 1$ is definitely correct and all of them sign the same pk_{j+1} therefore all of them must be correct. □

Claim 2. *There will always exists $k + 1$ handover signatures signing the same pk_{j+1} for every $S_j[-6k : -4k]$.*

Proof. The Chain Growth[1] lemma guarantees that during 2k consecutive slots, at least $k+1$ blocks will be generated. As the $k+1$ blocks have a causal relationship, they should have signatures signing the same pk_j. □

Now we know that given $k + 1$ handover signatures we can be certain about the pk_{j+1}. The handover signatures act as a succinct proof for the correctness of verifying $2k$ block proposers of the next epoch given the $2k$ block proposers of the current epoch.

Remark. *We concatenate the epoch index to the signed message of the handover signatures. This is required to avoid attacks where for some $l < m$, $S_l[-6k : -4k]$ is same as $S_m[-6k : -4k]$. In such a case, an attacker can use the σ_l in place of σ_m and replay the previous handover chain, and a client would not be able to differentiate the honest chain from this modified chain. In practice, this should not happen as it is assumed that the stakes are fairly distributed, and the probability for the same set of validators to be chosen for $2k$ slots should be negligible.*

Linear Verifier. Let us build a verifier using the handover signatures. This verifier will not be significantly better than a traditional light client but will play a pivotal role in understanding the optimised construction. A client bootstraps where he only knows the block proposers of the genesis epoch. The client connects to a prover which can present all the set of public pk_j and handover signatures σ_j for each epoch. The client can then sequentially go over the pk_j for each epoch. The client already knows pk_0 from the genesis proposer set. For each consecutive epoch j the client checks if pk_j correspond to $2k$ public keys and there exist $k+1$ valid signatures from pk_{j-1} to pk_j. The client finally ends at pk_m where m is the upcoming epoch, and pk_m is valid block proposers of the $2k$ slots of that epoch. Now the client can verify the block headers produced in that $2k$ slots of the epoch.

For the above linear verifier, if we look at pk_j presented by an honest proposer as compared to the pk^*_j presented by a malicious prover, a malicious prover could have changed pk^*_j starting from some i till the end to make the client believe about some false proposer set. The linear verifier would have discovered this problem while verifying the set of public keys pk^*_i at step i as the prover should not be able to valid $k+1$ handover signatures from epoch validator set of $i-1$ to i. This holds because of Lemma 2 that there should be at least one honest prover out of the $k+1$ proposers, which will not perform malicious signing.

Given two sequence $(pk_0, pk_1, ...pk_m)$ and $(pk_0, pk_1, ...pk^*_m)$ for each epoch and its corresponding sequences of handover signatures $(\sigma_0, \sigma_1, ...\sigma_{m-1})$ and $(\sigma^*_0, \sigma^*_1, ...\sigma^*_{m-1})$. And it is known that one of the sequences is certainly correct. We can find the correct sequence in two steps:

1. Find the first point of disagreement y between the two sets such that pk_y is not equal to pk^*_y
2. Check the correctness the signature σ_{y-1} and σ^*_{y-1}. Only one of them should be valid.

Anything before the first point of disagreement is the same in both sequences. Given that one of them is correct, both are correct until this point. We must choose between the two sequences only at the first point of disagreement. As only one of them should be able to present $k+1$ correct handover signatures, it is sufficient to check the correctness of one of the handover signatures to decide which sequence is correct. A naive way to find the first point of disagreement is to go sequentially from the start and compare the elements of both sequences one by one. This approach will require linear complexity. Step 2, the signature correctness check of the previous epoch has constant complexity. Hence the complexity of the overall protocol will be linear. If we can come up with an efficient way to find the first point of disagreement, we should be able to improve the complexity of the protocol. For this, we will use interactive Bisection Games.

3.2 Bisection Games

Lets again consider the two provers with set of handover public keys as $(pk_0, pk_1, ...pk_m)$ and $(pk_0, pk_1, ...pk^*_m)$ and set of handover signatures as $(\sigma_0, \sigma_1, ...\sigma_{m-1})$ and $(\sigma^*_0, \sigma^*_1, ...\sigma^*_{m-1})$. One of the provers is honest, and the other is malicious. First, each of them should create a Merkle tree where the tree's leaf at index i is pk_i concatenated with epoch index i. Let us assume a superlight verifier is bootstrapping, and it connects to both of these provers. The superlight client will first ask for the root of the Merkle tree. As both the sequences are not the same, the tree's roots must be different. The verifier should then ask each prover to present the root's children. The verifier can check the children provided by each prover by simply hashing the children and comparing them with the root. The verifier then compares each child provided by both provers starting from left to right. One of the children must be different as a hash of the root is different. The verifier finds the first child, which is different from the left side. The verifier then asks both the provers to open the children of this node. This process continues until the leaf of the tree is reached. This leaf is the first point of disagreement. We can imagine the bisection game as an iterative zoom-based check where we compare if the left or the right half of the complete tree is different in the first step. Then, based on the differing side, we zoom in to look only at that subtree and ask the same question on the left and right sides of the subtree. We zoom and compare the subtrees iteratively until we reach the node where we can not zoom anymore. This node is the first point of disagreement. If the node doesn't respond in time or provides children which don't match the correct parent hash they immediately lose the bisection game. The below code snippet of function treeVsTree is a recursive implementation of the bisection game taken from the implementation.

```
// returs the index for the first point of disaggreement
// If one of the provers responsds with incorrect children
// then the other one simply wins. Returns true if the
// prover1 wins
function async treeVsTree(
    prover1: IProver<T>,
    prover2: IProver<T>,
    tree1: Uint8Array,
    tree2: Uint8Array,
    node1: Uint8Array = tree1,
    node2: Uint8Array = tree2,
    index: number = 0,
): Promise<boolean | number> {
  // get node info
  const nodeInfo1 = await prover1.getNode(tree1, node1);
  const nodeInfo2 = await prover2.getNode(tree2, node2);

  if (nodeInfo1.isLeaf !== nodeInfo2.isLeaf)
    throw new Error('tree of unequal heights recieved');

  // if you reach the leaf then this is the first point of disagreement
  if (nodeInfo1.isLeaf) {
    return index;
  } else {
    const children1 = nodeInfo1.children!;
    const children2 = nodeInfo2.children!;

    // check the children are correct
    const parentHash1 = digest(concatUint8Array(children1));
    if (!isUint8ArrayEq(parentHash1, node1))
      return false;

    const parentHash2 = digest(concatUint8Array(children2));
    if (!isUint8ArrayEq(parentHash2, node2))
      return true;

    // find the first point of disagreement
    for (let i = 0; i < this.n; i++) {
      if (!isUint8ArrayEq(children1[i], children2[i])) {
        // recurrsively apply treeVsTree to get to the leaf
        return await this.treeVsTree(
          prover1,
          prover2,
          tree1,
          tree2,
          children1[i],
          children2[i],
          index * this.n + i,
        );
      }
    }
    throw new Error('all the children can not be same');
  }
}
```

The number of interactions required to reach the tree's leaf should be equal to the dept of the tree. Let us consider n as the dimension of the tree, and m is the number of leaves in the tree. The dept of the tree is $log_n m$. Using bisection games, we can improve the complexity of the linear verifier protocol to log.

3.3 Sync Committee & Light Client

Building light clients for a PoS system is much more complex than that for a PoW system. In a PoW system, it is sufficient to verify that the nonce in the block header leads to a block hash under the difficulty range, and the blocks are connected sequentially starting from the genesis. This construction works because generating a valid chain with the above properties is as difficult as generating the original chain. Whereas in the PoS system, we need to know the correct block proposers for the slot to verify the block header. The stake distribution SD_j and random value η_j for an epoch determine the block-proposer for slots. The block-headers of epoch j might contain the Merkle roots of SD_j and the η_j. It is not clear how a light client can verify the correctness of the Merkle roots SD_j and η_j without the access to the complete SD_{j-1}, η_{j-1} and then linearly applying the stake changes to SD_{j-1} for every tx (transaction) in epoch $j-1$. However, this will defy the purpose of building a light client as the complexity would include verification of every transaction.

For Oroborous protocol SD_j and η_j are already fixed 6k slots before the start of the epoch. This allows to commit the merkle roots of SD_j and η_j in slots $S_{j-1}[-6k : -4k]$. A similar argument to PoPoS construction using Honest Subsequence [2] and Chain Growth [1] can be made to prove the correctness of $k+1$ blocks. Hence using this trick, for Oroborous protocol SD_j and η_j can be verified without knowing SD_{j-1}, η_{j-1} and linearly applying all tx from epoch $j-1$.

Gasper does not guarantee the Honest Subsequence [2] or the Chain Growth[1]; hence the similar argument as Oroborous cannot be applied to build a light client for Gasper. To allow the Ethereum Beacon chain, based on Gasper, to be light client-friendly **Sync protocol** was introduced in the Altair Ethereum upgrade [14]. The Sync protocol adds two components to the specification:

1. Add sync committee role for a set of validators with appropriate participation rewards and penalties.
2. Specify a minimal light client for beacon chain based on sync committee.

Sync committee is a set of 512 validators chosen randomly from the complete validator set. Validators are selected to become a sync committee based on the stakes owned by the validator. A sync committee is fixed for 256 epochs ($\sim$ 1day), also referred to **Sync Period**. The sync committee is responsible for signing the tip of the beacon chain for every slot during the sync period. The block proposer of the next slot aggregates the sync committee signatures from the last slot and includes the aggregated signature in the block-header. Precisely the BeaconBlockBody (data structure for beacon chain block-header) consist of **sync_aggregate** of type SyncAggregate. The SyncAggregate is struct with two parameters **sync_committee_bits** and **sync_committee_signature**. The sync_committee_bits is a bit vector of size 512 representing the participation of the sync committee for the previous slot, and sync_committee_signature is a BLS aggregate signature from those participants. Each sync committee participant is rewarded a fixed amount for every sync signature that is included and penalized if the participant fails to submit the sync signature. The reward amount is the same as the penalty amount.

The sync committee is known one period in advance. The **current_sync_committee** and **next_sync_committee** is stored in the BeaconState and the Merkle root of the the BeaconState is stored in the BeaconBlockBody. As the current sync committee signs the BeaconBlockBody for blocks in the period, a Merkle inclusion proof can be made of the next_sync_committee to state root in BeaconBlockBody, the current sync committee indirectly attests to the next sync committee. The sync committee signatures form a chain of sync committees starting from the genesis sync committee, where every committee signs the next committee. The sync committee and sync signatures are similar to the epoch leaders and handover signatures from the PoPoS construction.

Light Client Once we have a chain of sync committees, we can verify the sync committee for each period instead of verifying the beacon chain blocks themselves. The sync committee chain allows verifying the beacon chain without using the stake distribution table. Also, the sync committee changes slowly (once every day), so the light client has to verify only one committee per period. Before we formalize the construction of the light client, let us define finality in terms of the sync committee.

Definition 4 (Sync Finalized)**.** A block is Sync Finalized if and only if it has 2/3 valid signatures from the current sync committee.

Remark. *The Sync Finalized is not the same as the finalization of Gasper. The Gasper finalization provides much more security. The official specification refers to both finalizations as just finalized, but we use "Sync Finalized" to make the distinction for this paper.*

The light client specified by the Ethereum consensus repository works by iteratively verifying the sync committee for each period starting from the genesis. Let us suppose the light client knows the sync committee for period n to verify the sync committee for the period $n + 1$ it only needs a single sync finalized block from period n. Once it gets such a block and sync signatures, the light client can check the finality conditions and confirm the next sync committee available in the BeaconBlockBody. The light client can do this until it reaches the latest sync committee. Once it has verified the latest sync committee, it can simply verify any finalised block in this period. The finalized block, sync aggregate signatures, the Merkle inclusion of next_sync_committee into the BeaconBlockBody, and some metadata is packed into a single data structure referred to as **SyncUpdate** in the specification. This structure allows for building a standard API exposed by the full nodes such that light clients can efficiently query all the data required to sync.

```
class LightClientUpdate(Container):
  # The beacon block header that is attested to by the sync committee
  attested_header: BeaconBlockHeader
  # Next sync committee corresponding to the active header
  next_sync_committee: SyncCommittee
```

```
next_sync_committee_branch: Vector[Bytes32, floorlog2(
    NEXT_SYNC_COMMITTEE_INDEX)]
# The finalized beacon block header attested to by Merkle branch
finalized_header: BeaconBlockHeader
finality_branch: Vector[Bytes32, floorlog2(FINALIZED_ROOT_INDEX)]
# Sync committee aggregate signature
sync_aggregate: SyncAggregate
# Fork version for the aggregate signature
fork_version: Version
```

Listing 1: The above code is from the light client specification [14]

There can be cases certain blocks do not surpass the 2/3 sync committee requirement. Such cases can happen if a set of sync committee nodes is offline, or if signatures did not arrive to the block proposer because of network delay, or if the block proposer is malicious. In such a case, the block cannot be used to verify the next sync committee. The current official specification still recommends using some heuristic to accept such a block for verifying the latest block header.

The Gasper consensus used by PoS Ethereum provides safety guarantees that no conflicting blocks can get finalized without 1/3 slashibility and proves probabilistic liveness. The current specification does not prove the safety and liveness of the sync protocol. Based on the safety and liveness assumption of the Gasper protocol, we would like to understand the safety and liveness of the sync protocol.

Definition 5 (Safety of Sync Protocol)**.** There will be no conflicting finalised blocks.

Definition 6 (Liveness of Sync Protocol)**.** There will exist at least one sync finalized block per sync period.

For the safety of the sync protocol, we would like to have no two conflicting blocks (block in different branch with a common ancestor) finalized. Conflicting blocks can have different sync committees, making it impossible for an external light client to decide which of them is correct. If there are two conflicting finalized blocks, then there must 1/3 of the sync committee that has signed conflicting blocks. The Gasper protocol assumes that 2/3 honest stake majority and the sync committee validators are randomly chosen out of this validator set. Assuming that the underlying validator set has 2/3 honest majority, the sync committee should also have the same distribution. However, an honest Gasper node might behave malicious sync committee node. Gasper enforces honesty by slashing, whereas there is no slashing in the sync protocol for signing conflicting blocks. Also, sync protocol is not secure against adaptive adversaries. An adaptive adversary can make a targeted attack on an individual sync committee by bribing an individual committee to behave maliciously. The protocol should still be secure with an adaptive adversary where the adaptivity rate is more than one period plus network delay. We can consider the lower limit to be two periods to be safe.

For the liveness of the sync protocol, we would like to have at least one sync finalized block per sync period. The sync finalized block is required for the light client to verify the next sync committee. If there is no sync finalized block in the period, the light client cannot verify the next sync committee. Even if the underlying consensus is live, it does not guarantee the liveness of sync protocol. There is a penalty for the sync committee for every slot in the sync period that they do not sign. Thought hypothetically, there can be an incentive for not signing by hacking the light client based on sync protocol, and this incentive could be higher than the penalty. The empirical data in plot 1 shows that the average sync committee participation during each period is much higher than the 2/3s margin.

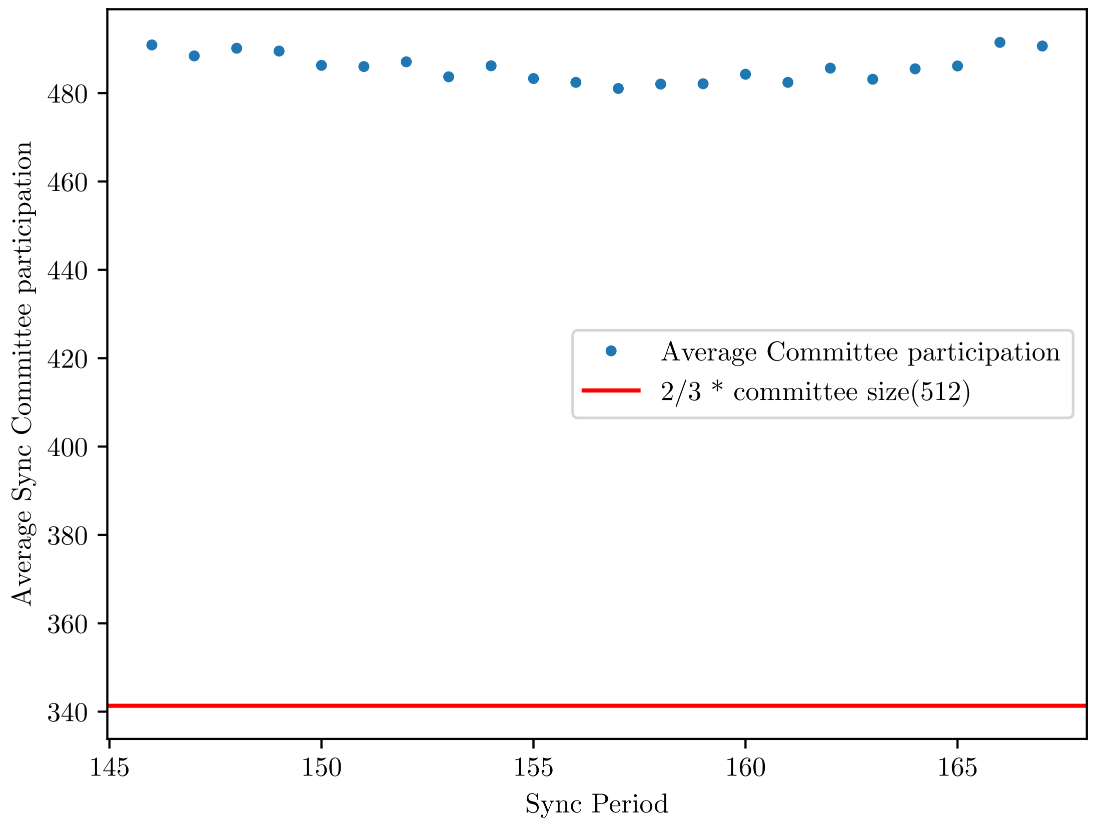

Figure 1: The plot shows the average sync committee participation of for different sync periods

For the PoS Ethereum superlight client construction, we would like to have both the safety and liveness of the sync protocol. The construction explained in this paper would also be vulnerable to safety and liveness issues that exist on the sync protocol or attacks on the PoS Ethereum consensus[29, 28]. If the Ethereum protocol fixes these issues in future without removing the sync protocol itself, the superlight client would also benefit in the same way as the light client.

3.4 Superlight Client for PoS Ethereum

The main drawback of the PoPoS construction is that it requires changes in the consensus protocol, which would need a hard-fork. The blockchain community does not appreciate hard-forks, and hard-fork requires all the validators to upgrade their software to support the new consensus changes. The security of the blockchain is majorly reduced during the upgrade process. The network is divided into validators who have upgraded their software and validators who have not. It is also a complex process to propose a change that would require a hard-fork, and it might take months for it to be accepted and released. This paper proposes a protocol that does not require any hard-fork.

If we can find an analogous check to the handover scheme in the current PoS Ethereum protocol, we can achieve the same benefits without a hard-fork. As discussed in the previous section, the sync protocol provides a simple way to sync to the latest state of the chain by quickly verifying the sync chain. Let us consider the public keys of the sync committee at period j as c_j and any valid 2/3 signatures of a finalised block from period j as τ_j. We can modify the PoPoS construction to work with sync protocol as follows:

1. Instead of using the sequence $(pk_0, pk_1, ...pk_m)$ to construct the merkle tree we will use $(c_0, c_1, ...c_l)$ where l is the last period
2. Instead of using σ_{j-1} (handover signatures) to validate pk_j we will use τ_{j-1} (sync committee signature).
3. τ_j is valid if it is a set of valid signature from $> 2/3$ sync committee of period j

The primary protocol remains the same. The client connects with two provers. One of them is surely honest. It asks for the Merkle roots of sync committee chains. If the roots are the same, then both the provers are honest, and the last committee of any one of them could be used. If they are not the same, the provers go through a bisection game. At the end of the bisection game, either one of the provers has already lost, or the client knows the first point of disagreement j. Once the client knows the first point of disagreement, then it remains to get the previous committee c_{j-1} and τ_{j-1} from both provers. From the safety assumption of the sync protocol, only one of the provers should be able to present the valid signatures.

3.4.1 Handling different chain sizes in case of network delays

While describing the PoPoS construction, we assumed that all the provers have the exact tree sizes. The tree sizes might not be the same, even for the honest nodes. The honest nodes can have ledgers at different latest epochs because of the network delay. The common prefix property of the blockchain still guarantees that two honest chains will only differ by k blocks. In the case of our superlight client construction, as the sync period is roughly one day, it is big enough to overcome any network delay between nodes. Some honest nodes might not have seen a sync finalized block during the period boundaries. Hence, honest nodes can differ by max one sync period in such cases. To avoid this issue, we can do the following. To verify the latest sync committee c_l, we only need one sync finalized block from period $l - 1$. The nodes might not have a sync finalized block from period l, but they must have a sync finalized block from the previous period $l - 1$. Also, the current sync period can be calculated based on the current timestamp. Hence in our construction, the prover can calculate the current period and construct the committee Merkle tree where the last committee is the current committee which can be verified using the sync finalized block from the last period. The client can also calculate the current sync period based on the timestamp and will know beforehand the size of the tree of an honest prover. Provers and Clients can still have clocks out of sync on the period boundaries. To solve this, the client should not sync for a small window of time greater than the network delay on the period boundaries.

3.4.2 Bisection games using Merkle Mountain Ranges

We have used Merkle Trees for bisection games in the construction so far. The latest committee index might not be a perfect power of tree dimension to construct the Merkle tree. In such a case, Merkle Mountain Range can be used. For any natural numbers n and m, n can be written as powers of m. This allows to take any set and break it into sets where each set has elements in perfect power of tree dimension. Using a Merkle mountain range, we can create a set of trees that represent the current committee index. The trees inside a Merkle mountain range are referred to as peaks. Now we will modify our initial bisection game construction to account for the Merkle mountain ranges. First, instead of using the root of the Merkle tree as the initial commitment sent by the provers to the client, we can use the root of the Merkle mountain range. The root of the Merkle mountain range is the hash of the concatenated roots of its peaks. Once the client has compared the root of the Merkle mountain range, it can compare the peaks. The client can sequentially check peaks from the left (biggest) to the right (shortest), and at the first point where the peak differs between the provers, it can run the bisection game algorithm (treeVsTree) on the Merkle trees. Below is a code snippet of function peaksVsPeaks which implements the comparison of peaks and calls the treeVsTree function.

```
// returns true if the prover one wins
function async peaksVsPeaks(
  prover1: IProver<T>,
  prover2: IProver<T>,
  peaks1: Peaks,
  peaks2: Peaks,
): Promise<boolean> {
  let offset = 0;
  for (let i = 0; i < peaks1.length; i++) {
    // check the first peak of disagreement
    if (!isUint8ArrayEq(peaks1[i].rootHash, peaks2[i].rootHash)) {
      // run tree vs tree bisection game
      const winnerOrIndexOfDifference = await this.treeVsTree(
        prover1,
        prover2,
        peaks1[i].rootHash,
        peaks2[i].rootHash,
      );
      if (typeof winnerOrIndexOfDifference === 'boolean')
        return winnerOrIndexOfDifference;
      else
        return this.checkNodeAndPrevUpdate(
          prover1,
          prover2,
          peaks1,
          peaks2,
          winnerOrIndexOfDifference + offset,
        );
    }
    offset += peaks1[i].size;
  }
}
```

3.4.3 Tournament among multiple provers

So far, we only discussed the case where there are two provers. We will try to relax this assumption. Let us assume that a client connects to multiple provers. The standard non-eclipsing assumption guarantees that at least one of the provers the client connects to is honest. All honest provers will construct the exactly same Merkle mountain root. Also, multiple malicious provers can have the same Merkle mountain root. We will do a tournament among provers by iteratively applying the original protocol. Let us assume we have a list of current winners. In the start, it simply has the first prover. We take one of the existing winners and the next prover. We first compare the Merkle mountain root. If the roots are the same, we add the prover to the list of winners. Otherwise, we go through peaksVsPeaks. If the new prover wins, we clear all the existing winners and add the new prover to the winner's list. If the new prover loses, we discard it. The honest prover should never lose. Hence, in the end, we are left with a list of provers that are honest and have the same Merkle mountain root. Below is the code snippet for the tournament.

```
type ProverInfo = {
  root: Uint8Array;
  peaks: Peaks;
  syncCommittee: Uint8Array[];
  index: number;
};

// returns the prover info of the honest provers
function tournament(proverInfos: ProverInfo[]): Promise<ProverInfo[]> {
  let winners = [proverInfos[0]];
  for (let i = 1; i < proverInfos.length; i++) {
    // Consider one of the winner for the current round
    const currWinner = winners[0];
    const currProver = proverInfos[i];
    if (isUint8ArrayEq(currWinner.root, currProver.root)) {
      // if the prover has the same root as the current
      // winners simply add it to list of winners
      winners.push(currProver);
    } else {
      const areCurrentWinnersHonest = await this.peaksVsPeaks(
        this.provers[currWinner.index],
        this.provers[currProver.index],
        currWinner.peaks,
        currProver.peaks,
      );
      // If the winner lost discard all the existing winners
      if (!areCurrentWinnersHonest)
        winners = [currProver];
    }
  }
  return winners;
}
```

4 Implementation and Evaluation

4.1 Implementation and Design

In this paper, not only do we propose a construction to build a superlight client for PoS Ethereum, but we also build the first implementation of a superlight client for a proof of stake blockchain. The primary motivation behind the implementation is to build a proof of concept that we can use to compare the light client and the superlight client. However, we wanted to build it to be extendable to a production-grade wallet. The implementation is done using NodeJs and Typescript. There are four primary entities in the implementation: Prover, Client, ProverStore, and VerifierStore.

Dummy Chain. Currently, the beacon chain with the sync protocol (after Altair Fork) is only 155 periods (155 days) old. This chain is not sufficiently long to see asymptotic effects superlight clients over light clients. Hence, we created a dummy sync committee chain comparable to the Beacon Chain, which can be extended to higher periods. We performed tests on both the dummy chain and the original beacon chain. We used a seedable random number generator to generate random BLS private keys for the sync committee of each period in the dummy chain. We then used the private keys to sign the public keys for the subsequent period. We used a seedable random number to ensure the same sync committee chain can be generated by multiple provers independently.

Malicious Chain. To test the working of the superlight client, we wanted some malicious provers. A malicious prover would be the same as an honest prover but serve malicious data. We built the malicious data in the following way. First, we generate or load the honest chain. Then we chose a random index in the range of sync periods. We override the existing committee with a random sync committee for each period starting from this index. The updated chain now has incorrect committee signatures starting from this index.

Prover Store. Any class that follows the *IProverStore* interface is a prover store. The store consists of the getters for the sync committee and the sync committee signature for each period. The prover store abstracts the chain specific implementation from the Prover. There are two implementations of the prover store: Beacon Store and Dummy Store. The beacon store has the beacon chain data, and the dummy store has the dummy chain data. Also, both of the stores accept a boolean *honest* as a constructor parameter. If it is set to false the store will serve malicious data.

```
export interface IProverStore<T> {
  getAllSyncCommittees(): {
    startPeriod: number;
    syncCommittees: Uint8Array[][];
  };

  getSyncCommittee(period: number): Uint8Array[];

  getSyncUpdate(period: number): T;

  updateToJson(update: T): any;
}
```

Prover. Any class that follows *IProver* interface is a *Prover*. The *IProver* interface exposes minimalistic set of functions that should be exposed by a Node to allow for a light client or superlight client for sync. The prover accepts an implementation IBeaconStore as a constructor parameter. This allows the prover to be abstracted of the chain itself. The prover constructs the merkle mountain ranges and answer the client queries about the merkle mountain range, sync committee and sync committee signatures. In the actual implementation the prover implementation is wrapped with an REST server using express framework. On the client side the REST client is made which calls the express server but follows the IProver interface. This allows for the client to completely abstract the complexity of handling HTTP requests, parsing, and failures.

```
interface IProver<T> {
  getLeafWithProof(period: number | 'latest'): AsyncOrSync<{
    syncCommittee: Uint8Array[];
    rootHash: Uint8Array;
    proof: Uint8Array[][];
  }>;

  getMMRInfo(): AsyncOrSync<{
    rootHash: Uint8Array;
    peaks: Peaks;
  }>;

  getNode(
    treeRoot: Uint8Array,
    nodeHash: Uint8Array,
  ): AsyncOrSync<{ isLeaf: boolean; children?: Uint8Array[] }>;

  getSyncUpdate(period: number | 'latest'): AsyncOrSync<T>;

  getSyncUpdateWithNextCommittee(
    period: number,
  ): AsyncOrSync<{ update: T; syncCommittee: Uint8Array[] }>;
}
```

VerifierStore. Any class that follows the *IVeriferStore* is a VerifierStore. A VerifierStore has chain specific functions like the getters for genesis data and sync committee signature verification that a client can use. The VerifierStore allows the client to build an implementation abstracted from the underlying chain.

```
export interface IVeriferStore<T> {
  syncUpdateVerify(
    prevCommittee: Uint8Array[],
    currentCommittee: Uint8Array[],
    update: T,
  ): boolean;

  getGenesisSyncCommittee(): Uint8Array[];

  getCurrentPeriod(): number;

  getGenesisPeriod(): number;

  updateFromJson(jsonUpdate: any): T;
}
```

Client. A client accepts a list of provers following the IProver interface as a constructor parameter and exposes a sync function which returns the honest provers with the latest sync committee. We made two implementations of the client: Light client and Superlight client. The Light client was inspired by the lodestar light client implementation [32]. The superlight client follows the construction discussed in the paper.

Remark. *Interactivity in the benchmarks below is referred to the numbers of rounds of prover client communication. In our current light client implementation we do a single request to fetch the sync update for each period. This leads to a linear increase in interactivity by the light client with increasing periods. The light client itself doesn't require interactivity and can fetch all the sync updates in one shot. In practice fetching sync updates are batched [32]. Hence the number of interactions by the light client depends on the implementation. This is not the case for the superlight client as the superlight client needs to perform bisection games which require the interactions. This is also a demerit of our superlight client construction.*

4.2 Benchmarks

To benchmark the light client vs superlight client following setup was made:

- Fourteen provers were created; each had 512 MB RAM. The prover servers were deployed using *Heroku Platform* on *free dynos*.
- Thirteen provers were malicious, and one prover was honest.
- The client was running on MacBook Pro 6-Core Intel Core i9 Processor 32 GB RAM with 40 Mbps internet speed.
- The For each setup, ten trials were made and for each trial, the time to sync, bytes downloaded, and interactions were recorded.
- For dummy data, all provers had the same seed to generate the same honest chain.

Beacon Chain Test. The first benchmark was performed on the Beacon Chain data with 155 periods and 8 provers (7 malicious and 1 honest). The table 1 shows the results of the benchmark. There was a 40% reduction in time to sync and an 8x reduction in data downloaded in superlight client implementation compared to the light client implementation.

Implementation	Time to sync	Bytes Downloaded	Interactions
Light Client	110.72 ± 7.00s	16.83 ± 0.21MB	159.60 ± 1.96
Superlight Client	62.63 ± 0.85s	2.29 ± 0.00MB	163.00 ± 0.00

Table 1: The table above shows the benchmark between light client and superlight client on beacon chain data

Dummy Chain Test. The second benchmark was performed on the Dummy Chain data with 1024 periods (2.8 years) and 8 provers (7 malicious and 1 honest). The table 2 shows the results of the benchmark. Compared to the light client implementation, there was a 9x reduction in time to sync and a 47x reduction in data downloaded in the superlight client implementation. We also did tests with chain size 256 and 512 and plotted the time to sync 2, bytes downloaded 3 and interactions 4 w.r.t to different chain sizes. The plots clearly show that the performance difference between light clients and the superlight client gets more and more significant with increasing periods.

Implementation	Time to sync	Bytes Downloaded	Interactions
Light Client	803.42 ± 3.69s	106.58 ± 0.22MB	1027.60 ± 2.11
Superlight Client	92.95 ± 1.97s	2.27 ± 0.00MB	205.00 ± 0.00

Table 2: The table above shows the benchmark between light client and superlight client on dummy chain data

Remark. *Notice that the data downloaded by the superlight client for the dummy chain is smaller than that of the Beacon chain, even though the dummy chain has a higher number of periods. The reduction is because the beacon chain implementation uses the SyncUpdate data type proposed in the sync protocol specs to send the sync committee* *signatures. This data structure consists of params not there in the dummy chain. We used SyncUpdate data structure instead of creating a new structure to have the superlight client implement as close to the sync protocol specification.*

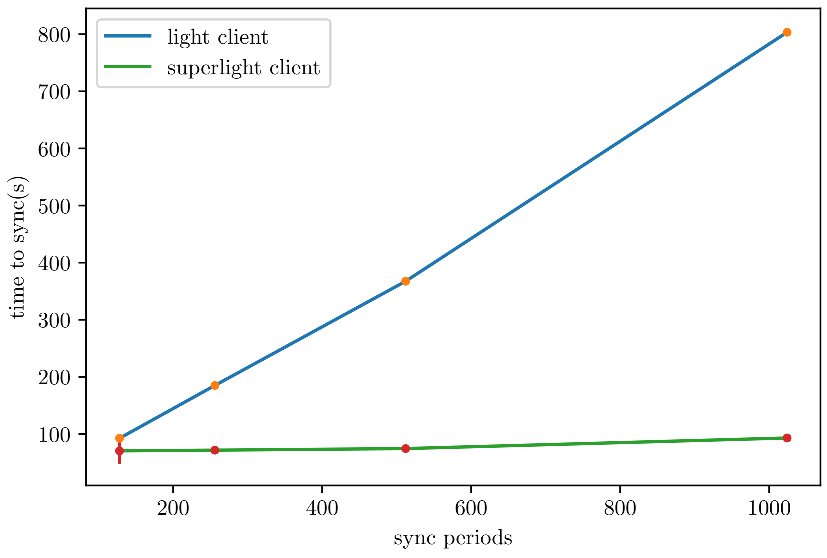

Figure 2: The plot shows time to sync for light client and superlight client w.r.t increasing sync periods

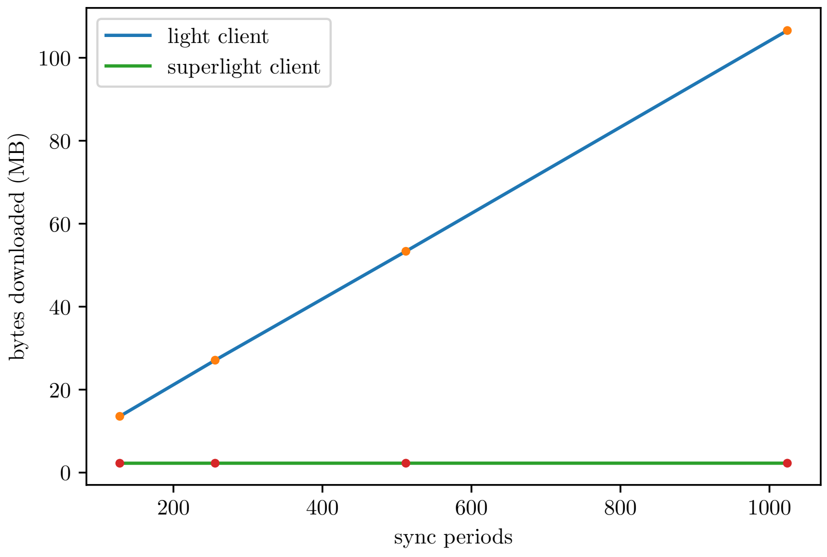

Figure 3: The plot shows bytes downloaded by light client and superlight client w.r.t increasing sync periods

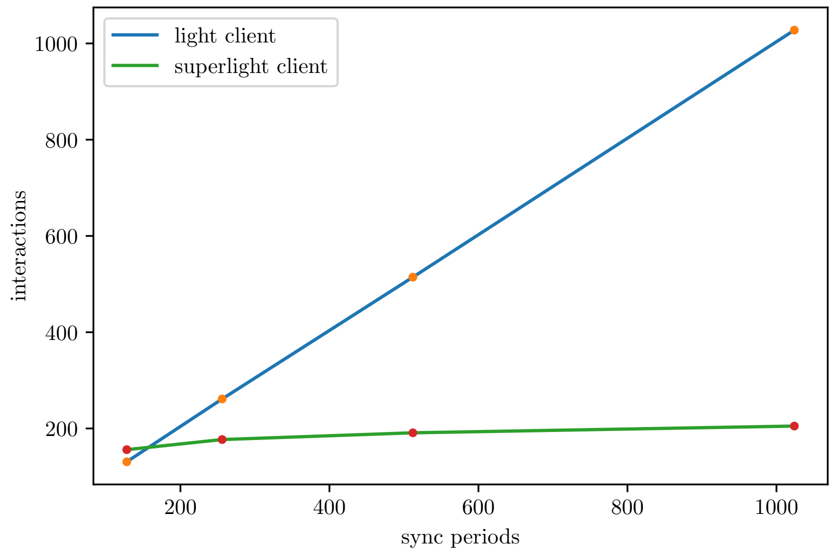

Figure 4: The plot interaction by light client and superlight client w.r.t increasing sync periods

Prover Count The third benchmark was performed to compare the time to sync, bytes downloaded and interactions of superlight client with different number of provers. The results are plotted in figure 5 6 7. The results show a linear increase in all the parameteres.

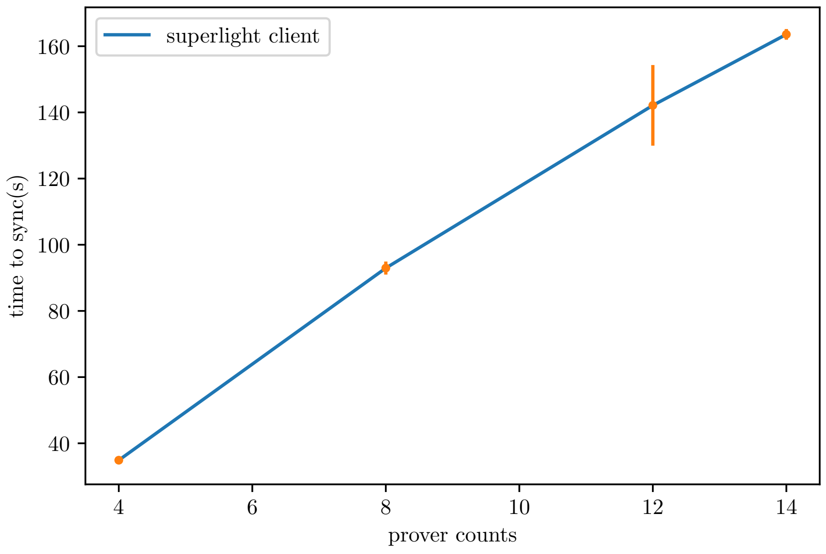

Figure 5: The plot shows time to sync for superlight client w.r.t increasing number of provers

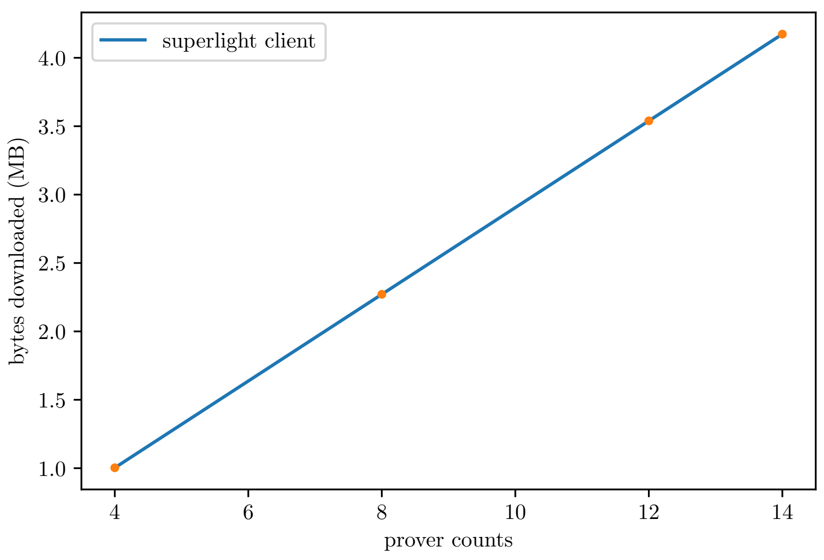

Figure 6: The plot shows bytes downloaded for superlight client w.r.t increasing number of provers

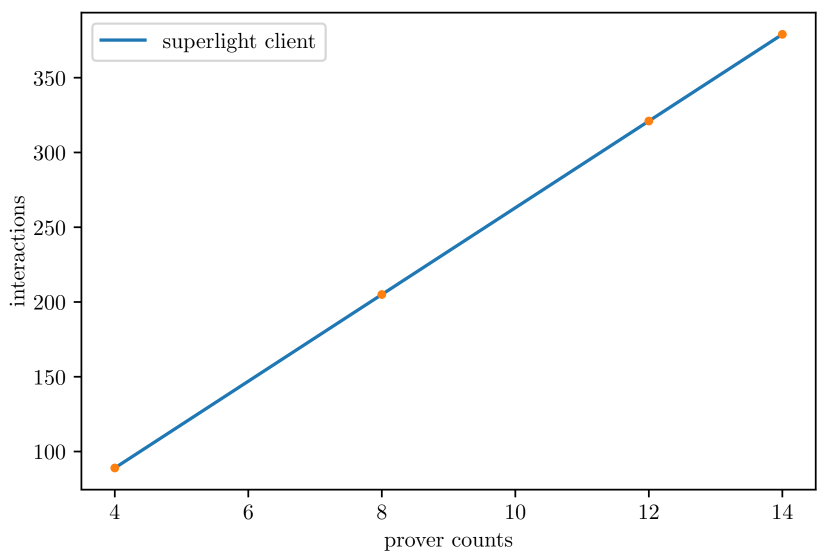

Figure 7: The plot shows interactions for superlight client w.r.t increasing number of provers

5 Conclusions

This paper proposed a superlight client construction for PoS Ethereum, which does not require a hard-fork. The results clearly show that the superlight client performs significantly better than the light client. Also, the benchmark shows that the superlight client can be implemented on a low-end device. Currently, all the wallets of PoW Ethereum use third-party providers, and these providers become a central point of failure and pose a huge risk to the Ethereum users. The implementation proposed in the paper can be easily extended to build secure wallets for PoS Ethereum. The construction proposed in this paper can also be extended to build cross-chain bridges where a verifier is a smart contract on another blockchain[22].

5.1 Other Approaches

We also explored alternate approaches to building superlight clients. Two of the notable approaches were:

- Using Casper FFG signatures for handover. The main benefit of doing so is that we do not have to make sync protocol's safety and liveness assumptions. The finalization of Casper FFG can provide the same level of safety to the superlight client as the full nodes. To use Casper FFG signature for handover, we would need a way to verify a finalized block based on the last finalized block. Casper FFG only provides plausible liveness, and hence the maximum slot count between two finalized blocks is uncertain. Also, the validator set changes continuously, making it challenging to use Casper FFG for handover.
- Using Progressive SNARKS instead of interactive bisection game. The main benefit of using SNARK is that it is not interactive and only needs constant time/space complexity. The SNARK based client would be even exponentially better than the superlight client. One of the major drawbacks is that the SNARK proof is very difficult/ compute-intensive to compute. This problem can be solved partially using progressive SNARK. Instead of generating proofs from the start for every extra period, the existing proof is used to generate a new proof quickly. Building SNARK implementation requires complex machinery. Circom is one of the most used libraries to build ZK SNARK circuits. There have been ongoing efforts to build a Circom circuit to verify BLS aggregate signature. Once we have a circuit to verify BLS aggregate, it might be worth constructing a client based on SNARKs.

5.2 Future Work

We plan to build to implement a fully functional production-grade superlight client for PoS Ethereum. This might be approached in several ways, like forking existing popular open-source Ethereum wallets or forking the lodestar light client implementation, or creating a daemon service that runs locally and exposes the standard RPC, which any wallet has can use. This protocol can also be added to the sync protocol specification in the Ethereum Consensus repository. Also, after the library for BLS aggregate library for Circom is ready, we would like to try the SNARK approach to build an ultralight client.

References

[1] Guido Bertoni et al. “Keccak”. In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 313–314.

[2] Dan Boneh, Manu Drijvers, and Gregory Neven. “Compact Multi-signatures for Smaller Blockchains”. In: *Advances in Cryptology – ASIACRYPT 2018*. Ed. by Thomas Peyrin and Steven Galbraith. Cham: Springer International Publishing, 2018, pp. 435–464. ISBN: 978-3-030-03329-3.

[3] Dan Boneh et al. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. In: *Advances in Cryptology — EUROCRYPT 2003*. Ed. by Eli Biham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 416–432. ISBN: 978-3-540-39200-2.

[4] Joseph Bonneau et al. “Coda: Decentralized Cryptocurrency at Scale”. In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 352.

[5] Joseph Bonneau et al. “Mina : Decentralized Cryptocurrency at Scale”. In: 2021.

[6] Benedikt Bünz et al. “FlyClient: Super-Light Clients for Cryptocurrencies”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 928–946. DOI: 10.1109/SP40000.2020.00049.

[7] Vitalik Buterin. “Ethereum: A Next Generation Smart Contract & Decentralized Application Platform”. In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Vitalik Buterin and Virgil Griffith. “Casper the Friendly Finality Gadget”. In: *CoRR* abs/1710.09437 (2017). arXiv: 1710.09437. URL: http://arxiv.org/abs/1710.09437.

[9] Vitalik Buterin et al. “Combining GHOST and Casper”. In: *CoRR* abs/2003.03052 (2020). arXiv: 2003.03052. URL: https://arxiv.org/abs/2003.03052.

[10] Ran Canetti, Ben Riva, and Guy Rothblum. “Refereed Delegation of Computation”. In: *Information and Computation* 226 (Mar. 2011). DOI: 10.1016/j.ic.2013.03.003.

[11] Stelios Daveas et al. *A Gas-Efficient Superlight Bitcoin Client in Solidity*. Cryptology ePrint Archive, Report 2020/927. https://ia.cr/2020/927. 2020.

[12] Stelios Daveas et al. “A Gas-Efficient Superlight Bitcoin Client in Solidity”. In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 132–144. ISBN: 9781450381390. DOI: 10.1145/3419614.3423255. URL: https://doi.org/10.1145/3419614.3423255.

[13] Ethereum Developers. “HOW TO RUN A LIGHT NODE WITH GETH”. In: *Ethereum Organization* (2022). URL: https://ethereum.org/en/developers/tutorials/run-light-node-geth.

[14] Ethereum Devs and Vitalik Buterin. “Ethereum Proof-of-Stake Consensus Specifications”. In: *Github* (2022). URL: https://github.com/ethereum/consensus-specs.

[15] Peter Gaži, Aggelos Kiayias, and Dionysis Zindros. “Proof-of-Stake Sidechains”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 139–156. DOI: 10.1109/SP.2019.00040.

[16] Developers Grin. “Merkle Mountain Ranges (MMR)”. In: (2020). URL: https://docs.grin.mw/wiki/chain-state/merkle-mountain-range.

[17] Jon Huang, Claire O’Neill, and Hiroko Tabuchi. “Bitcoin Uses More Electricity Than Many Countries. How Is That Possible?” In: *NewYork Times* (2014). URL: https://www.nytimes.com/interactive/2021/09/03/climate/bitcoin-carbon-footprint-electricity.html.

[18] Don Johnson, Alfred Menezes, and Scott Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA)”. In: *International Journal of Information Security* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[19] Harry A. Kalodner et al. “Arbitrum: Scalable, private smart contracts”. In: *USENIX Security Symposium*. 2018.

[20] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. “Proofs of Proofs of Work with Sublinear Complexity”. In: *Financial Cryptography and Data Security*. Ed. by Jeremy Clark et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 61–78. ISBN: 978-3-662-53357-4.

[21] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. “Non-interactive Proofs of Proof-of-Work”. In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Cham: Springer International Publishing, 2020, pp. 505–522. ISBN: 978-3-030-51280-4.

[22] Aggelos Kiayias and Dionysis Zindros. “Proof-of-Work Sidechains”. In: *Financial Cryptography and Data Security*. Ed. by Andrea Bracciali et al. Cham: Springer International Publishing, 2020, pp. 21–34. ISBN: 978-3-030-43725-1.

[23] Aggelos Kiayias et al. “Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol”. In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 357–388. ISBN: 978-3-319-63688-7.

[24] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. June 2013. DOI: 10.17487/RFC6962. URL: https://www.rfc-editor.org/info/rfc6962.

[25] Moxie Marlinspike. “My first impressions of web3”. In: (2022). URL: https://moxie.org/2022/01/07/web3-first-impressions.html.

[26] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[27] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Cryptography Mailing list at https://metzdowd.com* (Mar. 2009).

[28] Joachim Neu, Ertem Tas, and David Tse. *Two Attacks On Proof-of-Stake GHOST/Ethereum*. Mar. 2022.

[29] Caspar Schwarz-Schilling et al. “Three Attacks on Proof-of-Stake Ethereum”. In: *CoRR* abs/2110.10086 (2021). arXiv: 2110.10086. URL: https://arxiv.org/abs/2110.10086.

[30] Hursit Tarcan. “NODES AND CLIENTS”. In: *Ethereum Organization* (2022). URL: https://ethereum.org/en/developers/docs/nodes-and-clients.

[31] Ertem Nusret Tas et al. “Light Clients for Lazy Blockchains”. In: *Preprint* (2022). URL: https://arxiv.org/abs/2203.15968.

[32] Lodestar Team. “Lodestar Light-client”. In: *Github* (2022). URL: https://github.com/ChainSafe/lodestar/tree/master/packages/light-client.

[33] Nimbus Team. “Introducing Fluffy - an ultra-light client for Ethereum”. In: (2021). URL: https://our.status.im/nimbus-fluffly.

[34] Peter Todd. “Merkle Mountain Ranges”. In: (2012). URL: https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

[35] Psi Vesely et al. “Plumo: An Ultralight Blockchain Client”. In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 1361.

[36] Gavin Wood. “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER”. In: (2014). URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[37] Dionysis Zindros. “Proofs of Proof-of-Stake with Sublinear Complexity”. In: *Unpublished Manuscript* (2022).