Journal Pre-proof

Efficient Query Verification for Blockchain Superlight Clients Using SNARKs

Stefano De Angelis, Ivan Visconti, Andrea Vitaletti and Marco Zecchini

PII: S2096-7209(25)00123-X
DOI: https://doi.org/10.1016/j.bcra.2025.100396
Reference: BCRA 100396

To appear in: *Blockchain: Research and Applications*

Received date: 28 February 2025
Revised date: 8 August 2025
Accepted date: 15 September 2025

Please cite this article as: S. De Angelis, I. Visconti, A. Vitaletti et al., Efficient Query Verification for Blockchain Superlight Clients Using SNARKs, *Blockchain: Research and Applications*, 100396, doi: https://doi.org/10.1016/j.bcra.2025.100396.

Efficient Query Verification for Blockchain Superlight Clients Using SNARKs

Stefano De Angelis[b,c], Ivan Visconti[a], Andrea Vitaletti[a], Marco Zecchini[a,*]

[a] *Sapienza University of Rome, Via Ariosto, 25, Rome, 00185, Italy*
[b] *University of Salerno, Via Giovanni Paolo II, 132, Fisciano, 84084, Italy*
[c] *Nethermind Research*

Abstract

As blockchain-based decentralized applications continue to gain adoption, an increasing number of users wish to interact with them, inspect their state, and extract meaningful insights from on-chain activity. However, due to the decentralized nature of blockchains, safely accessing such information typically requires running a full node, maintaining a local copy of the ledger. This can be prohibitive for resource-constrained users, as the ledger size grows continuously.

In practice, most users rely on lightweight clients that query blockchain data through remote servers, without verifying the integrity of the responses. In many real-world scenarios, application-specific queries can only be answered by a small number of servers, sometimes all controlled by a single entity, thus reintroducing a single point of failure.

In this work, we present an architecture that enables superlight clients (i.e., clients unwilling or unable to download full transaction data) to outsource query execution to untrusted servers while still receiving trustworthy results. Our design leverages the power of SNARKs to ensure verifiable computation, made practical through data accessed from full nodes and blockchain explorers, and optionally supported by smart contracts.

We validate the feasibility of our approach through experimental evaluation on concrete use cases. Our results pave the way toward truly decentralized and reliable blockchain-based information systems, accessible even from constrained devices such as smartphones

Keywords: Superlight Client, Blockchain Scalability, SNARK

1. Introduction

Most of the emphasis in blockchain research has focused on designing and studying protocols capable of efficiently allowing a set of distrusting peers to achieve consensus on the

*Corresponding author.
Email addresses: sdeangelis@unisa.it, stefano@nethermind.io (Stefano De Angelis), visconti@diag.uniroma1.it (Ivan Visconti), vitaletti@diag.uniroma1.it (Andrea Vitaletti), zecchini@diag.uniroma1.it (Marco Zecchini)

status of the system, balancing efficiency (in particular, scalability) and security (in particular, increasing the number of participants in the consensus protocol) without relying on a third trusted party (TTP). The goal of the consensus protocol is to assemble transactions into blocks and order such blocks on the blockchain (ensuring a total order of transactions). Each transaction updates the state of the system according to some rules that can be either known a priori to all the participants (e.g., denying a user the ability to double-spend some cryptocurrency) or can be specified by users deploying smart contracts that can also be seen as programs running on the blockchain. However, not all peers have enough resources to store the blockchain and, therefore, locally access/read the ledger.

The RPC model. Originally, Remote Procedure Calls (RPCs) were designed to facilitate the access of off-chain applications developed locally on nodes participating in the consensus (hence, nodes that have autonomously checked the consistency and integrity of the blockchain). Instead, nowadays it is very common in many online application contexts to just rely on information received via RPCs without actually resorting to the chain of blocks to check the received answers. For example, Metamask[1], which is a famous application that allows users to ease the management of their cryptocurrency and the interaction with the blockchain, makes use of JSON-RPC to connect to a single full node and access the blockchain. In general, this design has opened access to many developers, but it has also introduced again a centralization issue: developers have to trust the reply of the RPC. Recently, a cryptocurrency scam has been discovered that operates by tampering with Ethereum node RPC to defraud unsuspecting victims[2]. In the above-described common scenarios, users owning constrained devices do not have many more guarantees than in standard client/server applications.

Application-specific queries. Such centralization issues related to the RPC architecture could be obviously addressed by having constrained clients repeat the same request to multiple independent full nodes and compare their responses to spot possible misbehaviours.

Of course, since these limited users are interrogating independent servers, this approach can work only on very general-purpose queries (e.g., state of a smart contract, existence of a transaction) that are standardized among different RPC servers. In some cases, servers can go beyond classic full node behavior and setup their own web services providing aggregated data which is of general interest for blockchain users (e.g., the number of transactions sent or received by an account or a smart contract), as in the case of blockchain explorers such as Etherscan or Blockchain.com. These web services, as well as RPC servers such as Infura, have created a business activity on this information and, thus, they have all the interests in answering honestly to such queries, otherwise they risk losing customers.

Nevertheless, there is a large number of blockchain applications that might have their own specific interest in processing data, either because they want to retrieve filtered transactions with ad-hoc filters (e.g., a user asks for all Bitcoin transactions with transferred value >1

[1] `https://metamask.io`

[2] `https://cryptopotato.com/slowmist-exposes-scam-using-malicious-rpc-node-modifications/`

BTC), or because they want the result of some ad-hoc aggregation (i.e., the average tip left by a transaction in 2024). A full node, with suitable incentives, could analyze the content of the transactions of the blockchain to answer such arbitrary queries. However, it is completely unclear how application developers requiring such sophisticated features can motivate a sufficiently large number of servers to implement the desired additional logic.

For these reasons, in the wild[3],there are natural useful queries that are neither handled through standard RPCs nor handled by well-known general-purpose services like blockchain explorers. We refer to this type of queries as *application-specific queries*.

Application-specific queries include requests to which a full node cannot reply without implementing new functionalities. Individual servers or small groups (that are usually the organization in charge of a specific blockchain application) can then setup their own servers, that could be also full nodes, but more importantly are designed to answer a specific set of application-specific queries that are of interest for the application. However, since such servers are limited in number and could all belong to the same organization, this again introduces an even more urgent centralization problem for those users who are not able to setup their own full nodes.

The light client model. To reduce dependence on full nodes, developers can deploy light clients. Designed by Nakamoto's seminal paper [1], a light client stores locally only a portion of the blocks of the blockchain (i.e., the headers) that is sufficient to guarantee the integrity of the rest of the block and to verify the correctness of the consensus protocol. In particular, in the headers, there is the root hash of a Merkle Tree built using as leaves the transactions of the block. In this way, a light-node client can ask a full-node one or more transactions (which are included in the part of the block that a light-node does not store) along with the Merkle proof used to attest that such transactions are actually in the ledger. Hence, light clients drastically reduce the amount of memory required for developers to verify the integrity of blockchain transactions or smart contract state. However, despite avoiding downloading and storing the entire blockchain, there are several important shortcomings.

First of all, the amount of memory required by light clients still grows linearly with the size of the blockchain. Hence, eventually, the size of the chain of headers exceeds the memory available on constrained devices (e.g., smartphones). To work around this problem, Alchemy, a relevant company building blockchain software products, suggests pruning the chains of headers requiring only 400 MB.[4] However, if a user wants to read some information that is stored in the pruned part of the ledger, she has to ask for such information from a full node without being able to verify it.

Next, in case a light client is always online updating the chain of headers, there is a significant consumption of resources (e.g., the battery of a smartphone). If, instead, is only sometimes online, there are significant resources to use to get and check the missing blocks.

Last, but not least, storing headers does not allow one to quickly verify the answer to a

[3]See for example, the queries elaborated through Dune Analytics available at: `https://dune.com/queries/4069681` and `https://dune.com/hildobby/btc-etfs`

[4]`https://www.alchemy.com/overviews/light-node`

application-specific query that affects a large number of transactions, since the light client should first download all the relevant transactions, then verify them, and finally run the query over them. Again, this can be excessive for a resource-constrained device (e.g., a smartphone with limited storage and battery charge).

There is a large scientific literature studying techniques to make Nakamoto light clients more efficient (see Sec. 2). However, as also analyzed in [2], this above prior works faces the following question (see Section 3.1 [2]): *is a specific transaction or account state included in the blockchain?* Again, this does not allow to answer efficiently application-specific query depending on many transactions, thus limiting the adoption of blockchain applications in resource-constrained devices.

Smart contract to store application-specific query answers. Smart contracts allow application-oriented storage on the blockchain, in addition to raw transactions. Full nodes and blockchain explorers allow fast access to such information. Therefore, one might think of keeping updated on the smart contract's state and the output of application-specific queries. In this way, users running a light client simply need to access and verify one single state of the smart contract.

However, such an approach, while theoretically feasible, suffers from very severe practical limitations. First of all, storing and keeping updated a large amount of data (e.g., the results of a huge amount of interesting queries) on a smart contract impacts the cost that users have to support in order to interact with a blockchain application. Indeed, transaction fees are usually computed in relation to the effort (both in terms of storage and computational complexity) that validators have to do in order to execute such transactions. Second, it is reasonable to assume that, in some application contexts, interesting readings of the blockchain for users might arise dynamically while they are using the application (new features are added to web services very often). Unfortunately, many blockchain technologies (e.g., Ethereum Virtual Machine-based blockchains) do not natively allow one to modify the code of a smart contract once it is deployed, making it hard to store the answers to the new application-specific queries. Finally, some blockchain technologies, such as Bitcoin, do not have smart contracts and thus there is no help from the above techniques.

A server answering application-specific queries with cryptographic proofs. Merkle proofs can be used to attest the inclusion of single transactions but they become inefficient for queries involving many transactions due to high communication costs. To address this, [3] proposes to use succinct cryptographic proofs that allow light clients to verify that a block belongs to the ledger given a trusted block using space and time sublinear to the longest chain (for more details, see Sec.2). In theory, this approach can be extended to certify that (a) the longest chain is valid, (b) all relevant transactions for a application-specific query are included, and (c) the query result is correct. However, generating such proofs over large blockchain segments is computationally intensive, with estimated annual costs in the tens of millions of dollars [4], making this approach impractical for general use.

1.1. Motivation

Accessing trustworthy on-chain data remains a fundamental challenge for resource-constrained clients, which we refer to as *superlight clients* throughout this work. The majority of blockchain research has focused on consensus protocols that enable decentralized agreement on the state of the system, ensuring security and scalability without relying on a third trusted party. However, this focus often overlooks the limitations of end-user access to blockchain data, especially for devices that cannot store or process the full ledger.

In practice, many applications rely on RPCs to retrieve blockchain data from centralized servers, introducing new trust dependencies. While general-purpose queries are typically supported by services like Infura or Etherscan, developers often need application-specific queries, namely custom data retrieval or aggregation logic that is not handled by standard full nodes or blockchain explorers. Supporting such queries requires additional server-side functionality, often reintroducing centralization and trust concerns.

Light clients offer a partial solution by allowing users to verify the inclusion of transactions through suitable proofs (e.g. Merkle proofs), but they remain limited in functionality and scalability. They cannot efficiently handle application-specific queries involving multiple transactions, and their resource requirements grow linearly with blockchain size. Alternative approaches like providing answers through smart contracts are impractical due to storage costs and the lack of smart contract support in some blockchains like Bitcoin.

Figure 1 summarizes existing approaches for computing application-specific queries and highlights their infeasibility on smartphones and similar devices. A more practical approach would involve relying on powerful servers to execute queries and return results. However, this creates a tension with the decentralization ethos of blockchain: trusting a single server undermines the very principles that blockchains were designed to uphold.

One way to reduce this trust assumption is to query multiple independent servers and rely on a majority consensus, under the assumption that reputation-sensitive servers will provide trustworthy answers. Yet, this assumes a mature ecosystem, where multiple independent servers are already interested in supporting the application. In reality, most services undergo a bootstrap phase, where only a few servers, often run by the same organization, are available. During this early phase, smartphones have little choice but to trust a single, potentially unvetted server.

These constraints highlight a critical gap in current blockchain infrastructure: there is no scalable, verifiable, and resource-efficient solution for enabling application-specific queries on constrained devices. Addressing this gap is essential for expanding the usability and trustworthiness of decentralized applications, especially on mobile platforms.

1.2. Research Problem

In light of the above discussion, blockchains currently fail to realize a general-purpose decentralized platform to answer application-specific queries made by superlight clients (e.g., smartphones) that today represent the most widespread devices among the population. The above state of affairs motivates the natural open question of designing a decentralized architecture that allows superlight clients to efficiently and reliably obtain answers to application-specific queries without relying on individual TTPs. However, it can still be acceptable to

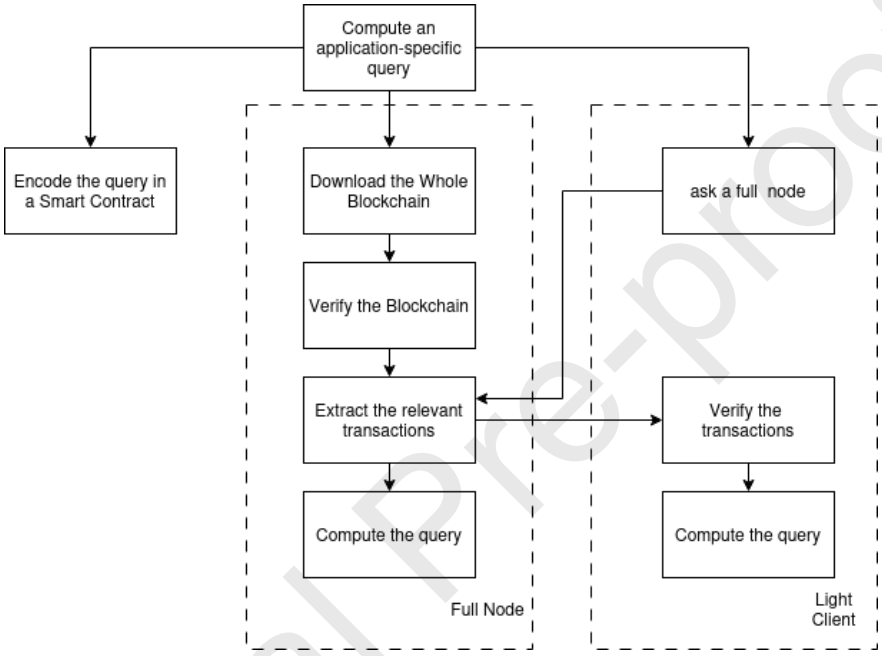


Figure 1: Current approaches to compute application-specific queries might be unfeasible or unpractical on resource constrained devices, such as smart phones. Implementing complex queries on smart contracts can be unfeasible or result in prohibitive costs. Smartphones lack the storage capacity to download the entire blockchain, and when a query demands significant bandwidth or memory, even light clients may become impractical. There is the need of a solution that allows superlight clients to efficiently and reliably obtain answers to application-specific queries without relying on individual TTPs.

exploit independent full nodes or any other standard components of the blockchain architecture that are currently queried by a multitude of applications using standard RPC calls.

1.3. Our contribution and technical overview

In this work, we propose an architecture where superlight clients do not trust the server's answer as in a leap of faith, but instead get convinced by proofs generated by the server.

More in detail, our approach is based on the following main idea. The light client can verify the answer provided by the server through various verification steps that include the verification of succinct proofs, called SNARKs, that are very fast to verify. While it is well known that the computation of SNARKs might be unfeasible on large datasets given in input to cryptographic hash functions, we present an effective solution allowing us to handle map-reduce application-specific queries, employing a divide-et-impera approach capable of dealing with large sets of transactions that would be untractable in a single SNARK. In particular, in our architecture, the server computes a SNARK only for the blocks actually containing relevant transactions. However, there is a subtle issue. A malicious server might cheat simply by omitting some important blocks, therefore, discarding some valuable transactions from the computation of the output of the query. To guarantee that all necessary transactions have been considered, the SNARK is given by the server along with a link to some publicly available information, obtainable by standard components of the blockchain architecture, so that by inspection the superlight client is guaranteed that the only way to answer a query according to the publicly available information is to consider all relevant transactions. The above publicly available information is crucial for our architecture and is clearly application-specific. Therefore, we will focus on those scenarios where either the state of smart contracts, or RPCs, or blockchain explorers provide this public value. Note that since the nature of the above publicly available information makes it accessible through multiple nodes and therefore the superlight client can contact some of them and proceed in the computation if and only if all the replies are consistent.

To further improve the efficiency of our solution, the per-block SNARKs are eventually merged into a single SNARK that is verified by the superlight client.

We stress that, while the map-reduce approach allows us to efficiently deal with application-specific queries, it is not efficiently applicable to all possible kinds of queries. This happens in particular when the computation of the query can not be split in subcomputations affecting only a subset of the relevant transactions. While this means that our architecture does not provide a general-purpose solution, we make significant progress enabling several natural and significant use cases.

We first consider a use case where the application-specific query is the computation of the average BTC transferred by a Bitcoin account, specifically the Satoshi Nakamoto's account on Bitcoin that, at the time of writing, is involved in around 40k transactions. The experimental results show that our approach effectively verifies and computes the query using approximately 15 MB. Answering the same query with a traditional Nakamoto light client requires around 700 MB, while computing it with a light client that uses a sublinear amount of space in storage to keep track of the blocks (which is the goal of the mentioned track of research on light clients) requires around 200 MB.

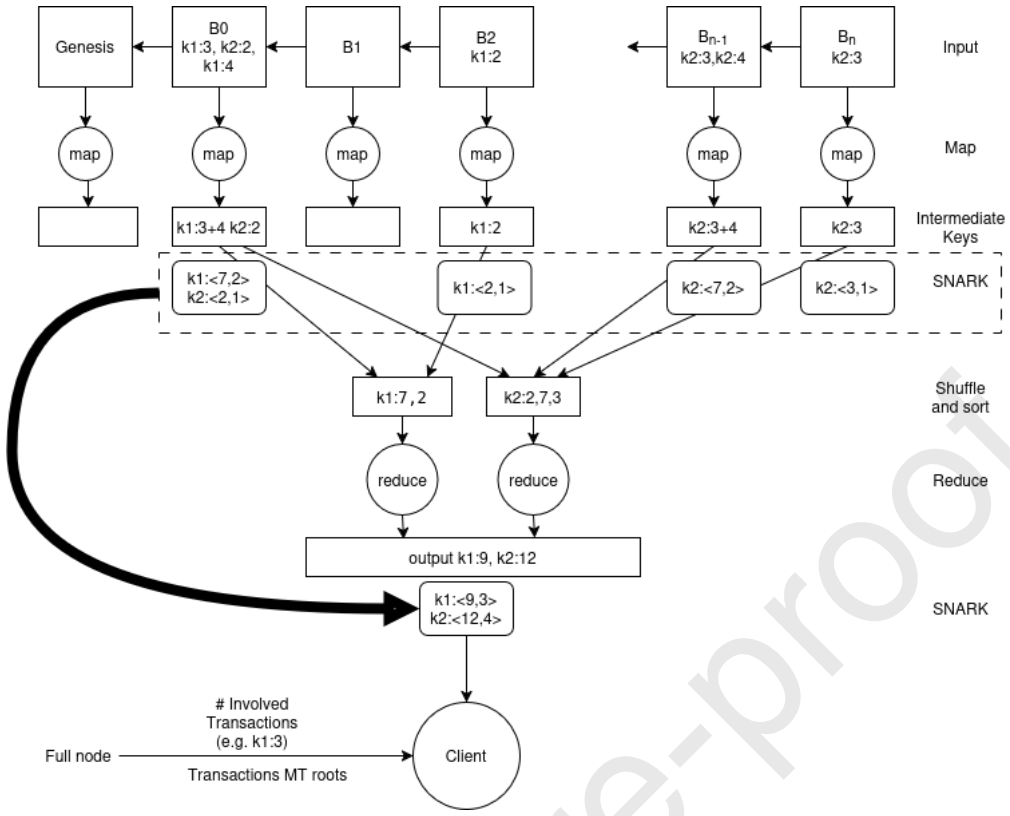


Figure 2: In this example we are interested in computing the average value of all the transactions generated by users k1 and k2. For the sake of simplicity, we consider a mapper for each block. The SNARKs are used to prove the result of the computation. To prove all relevant transactions have been considered, the SNARK output is linked to a cumulative value available in a full node; in the example the toal number of involved transactions.

The second use case involves an on-chain election handled with a smart contract (see Appendix A). The application-specific query is about the average number of votes for a specific candidate in a given time-interval. Our experiments, considering an election period spanning 7000 blocks (around one day) and 70000 votes, confirm that the approach effectively verifies and computes the query using approximately 9 MB of data. Answering the same query with a traditional Nakamoto light client requires around 1.1 GB, while computing it with a light client that uses a sublinear amount of space in storage to keep track of the blocks also requires around 1 GB.

A concrete example. Introduced in 2008 by [5], Map-Reduce is a programming model for processing large datasets, in our case, the set of transactions in the blockchain. Figure 2 summarizes the main tasks of the model: 1) the Map task takes a set of data as input, in our case blocks and transactions, and converts it into a new set of data, where individual elements are broken down into intermediate key-value pairs; 2) the Reduce task takes the output from the Map as input and combines those intermediate key-value pairs into a smaller set of tuples used to eventually compute the output.

2. Related Works

The design of light clients has been a central topic in blockchain research, with the goal of allowing resource-constrained devices to interact with a blockchain network without needing to store or process the full state. A comprehensive systematization of this area is provided in the work by Chatzigiannis et al., [2], which surveys the range of models, security guarantees, and trade-offs involved in building light clients for different blockchain architectures.

In the context of Proof-of-Work (PoW) blockchains such as Bitcoin, the classic model for light clients is Simplified Payment Verification (SPV) [1]. In this model, a client follows the longest chain by downloading and verifying block headers, ensuring that a transaction is included in a block with sufficient proof-of-work. However, the main limitation of this approach is its linear communication complexity: a light client must download a number of block headers proportional to the length of the chain. This limits its scalability and efficiency on devices with constrained storage or intermittent connectivity.

To overcome this limitation, a line of work has focused on constructing proofs of chain validity with sublinear complexity. Kiayias et al., introduced Proofs of Proof-of-Work (PoPoW) [6], where they construct succinct proofs based on superblocks that carry stronger proof-of-work than normal blocks and allow a verifier to be convinced of the existence of a valid, high-work chain without downloading all headers. As an extension of this work, the authors presented Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [7], presenting a framework for compressing the chain into a small number of block headers selected probabilistically. Subsequently, FlyClient [8] refined this direction by eliminating the need for superblocks and instead using probabilistic sampling of block headers. In FlyClient, a verifier samples a small subset of block headers from the chain and verifies their correctness using Merkle proofs, achieving logarithmic communication overhead in the number of blocks. PoPoW, NIPoPoWs, and FlyClient represent a major step forward in making light clients more scalable, but they are still tailored to following the chain's state over time, and differently from our work, they do not directly address the need to verify application-specific queries, requiring light clients to still download a significant amount of unnecessary information.

A separate direction has emerged based on using succinct proof systems, particularly SNARKs, to prove statements about blockchain execution. In this line of work, Kattis et al., [9] proposed a system where a prover generates a SNARK that attests to the existence of a valid PoW chain and the correct execution of state transitions. This model shifts the burden of verification to a single succinct proof, removing the need for block-by-block validation on the client side. However, the proof still requires encoding the entire consensus logic and execution model of the blockchain, which can be computationally heavy for the prover and lacks flexibility for more granular queries.

Building on this idea, in [10] the authors formalize a light client protocol as a classical two-party proof system between a prover and a verifier. In their formulation, the prover (e.g., a full node) commits to a blockchain and proves knowledge of its validity through a SNARK. An important insight is that most prior light client protocols implicitly operate in this two-party model, yet rely on the assumption of multiple honest provers. By contrast, [10] make this structure explicit and explore the construction of incremental SNARKs over

sequential PoW blocks. While our work shares this two-party framing, it introduces a hybrid approach in which the verifier does not fully trust the prover. Instead, the verifier uses a SNARK to attest to the correctness of a computation (e.g., the result of a query), and then contacts a set of public data providers—such as blockchain explorers or RPC nodes—to retrieve the minimal public information required to validate the proof. This design preserves the succinctness benefits of SNARKs while improving robustness through decentralization of the data sources used for verification.

For Proof-of-Stake (PoS) systems, where consensus is achieved through validator signatures rather than proof-of-work, the problem of succinct verification takes a different form. Naively, a light client verifying a PoS chain would need to download all validator signatures for each block, which is infeasible in practice. To address this, Plumo [11] presents a SNARK-based protocol to compress the execution of the PoS consensus over many subsequent blocks. The SNARK proves that each block was signed according to the protocol rules, and that validator set updates were performed correctly. This allows the light client to verify the validity of the chain by checking a single proof, making it extremely efficient and suitable for mobile deployment. However, Plumo assumes a trusted checkpoint of the validator set, and its proof model is tightly coupled to the underlying consensus protocol.

Similarly, zkBridge [3] uses SNARKs to prove that a block from a source chain satisfies its consensus rules and can therefore be trusted by a destination chain. This system is designed for cross-chain interoperability and targets smart contracts as verifiers. Like Plumo, zkBridge assumes a trusted setup or checkpoint, and focuses on verifying the validity of headers, rather than enabling application-specific queries over the state of the chain. In both cases, SNARKs are used to attest to the consensus correctness of the source chain.

Our work differs fundamentally in focus and trust model. Rather than proving the validity of consensus, we focus on proving that the result of an application-specific query over the blockchain state is correct. The prover may be untrusted, but the SNARK proof can be verified using publicly available data, retrieved independently by the client from multiple full nodes or explorers. This allows our architecture to support a broader class of queries, while remaining chain-agnostic and significantly reducing prover complexity compared to consensus-aware SNARKs.

Another important direction involves protocols that combine interactive sampling and economic incentives [12, 13, 4]. These works explore game-theoretic and interactive models in which a superlight client engages in a dialogue with a set of full nodes. These protocols are often probabilistic and rely on multiple rounds of interaction, with security guarantees holding under assumptions of partial honesty among the participants. While these designs reduce trust in any single node, they typically require synchronous communication and multiple rounds of message exchange, which can limit their practicality.

In contrast, our approach uses a SNARK-based model, where the light client receives a single proof from the prover and fetches a small amount of public data to validate it from external parties. This design combines the succinctness and modularity of SNARK-based protocols with the resilience and decentralization of multi-source data retrieval. To our knowledge, it is the first light client design that explicitly decouples consensus verification

from query verification in a SNARK-based setting, enabling flexible, application-oriented use cases without relying on trusted validators or checkpoints.

Wang et al. [14] propose PARP, a Permissionless Accountable RPC Protocol that allows light clients to query full nodes through payment channels, where each response is accompanied by a Merkle proof to ensure data integrity. The full nodes are collateral that can be slashed upon submission of invalid data, proven via an on-chain fraud proof mechanism. While effective for incentivized RPC provisioning, PARP is limited to relatively simple blockchain queries whose correctness can be attested via Merkle inclusion proofs and cannot handle the richer application-level queries supported by our system.

3. Model

3.1. Preliminaries

Notation. For an array (or list) X of size n, where $n \in \mathbb{N}$, we denote with $\{x_0, x_1, \ldots, x_n\}$ the expanded list of its elements and with $|X|$ the size of the array.

For probabilistic polynomial-time algorithms A and B, we denote by $\mathsf{P}\langle \mathsf{A}(a), \mathsf{B}(b)\rangle \to x$ the random process P obtained by having A and B interact on (private) auxiliary inputs a and b, respectively, and with independent random coin tosses for A and B. The value x represents the output of A after the interaction. We extend this notation denoting with $\mathsf{P}\langle \mathsf{A}(a), \mathsf{B}_{1,\ldots,n}(b)\rangle \to x$, the random process P obtained by having A and n executions of B interact on (private) auxiliary inputs a and b respectively, and with independent random coin tosses for A and every B_i. The value x represents the output of A after the interaction.

Merkle Tree. A Merkle Tree computed over input values $x_1, \cdots, x_n$ is a binary tree in which the input values are placed at the leaves all with the same largest depth, and the value at each internal node is the collision-resistant hash of the values of its two children (or just the hash of the left child if the right child is missing). If n is not a power of 2, the right part of the tree will have missing nodes. The height of the tree is logarithmic in the number of leaves. The root of the Merkle Tree is a succinct representation of the entire sequence $x_1, \ldots, x_n$. We denote the root of a Merkle tree as $\mathbb{R}$.

Blockchain. We consider a blockchain network where validator nodes collectively maintain and update a valid chain B. A chain is *valid* whether the honest majority of blockchain validators agreed on its content according to the network consensus rules. We also refer to a valid chain as *canonical*. The chain maintains a list of transactions T executed by the users uniquely identified with an account address, denoted with acc.

A chain B is an array of blocks, where the block b_0 is the genesis block. Every block includes a list of transactions. We denote the list of transactions in a chain as T, and with T_{b_i} the transactions of block b_i. A transaction tx executes chain-specific tasks, e.g., transferring an amount of tokens from one account to another or invoking the execution of a smart contract.

Every new block added to the chain updates the state of the ledger from st to st'. Note that, for simplicity, the state of B reflects the view of B for the users, including not only

the states of each account or smart contract that has sent or received a transaction but also other information available on the ledger, such as the block hash, the difficulty of a round of consensus, and the block number.

The list of the transactions and the list of the states of each account or smart contract that has sent or received a transaction are leaves of a Merkle Tree. The block header maintains the transactions root. For a block b_i we denote with $\mathbb{R}_T^{b_i}$ the transactions root. We denote the inclusion of a block b_i in a blockchain B with $b_i \in B$. A transaction tx is said to be part of the canonical chain in the block b_i, if tx is a leaf of the Merkle Tree with root $\mathbb{R}_T^{b_i}$ and $b_i \in B$, and we denote it with $\mathsf{tx} \in B$.

SNARKs. A proof system involves two parties: a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. The prover processes a public claim x along with a private witness w satisfying a polynomial relation $\mathcal{R}$ between x and w (i.e., $\mathcal{R}(x, w) = 1$) and generates a proof π. The verifier then checks π against x, and outputs 1 if the proof is valid or 0 otherwise.

A *Succinct Non-Interactive Argument of Knowledge* (SNARK) is a specific type of proof system providing these key properties:

- *Non-interactivity:* $\mathcal{P}$ sends only one message containing π to $\mathcal{V}$ with no further communication required.
- *Completeness:* if $\mathcal{R}(x, w) = 1$ and $\mathcal{P}$ honestly generates π, then $\mathcal{V}$ verifying π outputs 1 with overwhelming probability.
- *Knowledge Soundness:* if $\mathcal{R}(x, w) = 1$ and $\mathcal{V}$ verifying π outputs 1, then $\mathcal{P}$ knows w for x with overwhelming probability.
- *Succinctness:* The size of the proof π is sublinear in the size of w and to verify it requires time complexity that is linear to size of x plus a value that is sublinear in the size of w.

Prover and verifier run their respective algorithms having as input a set of parameters, used as a common reference string. Depending on the different SNARK constructions, such parameters can be generated autonomously by the parties or must be generated by a third party that generates the common reference string with some secret that, then, is trusted to destroy.

A *recursive* SNARK system leverages these properties so that one proof can verify the correctness of other proofs within its own proving circuit. In this paradigm, a single proof π attests that several proofs $\pi_1, \ldots, \pi_n$, each with a corresponding public claim $x_1, \ldots, x_n$, have been validated correctly. When the outer proof π is verified, it implies that all proofs $\pi_1, \ldots, \pi_n$ are also valid.

3.2. Map-Reduce Blockchain Queries

Introduced in 2008 by [5], Map-Reduce is a programming model processing large datasets that can be used in a broad variety of real-world tasks.

In the context of blockchain-based queries, the dataset is the canonical chain B and each query accesses multiple blocks.

These queries are decomposed into smaller, independent sub-queries that perform a generic computation of a function over a sublist of transactions. For example, the average of tokens per transaction transferred from a blockchain account can be computed as a map-reduce query that splits the calculation across relevant blocks, i.e. all those blocks that include relevant token transfer transactions.

In our work, we formalize the definition of map-reduce blockchain query. A map-reduce query is an operation made of three steps. The first two steps follow the paradigm proposed by [5] (see Section 2):

- map: takes as input a key/value pair and outputs a set of intermediate key/value pairs. Its main goal is to split a large computation in smaller tasks (e.g., from the entire blockchain to one computation per block) and produce a set of intermediate key/value pairs for each task (e.g., one set per block) computed according to a function $f_{\mathsf{M}} : \{0,1\}^* \to \{0,1\}^*$.

- reduce: takes as input an intermediate key and a set of values for that key and outputs a smaller set of values obtained by merging and combining together the values of the input set computed according to a function $f_{\mathsf{R}} : \{0,1\}^* \to \{0,1\}^*$. Its main goal is to allow to compute a function over a set of values too large to fit in memory in a sequential way.

In the following, for the sake of simplicity, we define a map-reduce query only over the set of transactions in a blockchain B, however, the domain for this function can be easily extended also to other information of the ledger. Formally, a map-reduce blockchain query is a function $f_{\mathsf{MR}} : B \to \{0,1\}^*$ defined over the canonical chain B. The returned value of this function is a binary string $\{0,1\}^*$.

3.3. System Model and Properties

In this subsection, we introduce and formalize which parties are involved in the system and which status of the system they can reach (and cannot reach) by defining the properties of the system.

In the system, there are three types of users:

Full Node A full node is a blockchain node that maintains the full history of the chain B including the blocks, and their updated states. After a bootstrap period, every full node is able to remain synchronised with the longest valid chain. Therefore, the amount of data a full node downloads when joining the blockchain is linear in bandwidth and storage with the length of the chain $|B|$. Full nodes can autonomously query the chain, and execute transactions and smart contracts functions. These nodes compute a set of function queries $f_1, \ldots, f_m$ about the ledger and make the output available to superlight clients. Such functions can be computed either on raw data (e.g., retrieving a block header, the details of a transaction included in the chain) or on more sophisticated

but of general interest data as in the case of blockchain explorer (e.g., balance of an account, token transfer histories, account-level analytics). Full nodes expose either an RPC interface over a communication protocol like HTTPS or WebSocket or a user-friendly web sites (as in the case of blockchain explorer) to provide to the superlight clients the output of $f_1, \ldots, f_m$. Each of these functions takes as input B and outputs a state for the blockchain st.

Oracle Server An oracle server, referred to as an Oracle, extends the functionalities of a full node and exposes map-reduce queries that are not available in the other full nodes because these are application-specific and require the execution of intense computing tasks. These servers leverage high-performance infrastructures with minimal constraints on bandwidth, storage, or computation. Oracles maintain access to the entire history of the blockchain, either by querying full nodes or running their own instance.

Stateless Superlight Client (SSLC) A computationally constrained node (e.g., smartphone, laptop) that aims to track a specific state or answer a application-specific query over an application or account on the blockchain, without downloading the full chain history of size $|B|$. We define this type of client also as *stateless*, because it does not need to maintain the blockchain state locally once it has verified a application-specific query.

In our model, we consider queries that are map-reduce ones and that are application-specific which means that are not supported by general-purpose full nodes. In other words, the output of the query is not obtained by simply running $f_1, \ldots, f_m$ (which as already stated are the set of queries computed by a full node). While full nodes allow retrieving transactions, they are not supposed to compute their content, especially when the computation is not a general-purpose one. Typical examples of queries that are not supported by full nodes consist of filtering the transactions based on their content and, finally, computing a function using as input the content of filtered transactions. Formally, given a blockchain B, we consider a map-reduce query f_{MR} such that $f_{\mathsf{MR}} \neq f_i, \forall i \in \{f_1, \ldots, f_m\}$.

In the following, we present a model for our system that is inspired by [2]. Given a SSLC L, an oracle server S and a set of full nodes $\mathsf{FN}_1, \ldots, \mathsf{FN}_n$, our model presents the following protocols:

$\mathsf{Q}\langle \mathsf{L}(f_{\mathsf{MR}}), \mathsf{S}(B) \rangle \rightarrow (r, \pi)$ To compute the answer to an application-specific query, a SSLC L, taking as input a function f_{MR} such that $f_{\mathsf{MR}} \neq f_i, \forall i \in \{f_1, \ldots, f_m\}$, runs an interactive protocol Q together with an Oracle server S, taking as input a blockchain B, and that outputs a reply $r := f_{\mathsf{MR}}(B)$ and a proof π.

$\mathsf{Vrfy}\langle \mathsf{L}(f_{\mathsf{MR}}, r, \pi), \mathsf{FN}_{1,\ldots,n}(B) \rangle \rightarrow b$ To verify the correctness of an answer to an application-specific query, a SSLC L, taking as input a function f_{MR}, a reply to an application-specific query r and a proof π, runs an interactive protocol Vrfy together with full nodes $\mathsf{FN}_1, \ldots, \mathsf{FN}_n$, taking as input a blockchain B, and outputs $b \in \{0, 1\}$ such that if the proof is correct $b = 1$; if the proof is not correct, $b = 0$.

Our model makes use of the following assumption:

Existential Honesty There is at least one honest full node in the set of full nodes. Such assumption ensures that, given a blockchain B with state st, a SSLC contacting independent full nodes $\mathsf{FN}_1, \ldots, \mathsf{FN}_n$ during $\mathsf{Vrfy}()$ gets the same st; otherwise, if it only receives one st' such that $\mathsf{st}' \neq \mathsf{st}$, $\mathsf{Vrfy}()$ outputs 0 and L restarts it with a different set of full nodes.

Our model ensures the following properties:

Secure Querying Given a blockchain B with state st that can be obtained by L contacting $\mathsf{FN}_1, \ldots, \mathsf{FN}_n$, for any application-specific query function f_{MR} involving the transactions $\mathsf{tx}_1, \ldots, \mathsf{tx}_k$, it is computationally hard for a malicious oracle server to generate a proof π for a reply r' such that $\mathsf{Vrfy}\langle \mathsf{L}(f_{\mathsf{MR}}, r', \pi), \mathsf{FN}_{1,\ldots,n}(B)\rangle = 1 \wedge (\exists i \in \{1, \ldots, k\} \text{ s.t. } \mathsf{tx}_i \notin B \vee f_{\mathsf{MR}}(B) \neq r')$.

Efficiency We identify the following properties in terms of storage, computation, and communication costs.

- *Efficient storage:* The amount of memory needed by the SSLC to run $\mathsf{Q}()$ and $\mathsf{Vrfy}()$ is sublinear to $|T|$.
- *Efficient communication:* $\mathsf{Q}()$ and $\mathsf{Vrfy}()$ requires communication costs that is sublinear to $|T|$.
- *Efficient client computation:* $\mathsf{Q}()$ and $\mathsf{Vrfy}()$ requires computation costs for the SSLC that are sublinear to $|T|$.

Reputation of full nodes. We have adapted to our case the Existential Honesty assumption from [13]. Full nodes (such as Infura) or blockchain explorers (such as Etherscan) have founded business activities on their ability to maintain and answer queries to users with constrained devices and, thus, they have all the interests in answering honestly to such queries; otherwise, they risk losing customers (indeed, a user can interrogate different services comparing their answers and eventually detect bad behaviors). Hence, in our model, full nodes are interested in the reputation they have with the other users in the system. For this reason, it is unlikely that they answer dishonestly to queries made by a SSLC, therefore the Existential Honesty assumption described above is concretely realistic.

Differences and similarities with [2]. As previously mentioned, the model presented in this section is inspired by the definition of Section 3.1 of [2]. However, we deviate from their work to make the model more suitable for the context (i.e., we consider application-specific queries) that we are considering. In particular, our model enables us to answer queries related to many transactions and a function f_{MR} maintaining the same efficiency properties. Our system relies on the Existential Honesty assumption which is an extension of one mentioned in [2] called "Game-theoretic assumptions". Differently from them, our model gives more detail about this assumption regarding the behavior of full nodes and how this affects our

system. Given this assumption, the SSLCs in our model do not need to go through any bootstrap or sync phase (i.e., starting from a known genesis block, SSLCs update and verify the state st of B). Even though these phases enhance the security of the system, we believe that the assumptions of our model are reasonable in many real-world scenarios. However, our model can be easily modified to include a bootstrap or sync phase, as in [2].

4. Stateless Superlight Clients

In this section, we show how to realize a *Stateless Superlight Client (SSLC)*, a solution that enables light clients to execute application-specific queries on blockchain data with minimal trust, computation, and storage requirements. At their essence, an SSLC is a resource-constrained device running a blockchain-specific application that requires the execution of computationally expensive queries on a blockchain B. We restrict the scope to map-reduce blockchain queries f_{MR} for a canonical chain B. We idealize these functions to be available through an oracle that provides computational results on demand. This assumption allows us to analyze the security and correctness of the protocol without requiring the verifier to perform the full computation, relying instead on succinct SNARK proofs of correctness.

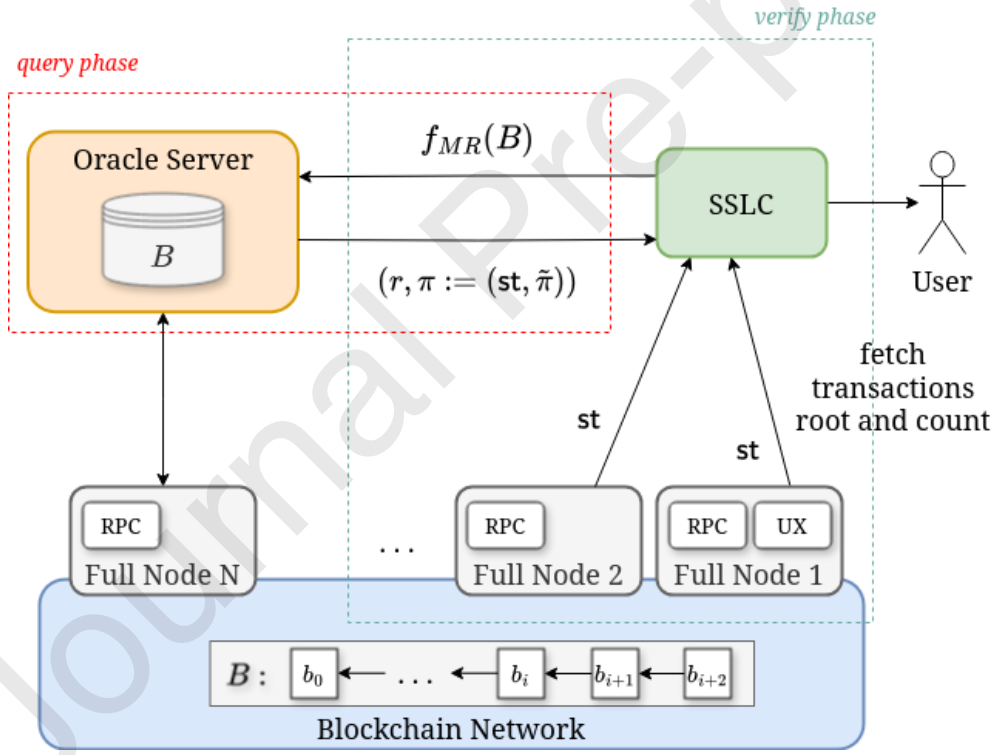


Figure 3: Stateless Superlight Client Architecture. Note that, in the figure, the SSLC connects Full Node 1 and Full Node 2 but it is free to choose any Full Node between 1 and N.

Figure 3 provides a graphical representation of the solution. The SSLC (verifier) engages in an interactive protocol with an application-oriented Oracle Server (prover) and general-purpose full nodes that maintain a valid blockchain B. The protocol consists of two phases:

query and *verify*. In the query phase, the verifier requests the execution of a map-reduce query to the prover. The prover runs the computation and returns the result r along with some information about the blockchain st and SNARK proof $\tilde{\pi}$ that attests to its correctness.

In the verify phase, the verifier connects to different general-purpose full nodes to collect relevant information about the ledger st to verify the proof, such as the root of transaction Merkle trees in each block involved in the computation and the total number of transactions allegedly used. As we will see, this information is crucial for the security of the protocol; without it, the client cannot ensure that the prover has correctly accounted for all valid transactions in the canonical chain that were involved in the computation.

It is worth noting that the number of downloads of transaction roots is sublinear in the number of transactions processed to compute the final result r, assuming that each computed block includes a large number of transactions involved in the computation. Therefore, with this approach, we ensure that the communication complexity remains proportional only to the number of connections to full nodes while the amount of downloaded data remains sublinear to the growing chain. We will see that for application-specific use cases, this approach is optimal, as it allows light clients to download a significantly smaller amount of data. In the next section, we present in detail the SSLC solution. We first proceed in Section 4.1 with the description of how to compute the map-reduce queries in our context. Then, in Section 4.2, we show how to compute the SNARK proof attesting to the correctness of the computation of the function f_{MR}. Finally, in Section 4.3, we provide an instantiation of the SSLC and a detailed description of the protocol. Note that the map-reduce computation paradigm and the computation of SNARK proofs on arbitrary circuits, presented respectively in Section 4.1 and 4.2, are not novel notions. However, no previous work (presented in Section 2) appears to have explored the use of these combined notions for the construction of a light client. Furthermore, these are preliminary notions needed to build a SSLC, which is the main contribution of the paper.

4.1. Map-Reduce Query Computation

The oracle server maintains a copy of the canonical chain B and provides an interface for the SSLC to execute map-reduce blockchain queries. The computation of a map-reduce query is application-specific and for a blockchain application corresponds to the implementation of the function $f_{\mathsf{MR}}(B)$. Obviously, for an application, there might be more than one map-reduce query available. The function selects and operates only on specific blockchain transactions, such as those sent or received by one or more blockchain accounts or one or more smart contracts.

We describe the computing steps of the map-reduce query in Figure 4. For the sake of simplicity, in Figure 4, we show how to compute a map-reduce query for transactions involving a single blockchain account (acc_1), but the described steps can be easily applied also to the case with more blockchain accounts or smart contracts.

Algorithm to compute f_{MR}

We assume we are interested in computing a function f_{MR} for the transactions involving a specific account acc_1:

Map Phase $\mathsf{map_i}$: it takes as input a block b_i as key and the set of transactions of that block T_{b_i} as value and

1. In T_{b_i}, identify the set of transactions $\{\mathsf{tx}_1, \ldots, \mathsf{tx}_{k'}\}$ sent or received by acc_1;
2. Compute a function f_{M} on this set of transactions and save its value in a variable v_i;
3. Output the pair (acc_1, v_i)

Reduce Phase reduce: it takes as input a blockchain account acc_1 as key and a set of values $\{v_1, \ldots, v_{|B|}\}$ computed during the map phase and

1. Combines the values $\{v_1, \ldots, v_{|B|}\}$ into a single result v by applying the function f_R.
2. Output the pair (acc_1, v).

Figure 4: Computing steps of a Map-Reduce query.

4.2. A SNARK Proof System for the Oracle Server and the SSLC

A naive approach to query verification requires the oracle to provide a list of all transactions used to compute the result. The light client could then independently query full nodes to verify their presence in the blockchain. However, this approach results in query complexity linear in the number of transactions, making it impractical for large-scale computations. For example, a simple P2PKH (Pay-to-PubKey-Hash) transaction on the Bitcoin network with one UTXO input and one output is large 200 bytes [15], and the Bitcoin RPC call getwartransaction [16] with the full verbose format will add 1KB metadata, leading to 1.2KB per transaction. For a computation involving 1M transactions, the client will download 1.2GB of data from each full node, which is unrealistic for a resource-constrained device.

Instead, we use SNARK proofs, which allow a prover to generate a succinct proof of correct computation. The oracle server computes the map-reduce query in sub-tasks and, for each sub-task, returns the result alongside a SNARK proof $\tilde{\pi}$. The proof must ensure that (i) the function $f_{\mathsf{MR}}(\cdot)$ was correctly executed, and (ii) each transaction used belongs to the transaction root $\mathbb{R}_T^{b_i}$ of some block b_i. By proving the existence of transactions within their respective blocks, the SSLC only needs to check the existence of those blocks in B.

In the rest of this section, we describe the proof system of our SSLC interactive protocol. As we said above, the prover produces a SNARK per f_{MR} computation. In particular, given

a set of transaction roots $\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}$ with $1 \leq i_1 < i_2 < \cdots < i_m \leq |B|$[5], a public claim $x := (\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, r)$ and a private witness $w := (B)$, an oracle server is able to generate a SNARK proof $\tilde{\pi}$ when x and w satisfy the following relation $\mathcal{R}$:

- $f_{\mathsf{MR}}(B) = r$ and
- each transaction tx, used to compute f_{MR} and contained in a block b_i of $|B|$, is a leaf of the transaction Merkle Tree identified with $\mathbb{R}_T^{b_i}$.

To compute such SNARK proof and ensure secure querying in our protocol, we need to consider two issues. First of all, a dishonest oracle server may deviate from the correct computation of f_{MR} in various ways (e.g., she can discard some transactions during the computation or use multiple times a single transaction during the computation) still producing an accepting proof $\tilde{\pi}$. In this section, we are presenting some techniques to ensure the secure querying in our protocol.

Second, f_{MR} could be computed over many transactions (in the order of magnitude of thousands or tens of thousands of transactions) and, therefore, computing such proof for such a large claim may require enormous computational resources for the oracle server or even be infeasible. In the following, we present a technique to improve the scalability for the prover exploiting the map-reduce paradigm and, in particular, its capacity of dividing the load of computing a larger task into smaller sub-tasks.

Preventing malicious behavior of the oracle server. A dishonest server may deviate from the correct computation of f_{MR} while still being able to produce a proof $\tilde{\pi}$ that is accepted by an SSLC when:

- the server does not include some transactions during the computation of f_{MR} that should be included (because these involve the account acc_1);
- the server includes multiple times the same transaction tx omitting others in the same block;

To ensure that the oracle server includes all the transactions to compute f_{MR}, the SSLC must compare, during the $\mathsf{Vrfy}(\cdot)$ protocol, the st received by the oracle server with the values of st retrieved by a set of independent full nodes. This means that st has to contain information on the cardinality of transactions involved in the computation of f_{MR}, such as the number of transactions, and that can be easily retrieved by full nodes (for example, with RPC calls). We will discuss more details about this aspect in the next section.

To guarantee the uniqueness of processed transactions, given a set of transactions, the prover must show that the hash of a transaction is different from any other hash used to compute f_{MR} in the same block.

[5]This set of transaction roots is composed of transaction roots belonging to blocks that are not necessarily subsequent in the ledger.

An efficient approach to prove this statement is to sort the set of transactions $\{\mathsf{tx}_1, \ldots, \mathsf{tx}_{k'}\}$ (computed in step 1 of the map_i algorithm described in Figure 4) according to their transaction hash and, then, verify the Merkle path of each transaction following the order of their hash checking. In particular, for every transaction, the prover must show that its transaction hash is strictly greater than the previous transaction in this ordered transactions set.

Hence, in the light of this discussion, we must modify the relation as follows: given an integer k and a set of transaction roots $\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}$ with $1 \leq i_1 < i_2 < \cdots < i_m \leq |B|$[5], a public claim $x := (\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, k, r)$ and a private witness $w := (B)$, an oracle server is able to generate a SNARK proof $\tilde{\pi}$ when x and w satisfy the following relation $\mathcal{R}'$:

- $f_{\mathsf{MR}}(B) = r$ and,
- the total number of transaction involved in the computation of $f_{\mathsf{MR}}(B)$ is k and,
- each transaction tx, used to compute f_{MR} and contained in a block b_i of $|B|$, is a leaf of the transaction Merkle Tree identified with $\mathbb{R}_T^{b_i}$ and,
- for each transaction tx, used to compute f_{MR} and contained in a block b_i of $|B|$, its hash is different from the hash of any other transactions in b_i used to compute f_{MR}.

Recursive proving of transaction batches. As we said above, f_{MR} could be computed over many transactions (in the order of magnitude of thousands or tens of thousands of transactions) and, therefore, computing such proof for such a large claim may require enormous computational resources for the oracle server or even be infeasible. We want to devise a system that is advantageous for the verifier when the number of transactions on which f_{MR} becomes prohibitively large, ensuring, at the same time, that proof generation remains feasible for the prover.

In particular, the size of the claim scales mostly linearly with the number of transactions included in the computation but remains sublinear to the total number of transactions in a block. Specifically, given a block b_i containing $N = |T_{b_i}|$ transactions, we aim to prove a subset of $M < N$ transactions. For each transaction in this subset, the SNARKs prove the correctness of a Merkle proof against a tree of N leaves with root $\mathbb{R}_T^{b_i}$, resulting in $O(M \log N)$ hash computations.

This is a reasonable scenario because in modern blockchain systems, the number of transactions per block continues to expand to support higher throughput [17, 18], generating an SSLC proof for an entire block in a single arithmetic circuit could become infeasibly expensive. To address this challenge, we propose a recursive SNARK composition approach [19]. Inspired by the insight of [20, 21] which demonstrates that generating many smaller proofs is often more efficient than computing a single large proof for an extensive claim, we decide to adopt recursion. This allows the oracle server to generate one SNARK per f_{MR} while still relaxing the memory requirement by splitting the claim into smaller sub-claims.

In particular, for each f_{MR}, we partition the set of transactions into batches, one per each block or interval of blocks. For each batch, we generate a SNARK which proves that map_i (or

multiple map functions) has been correctly computed on distinct transactions of the block b_i, that no transaction has been omitted, and that also verifies the proof of the previous block. Finally, a final SNARK proves that reduce has been correctly computed using as input the outputs computed by the map functions. Such an approach forms a linear recursive chain of proofs, ensuring that all inner proofs are verified if the final proof is valid.

This recursive approach significantly reduces memory and computational costs, as each proof circuit processes a smaller claim. Furthermore, each recursive step incurs a fixed overhead for verifying the previous proof, which remains independent of the overall transaction set size, making recursion scalable for large proofs.

4.3. The SSLC Protocol

The SSLC interactive protocol is illustrated in Figures 5 and 6. The protocol involves an SSLC L communicating with an Oracle server S and a set of full nodes $\mathsf{FN}_1, ..., \mathsf{FN}_n$ under the assumption of *existential honesty*. At a high level, the client submits a request for the execution of a map-reduce blockchain query f_{MR}. The oracle server processes the query using the map-reduce paradigm and returns the computed result along with a proof certifying the correctness of the computation.

To validate the proof, the SSLC interacts with the full nodes to retrieve the information st necessary for verification, such as the transaction roots of the blocks specified by the prover and the corresponding count of transactions involved in the computation. Finally, the client verifies the proof against the acquired data; if the verification succeeds, the result is accepted; otherwise, it is rejected.

The SSLC protocol is formally defined as the tuple $(\mathsf{Q}(\cdot), \mathsf{Vrfy}(\cdot))$, where: - $\mathsf{Q}(\cdot)$ represents the *query phase*, in which the SSLC sends the request of computing a function f_{MR} to the oracle server and collects the results r along with a SNARK proof π and a view of the blockchain st. - $\mathsf{Vrfy}(\cdot)$ represents the *verify phase*, in which the SSLC connects to n distinct full nodes to collect st and validate the proof.

Query phase. The protocol $\mathsf{Q}(\cdot)$, described in Figure 5, is executed by a light client L and an Oracle server S. In the first step, L connects to S, sending the description of a map-reduce query f_{MR} and the S with input the blockchain B.

Upon receiving the request, the server executes $f_{\mathsf{MR}}(B)$ according to the description of Figure 4.

The oracle server also produces a SNARK proof certifying the correctness of the computation of f_{MR}. Specifically, the proof attests that the server correctly applied the function f_{MR} over a subset of distinct transactions belonging to a set of transaction roots belonging to a not necessarily subsequent set of blocks (i.e., $\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}$ with $1 \leq i_1 < i_2 < \cdots < i_m \leq |B|$). Once the server ends the computation, the server sends to the SSLC:

- The final computed result r.
- A proof π containing:
  - A SNARK proof $\tilde{\pi}$.

– A view of the blockchain st, including:
  * $\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}$: the transaction roots belonging to a not necessarily subsequent set of blocks and including the transaction on which f_{MR} has been computed;
  * k: the number of transactions processed by f_{MR}

The SSLC Protocol

Query Phase: $\mathsf{Q}\langle \mathsf{L}(f_{\mathsf{MR}}), \mathsf{S}(B)\rangle \rightarrow (r, \pi := (\mathsf{st} := (\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, k), \tilde{\pi}))$

- Protocol steps for the SSLC L:
  (1) Request. Connect to the oracle server S:
  - send a map-reduce query f_{MR}

  (2) Update. Upon receiving (r, π) from S:
  - store and outputs (r, π).
- Protocol steps for the oracle server S:
  (1) Execute. Upon receiving a request for f_{MR}:
  - Compute $r \leftarrow f_{\mathsf{MR}}(B)$ according to Figure 4
  - Generate a SNARK for the relation $\mathcal{R}'$ described in Sec. 4.2

  (2) Respond. Send $(r, \pi := (st := (\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, k), \tilde{\pi}))$ to L

Figure 5: Description of the Query phase of the SSLC protocol.

Verify phase. In the verify phase, the SSLC client L interacts with a set of n full nodes $\mathsf{FN}_1, ..., \mathsf{FN}_n$ running the $\mathsf{Vrfy}(\cdot)$ protocol described in Figure 6. The main goal of this protocol is to accept or reject the final result provided by the server S in the query phase.

The light client retrieves from the full nodes: (i) the total number k' of transactions sent or received by an account acc_1, and (ii) the transaction roots $\{\mathbb{R}'^{b_{i_1}}_T, \ldots, \mathbb{R}'^{b_{i_m}}_T\}$ from canonical blocks in B.

Full nodes respond if and only if there exists a simple query function $f(\cdot)$ that, given a chain B and additional information (e.g., the blockchain account address acc_1), returns the expected result with a minimal computation task. The light client proceeds only if there is no contradiction among the answers of the full nodes. If only one answer differs from on of another full node, the SSLC aborts the protocol.

Upon receiving the responses from all n full nodes, the client proceeds with the verification steps. The first step is to verify the correctness of the SNARK proof on the claim $x := (\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, k, r)$. If the SNARK verification succeeds, the SSLC proceeds to the next step.

The SSLC Protocol

Verify Phase: $\mathsf{Vrfy}\langle \mathsf{L}(f_{\mathsf{MR}}, r, \pi), \mathsf{FN}_{1,\ldots,n}(B)\rangle \rightarrow b$

- Protocol steps for the SSLC L:
  - **(1) Request.** For each FN_i:
    - request transactions count k for account acc_1 on B;
    - request transactions roots for blocks $b_{i_1}, \ldots, b_{i_m}$ from B;
  - **(2) Update.** Upon receiving an answer from FN_i:
    - if k' received by FN_i is different from a previously received one or it has received $\perp$, **abort**; otherwise **continue**;
    - if the set $\{\mathbb{R}'^{b_{i_1}}_T, \ldots, \mathbb{R}'^{b_{i_m}}_T\}$ received by FN_i is different from a previously received one or it has received $\perp$, **abort**; otherwise **continue**;
  - **(4) Verify.** Upon receiving n responses from full nodes:
    - check if $k' \neq k$; if yes, set $b \rightarrow 0$ and **return**;
    - check if $\{\mathbb{R}'^{b_{i_1}}_T, \ldots, \mathbb{R}'^{b_{i_m}}_T\} \neq \{\mathbb{R}^{b_{i_1}}_T, \ldots, \mathbb{R}^{b_{i_m}}_T\}$; if yes, set $b \rightarrow 0$ and **return**;
    - check if the $\tilde{\pi}$ on the claim $x = (\{\mathbb{R}^{b_{i_1}}_T, \ldots, \mathbb{R}^{b_{i_m}}_T\}, k, r)$ is correctly verified; if no, set $b \rightarrow 0$ and **return**;
    - set $b \rightarrow 1$ and **return**;
- Protocol steps for the full node FN_i:
  - **(1) Respond Count.** Upon receiving a request for acc_1 on chain B:
    - **if** acc_1 exists, send the number of transactions sent or received by acc_1 to L; otherwise, send $\perp$;
  - **(2) Respond Block.** Upon receiving a request for a transaction root of b_i:
    - **if** exists b_i exists, send the transaction root $\mathbb{R}'^{b_i}_T$ to L; otherwise, send $\perp$

Figure 6: Description of the Verify phase of the SSLC protocol.

Security and efficiency analysis of the SSLC protocol. We want now to evaluate the security of the SSLC protocol with respect to the secure querying property described in Section 3.3. This property says that it is computationally hard for a malicious oracle server to produce an accepting proof π if: 1) the attacker computes f_{MR} on a blockchain view st' different from the one the SSLC can retrieve by the full nodes or 2) the attacker computes f_{MR} on transactions that are not in the blockchain or 3) the attacker does not correctly compute f_{MR}.

In 1), the attacker uses the following strategy: she generates an accepting SNARK proof $\tilde{\pi}$ for an ad-hoc realized st' that differs from st. To achieve this goal, since during $\mathsf{Vrfy}(\cdot)$ L contacts n independent full nodes, the malicious server should be able to corrupt all these full nodes. However, if she achieves in doing so, the existential honesty assumption of our system would be violated. Hence, the attacker cannot use this strategy.

In 2), the attacker uses the following strategy: she generates an accepting SNARK proof $\tilde{\pi}$ for an ad-hoc realized set of transactions. This means that the attacker is able to find at least another transaction such that its hash corresponds to one of the transactions involved

in the computation of f_{MR}. However, if she succeeds in doing so, it means that the attacker can find a collision for a collision-resistant hash function, which is something that happens only with negligible probability. Hence, the attacker cannot use this strategy.

In 3), the attacker deviates from the algorithm to compute f_{MR} in the following ways: a) she omits some transactions that should be involved during the computation of f_{MR}; b) she uses a transaction involved in the computation of f_{MR} multiple times; c) she deviates from f_{M} or f_{R}. Since L, during $\mathsf{Vrfy}(\cdot)$ protocol, obtains the number k of involved transactions from the full nodes, if the attacker succeeds in a), it means either that she corrupts the full nodes, violating the existential honesty assumption of our system or it can break the knowledge soundness property of the SNARK but this happens with negligible probability. If the attacker succeeds in b) and c), it means that the attacker is able to break the knowledge soundness property of the SNARK but this happens with negligible probability. Hence, the attacker cannot use this strategy.

Such arguments discussed so far guarantee that our SSLC protocol ensures secure querying.

We are left with showing that our SSLC protocol respects the efficiency property described in Section 3.3. During $\mathsf{Q}(\cdot)$, the SSLC only needs to send the description of f_{MR} to S, therefore, respecting efficient storage, communication, and client computation. During $\mathsf{Vrfy}(\cdot)$, the SSLC needs to download (from the full nodes) the claim of $\mathcal{R}'$ on which the SNARK is computed which is $(\{\mathbb{R}_T^{b_{i_1}}, \ldots, \mathbb{R}_T^{b_{i_m}}\}, k, r)$. Then, it reads such a claim while verifying SNARK. By definition of SNARK, the dimension of such proof is sublinear to the size of the transaction list $|T|$, which is included in the witness B of $\mathcal{R}'$. The reply r to f_{MR} is independent of the size of the blockchain because it is aggregated data and st is composed of a set of transaction roots of the blocks containing the transactions on which f_{MR} is computed. By definition of the Merkle Tree, the size of such a set of transaction roots is sublinear to the set of transactions involved in f_{MR}. Hence, since $\tilde{\pi}$, st, and r are sublinear to $|T|$, the efficient storage and communication properties are respected.

Finally, we need to evaluate whether the computational complexity of the SSLC protocol is sublinear to the size of the transaction list $|T|$. Let us suppose that f_{MR} is computed over all the transactions in $|T|$. In the SSLC, we compute $\mathsf{Q}(\cdot)$ in an amount of time that is logarithmic with respect to $|T|$ because the SSLC only needs to output the transaction roots of the blockchain. In $\mathsf{Vrfy}(\cdot)$, the SSLC executes the algorithm in logarithmic time with respect to $|T|$ because, in the protocol, it downloads a constant number (i.e., k) and the transaction roots, whose number is logarithmic w.r.t. the number of transactions, and it verifies a SNARK that by definition is verified in time linear to its claim (i.e., the transaction roots) plus an amount of time sublinear w.r.t. its witness (i.e., B that contains T).

5. Use Case Analysis of SSLC Protocol

In this section, we show how to adopt the SSLC protocol in two concrete application use cases: a) a Bitcoin SSLC that queries the oracle server to process a statistical analysis on the exchange volume of a given Bitcoin account and b) an Ethereum SSLC to compute statistics

over a given time interval on the votes of an on-chain voting system implemented by a Smart Contract.

To evaluate the performance of our SSLC protocol, we compare it with approaches relying on the Original Nakamoto Light Client (ONLC) and the classes of sublinear light clients (SLCs), introduced in Section 1 which are an optimization of ONLC allowing us to verify the presence of a transaction in a block in sublinear space.

The results of the analysis, summarized in Table 1, confirm that our SSLC approach provides a substantial reduction in data transfer, not only minimizing the bandwidth requirements but also enhancing the efficiency and scalability of blockchain clients, especially on resource-constrained devices.

Table 1: Comparison of the three approaches on the Bitcoin and Ethereum use cases.

	ONLC	SLC	SSLC
Bitcoin	726MB	201MB	15MB
Ethereum	1.1GB	1GB	9MB

5.1. The case of Bitcoin

We first evaluate our solution for monitoring BTC exchange volumes for a given account. Specifically, the oracle server answers queries on the average amount of BTC transferred to or received by a specific account.

To verify the correctness of a response to a query, the application light client executes the following steps: 1) it retrieves from a blockchain explorer the number of transactions involving a certain address; 2) it obtains from the server the answer to the query along with a proof showing that these transactions belong to specific blocks (identified with a hash) and how they contribute to the query result; 3) it queries full nodes to obtain block headers or block hashes for which the proof from the server exists; 4) it verifies the proof using these trusted hashes;

In our analysis, we use the Bitcoin address of Satoshi Nakamoto as an example. Three blockchain explorers, namely blockchain.com, mempool.space, and bitaps.com, agree that at block number 881358, Satoshi's address has been involved in 42381 transactions that can be stored in 32 bytes.

In the next, we use block 872028 with 3686 transactions to exemplify the computations necessary to compare the different solutions.

The oracle response includes the requested average amount (32 bytes) and the proof whose size depends on the circuit and typically ranges between 100 and 150 KB.

The RPC calls to answer this complex query are:

- `getblockhash`: 101 bytes to retrieve the block hash.
- `getblockheader`: 613 bytes to retrieve block headers.

- `gettxoutproof`: ∼1 KB for the Merkle tree proof of a known transaction ID.
- `getblock`: ∼10 MB to retrieve the full block.
- `getrawtransaction`: ∼3.8 KB for a transaction involving Satoshi.

The ONLC must store all block headers; at the time of writing they are 881358 (∼ 534 MB). Additionally, the client must download: all the 42381 transactions of the examined account (∼ 152 MB) and all Merkle Tree proofs (∼ 40 MB). In total, this approach requires approximately *726 MB*.

SLCs approaches need sublinear space. For the sake of simplicity, we assume they use log(size of the block header) = log(534) = 9.06 MB. The rest of the operations are the same as in the ONLC. Hence, in total, SLCs require approximately *201 MB*.

SSLC significantly reduces storage and computational requirements because 1) the light client only downloads 42381 block hashes (∼4 MB), 2) the number of transactions involving the target account (32 bytes) and 3) the Plonky2 proof from the oracle server (150 KB). Thus, the total downloaded data required to verify the oracle server's answer is $<$*5 MB*. Since we assume that a light client retrieves information from multiple full nodes via RPC, it will download this data multiple times. If three full nodes are contacted, the total download size is approximately *15 MB*.

5.2. The case of Ethereum

In this subsection, we present the application of our framework to the verification of statistics on an on-chain voting system governed by a smart contract.

The smart contract records votes for different candidates as a key(candidate)-value(number of votes) store. In Appendix A, we provide the pseudo-code of a simplified Solidity smart contract for this use case.

Our SSLC aims to verify the average voting participation over a given time interval.

To verify the correctness of the voting participation query, the SSLC executes the following steps: 1) The SSLC queries the smart contract through an RPC to retrieve the number of voting transactions associated with a candidate. 2) The SSLC obtains from the oracle server the computed answer along with a cryptographic proof that demonstrates how the retrieved transactions contribute to the final result. The proof guarantees that the transactions and their receipts are included in m blocks. 3) The SSLC fetches the relevant block headers from a set of full nodes. 4) Given the trusted headers, the SSLC verifies the proof provided by the oracle.

We consider an election lasting one day, spanning approximately 7000 blocks, with a candidate receiving 70000 votes, namely an average of 10 votes for every block.

The SSLC fetches the number of votes for a candidate from the smart contract, represented as a 32-byte value.

The oracle response includes the requested average votes (32 bytes) and the proof, whose size depends on the circuit and typically ranges between 100 and 150 KB.

The RPC calls to answer this complex query are:

- `ots_getBlockDetails`: $\sim$ 3 KB to return the details of the block with the specified block number. It is similar to the `eth_getBlockByNumber/eth_getBlockByHash` method, but an optimized version.

- `getProof`: $\sim$ 7.8 KB to obtain the account and storage values of the specified account including the Merkle proof.

The ONLC needs $\sim$ 21 MB to store the block headers corresponding to the election period. Additionally, considering the transaction with hash `0x12e86e5fa174562c61efc03369573ff7058c70229450fd1660e5db4122c4071d` as the reference to compute the following values, it must download: a) all the $\sim$ 70000 involved transactions, using the getTransactionByHash RPC call ($\sim$ 59 MB)b) all the Merkle Tree proofs using the proof_getTransactionByHash ($\sim 476MB$) , c) all the receipts of the transactions involved using the getTransactionReceipt ($\sim 112MB$) , d) the related MT proofs (again 476 MB) and finally, e) the proof that a smart contract is in the correct state ($\sim$ 7.8 KB). In total, this approach requires approximately *1.1 GB*.

SLC in this case downloads $\log(21) = 4.3$ MB. The rest of the operations are the same as in the ONLC. Hence, in total, SLCs require approximately *1 GB*.

SSLC significantly reduces storage and computational requirements. Indeed it downloads only 1000 block headers (3 MB) and verifies the proof received from the oracle. Even when contacting multiple full nodes (e.g., 3 nodes), the total download remains around 9 MB. The final verification step of the proof, assuming a Plonky2 proof of 150 KB, results in an overall memory requirement of approximately 9 MB, a considerable improvement over the other approaches.

6. Experiments

In this section, we present our experimental evaluation of the SSLC proving system, demonstrating that an oracle server with no resource constraints can efficiently generate SNARK proofs attesting to the correctness of map-reduce query executions. Our experiments were conducted on a high-performance server equipped with an Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz (64 cores) and 512 GB of RAM.

We implemented a SNARK circuit in Plonky2, with the source code publicly available on GitHub[6]. The implementation simulates the load for the prover and the verifier in realistic scenarios, such as the one of Bitcoin average use case introduced in Section 5.1. Specifically, our implementation verifies Merkle membership proofs while computing the average value of a set of Bitcoin transfer transactions.

Our results show that, even when handling many transactions per circuit, our proof system achieves reasonable memory consumption and proof generation time, while verification remains highly efficient, requiring minimal computational and bandwidth resources.

[6] `https://github.com/deanstef/awesome-plonky2`

6.1. Technical Choices

Plonky2. We construct our circuit using Plonky2 [22], a framework for recursive SNARKs. Plonky2 achieves scalability by combining PLONK-based arithmetization [23] with FRI-based polynomial commitment schemes [24]. Unlike traditional commitment schemes, FRI provides a transparent, trustless, and hash-based approach, eliminating the need for a trusted setup while enabling efficient polynomial proximity testing. This results in fast proof generation with efficient verification costs. Additionally, Plonky2 leverages the 64-bit Goldilocks field, which is optimized for arithmetic operations on modern hardware. Plonky2 generates polylogarithmic proof sizes while utilizing lightweight hash-based verification.

Circuit Configuration. Our Plonky2 circuit is instantiated using the Plonky2 CircuitBuilder with a customized configuration to balance performance and overhead. We set num_wires = 135 to accommodate the circuit's computational needs without introducing excessive overhead. The transaction Merkle tree is constructed using Plonky2's MerkleTree implementation, configured with the Goldilocks field and the Poseidon hash function [25]. Each transaction leaf is converted into four field elements to comply with Plonky2's hashing constraints. Specifically, Poseidon in Plonky2 is optimized for a state width of four elements, ensuring a balance between performance, in-circuit hashing efficiency, and security against collision attacks.

Note that, Poseidon is not yet widely adopted in production-ready blockchains. However, SNARK-friendly hash functions like Poseidon are increasingly seen as essential for blockchain scalability. For example, the transition toward Poseidon has been actively discussed for enabling stateless Ethereum validation via zk-STARKs[7]. Moreover, newer zk-friendly blockchain architectures—including Mina[8] and StarkNet[9]—have already adopted Poseidon as their core cryptographic hash function.

Consequently, we chose not to tailor our implementation to legacy, SNARK-unfriendly hash functions that are likely to become obsolete. Instead, our evaluation is designed to be future-proof, aligned with emerging blockchain developments. That said, it is important to highlight that our SSLC protocol remains compatible with alternative blockchain designs that utilize different hash functions, ensuring flexibility for diverse blockchain ecosystems.

6.2. Performance Evaluation

In this subsection, we present the results of our experiments and evaluate the performance and feasibility of our solution. We simulate the load for the prover and the verifier in realistic scenarios (such as the one of Bitcoin average use case) measuring the prover costs associated with generating SNARK proofs for the map-reduce task. Additionally, we assess the memory consumed by the SSLC during the verification process. Our experiments demonstrate that the solution is practical, enabling clients to verify map-reduce queries involving millions of

[7] https://vitalik.eth.limo/general/2024/10/23/futures4.html
[8] https://minaprotocol.com
[9] https://www.starknet.io

transactions while downloading only a few megabytes of data and using ≈ 2 GB of memory for near-instantaneous verification.

To simulate a heavy workload, we handle a block b_i containing a large transaction set of size 2^{20}. This means that the transactions will be the hashes of a Merkle Tree with 2^{20}. The transactions are organized in a Merkle tree using the Plonky2 library $\mathsf{MerkleTree}::\langle \mathsf{F}, \mathsf{PoseidonHash} \rangle::\mathsf{new}()$, where Poseidon is used as the cryptographic hash function, and F is instantiated with the Goldilocks field.

We conduct six experiments, each varying the number of transactions processed in the map-reduce computation. Specifically, we evaluate the test varying the size of the set of transactions on which f_{MR} is computed among $10^1, 10^2, 10^3, 10^4, 10^5$ and 10^6. For each experiment, we measure the time required to generate and verify the final proof, the proof size, and the memory consumption for both the prover and the verifier. The experimental results are summarized in Table 2.

Workload	**Proof Size (KB)**	**Prover**		**Verifier**	
		Time (s)	Memory (GB)	Time (s)	Memory (GB)
10^1	96.5	0.314	0.213	0.010	0.213
10^2	124.8	3.418	0.320	0.020	0.270
10^3	156.73	29.088	1.110	0.020	0.717
10^4	156.98	304.273	1.420	0.010	1.040
10^5	156.98	3052.504	1.480	0.010	1.090
10^6	156.98	30434.426	2.060	0.010	1.660

Table 2: Performance measurements for different numbers of leaves.

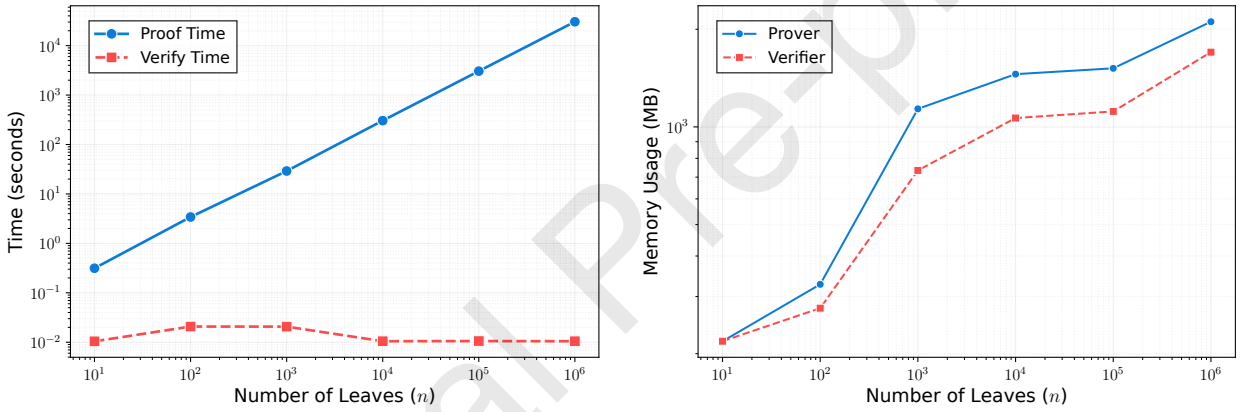
In our implementation, we have adopted a recursive proving approach, where at each recursive step our implementation handles a batch of 1000 transactions. This batch size simulates a reasonable number of transactions selected by f_{MR} per block during the map phase of the algorithm described in Figure 4. With this configuration, experiments involving up to 1000 transactions were executed within a single circuit step (i.e., without recursion). For recursive cases, we observed that the average recursion time per step to generate the proof was approximately 30 seconds.

In our Bitcoin average example, each circuit execution sequentially processes the transactions within a block. For each transaction, the circuit performs the following checks: (i) it verifies the Merkle path against the provided root $\mathbb{R}_T^{b_i}$, (ii) it verifies that f_{MR} has selected the relevant transactions (i.e., those sent or received by a specific account), (iii) it verifies that transactions are computed only once (through the ordering technique described in Section 4.2), and (iv) it computes the average. Additionally, in every recursive step, the circuit verifies the previous proof, which requires loading into the memory a proof for a precedent block along with its corresponding public inputs.

Figure 7a presents the measured times for generating and verifying the recursive SNARK. As expected, the prover time increases almost linearly with the number of processed transac-

tions, while the verifier time remains constant, around 10–20 ms. The memory consumption, as illustrated in Figure 7b, follows a similar trend for both the prover and the verifier. In Plonky2, verifying a recursive SNARK requires storing inner proof elements, including FRI commitments and evaluation points at multiple FRI layers [24], resulting in memory overheads similar to those of proving. However, for the SSLC, we observe that even in the worst-case scenario—processing a map-reduce query with 10^6 transactions—the verifier's memory consumption remains at 1.6 GB, which is within acceptable limits for resource-constrained devices.

The proof size grows up to the batch size. Specifically, we found that a SNARK proof computing 1000 transactions resulted in a proof size of 156.73 KB. By employing a recursive batching approach, we successfully processed large transaction sets without incurring excessive proof size overhead. Notably, the size of a recursive SNARK proof stabilized at 156.98 KB, remaining constant regardless of the number of recursive steps, with only a minor additional overhead from inner proofs. This result has significant implications for SSLC clients: even in the worst-case scenario, where millions of transactions are processed, the client only needs to download less than 200 KB of data per map-reduce query.



(a) Time Comparison

(b) Memory Consumption

Figure 7: Performance analysis: time and memory comparison.

6.3. Discussion

This analysis highlights that for map-reduce queries that involve a large number of transactions distributed across a limited number of blocks, our solution outperforms the ONLC and SLC approaches discussed in Section 5, enabling light clients to verify results with minimal computational and storage resources. However, in scenarios where queries span over a large number of blocks, the benefits of this approach become less apparent. In these cases, the SSLC has to download more transaction roots/block hashes, enhancing the amount of communication memory used during the protocol, making it more similar to that of ONLC and SLC.

Prover optimization with parallel SNARKs. In the current implementation, the prover time scales linearly with the number of map_i tasks, i.e., it increases by a factor of X, where X is the number of blocks involved in the computation. Consequently, for applications requiring the processing of a large number of transactions across many blocks, the time required to generate SNARK proofs could become prohibitively long.

However, map-reduce queries exhibit a high degree of parallelism. Hence, to optimize proving efficiency, we propose leveraging a distributed proving technique, similar to the approach adopted in [3]. In this scheme, recursive SNARK computations are offloaded to separate hardware units, and the resulting proofs are subsequently aggregated on a central node. This parallelized approach significantly reduces proving time, keeping it close to that of a single map_i recursive SNARK, irrespective of the number of map_i executions in parallel.

7. Acknowledgments

The work of Marco Zecchini was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The work of Andrea Vitaletti was partially supported by project PE11 - MICS (Made in Italy – Circular and Sustainable) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. Ivan Visconti is member of the Gruppo Nazionale Calcolo Scientifico Istituto Nazionale di Alta Matematica (GNCSINdAM). The research of Stefano De Angelis and Ivan Visconti is financially supported under the National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.1, Call for tender No. 104 published on 2.2.2022 by the Italian Ministry of University and Research (MUR), funded by the European Union-NextGenerationEU - Project Title "PARTHENON"-CUP D53D23008610006 - Grant Assignment Decree No. 959 adopted on June 30, 2023 by the Italian Ministry of Ministry of University and Research (MUR).

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2009).
URL `http://www.bitcoin.org/bitcoin.pdf`

[2] P. Chatzigiannis, F. Baldimtsi, K. Chalkias, SoK: Blockchain light clients, in: I. Eyal, J. A. Garay (Eds.), FC 2022, Vol. 13411 of LNCS, Springer, Cham, 2022, pp. 615–641. `doi:10.1007/978-3-031-18283-9_31`.

[3] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, D. Song, zkBridge: Trustless cross-chain bridges made practical, in: H. Yin, A. Stavrou, C. Cremers, E. Shi (Eds.), ACM CCS 2022, ACM Press, 2022, pp. 3003–3017. `doi:10.1145/3548606.3560652`.

[4] E. N. Tas, D. Zindros, L. Yang, D. Tse, Light clients for lazy blockchains, Cryptology ePrint Archive, Report 2022/384, accepted at FC 2024 (2022).
URL `https://eprint.iacr.org/2022/384`

[5] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113. doi:10.1145/1327452.1327492.

[6] A. Kiayias, N. Lamprou, A.-P. Stouka, Proofs of proofs of work with sublinear complexity, in: J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, K. Rohloff (Eds.), FC 2016 Workshops, Vol. 9604 of LNCS, Springer, Berlin, Heidelberg, 2016, pp. 61–78. doi:10.1007/978-3-662-53357-4_5.

[7] A. Kiayias, A. Miller, D. Zindros, Non-interactive proofs of proof-of-work, in: J. Bonneau, N. Heninger (Eds.), FC 2020, Vol. 12059 of LNCS, Springer, Cham, 2020, pp. 505–522. doi:10.1007/978-3-030-51280-4_27.

[8] B. Bünz, L. Kiffer, L. Luu, M. Zamani, FlyClient: Super-light clients for cryptocurrencies, in: 2020 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 2020, pp. 928–946. doi:10.1109/SP40000.2020.00049.

[9] A. Kattis, J. Bonneau, Proof of necessary work: Succinct state verification with fairness guarantees, in: F. Baldimtsi, C. Cachin (Eds.), FC 2023, Part II, Vol. 13951 of LNCS, Springer, Cham, 2023, pp. 18–35. doi:10.1007/978-3-031-47751-5_2.

[10] H. Abusalah, G. Fuchsbauer, P. Gazi, K. Klein, SNACKs: Leveraging proofs of sequential work for blockchain light clients, in: S. Agrawal, D. Lin (Eds.), ASIACRYPT 2022, Part I, Vol. 13791 of LNCS, Springer, Cham, 2022, pp. 806–836. doi:10.1007/978-3-031-22963-3_27.

[11] P. Vesely, K. Gurkan, M. Straka, A. Gabizon, P. Jovanovic, G. Konstantopoulos, A. Oines, M. Olszewski, E. Tromer, Plumo: An ultralight blockchain client, in: I. Eyal, J. A. Garay (Eds.), FC 2022, Vol. 13411 of LNCS, Springer, Cham, 2022, pp. 597–614. doi:10.1007/978-3-031-18283-9_30.

[12] Y. Lu, Q. Tang, G. Wang, Generic superlight client for permissionless blockchains, in: L. Chen, N. Li, K. Liang, S. A. Schneider (Eds.), ESORICS 2020, Part II, Vol. 12309 of LNCS, Springer, Cham, 2020, pp. 713–733. doi:10.1007/978-3-030-59013-0_35.

[13] S. Agrawal, J. Neu, E. N. Tas, D. Zindros, Proofs of Proof-Of-Stake with Sublinear Complexity, in: 5th Conference on Advances in Financial Technologies (AFT 2023), Vol. 282, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 1–24. doi:10.4230/LIPIcs.AFT.2023.14.

[14] W. Wang, T. V. Cutsem, Depermissioning web3: a permissionless accountable rpc protocol for blockchain networks, accepted at IEEE ICDCS 2025 (2025). arXiv:2506.03940.
URL https://arxiv.org/abs/2506.03940

[15] J. Hofmann, libtxsize - a library for automated bitcoin transaction-size estimates, CoRR abs/2102.12796 (2021). arXiv:2102.12796.
URL https://arxiv.org/abs/2102.12796

[16] Bitcoin.org, Bitcoin core rpc api: getrawtransaction, accessed: 2025-02-06.
URL https://developer.bitcoin.org/reference/rpc/getrawtransaction.html

[17] L. T. Thibault, T. Sarry, A. S. Hafid, Blockchain scaling using rollups: A comprehensive survey, IEEE Access 10 (2022) 93039–93054. doi:10.1109/ACCESS.2022.3200051.

[18] Q. Zhou, H. Huang, Z. Zheng, J. Bian, Solutions to scalability of blockchain: A survey, IEEE Access 8 (2020) 16440–16455. doi:10.1109/ACCESS.2020.2967218.

[19] N. Bitansky, R. Canetti, A. Chiesa, E. Tromer, Recursive composition and bootstrapping for snarks and proof-carrying data, in: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, STOC '13, Association for Computing Machinery, New York, NY, USA, 2013, p. 111–120. doi:10.1145/2488608.2488623.

[20] P. Della Monica, I. Visconti, A. Vitaletti, M. Zecchini, Trust Nobody: Privacy-Preserving Proofs for Edited Photos with Your Laptop , in: 2025 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2025, pp. 14–14. doi:10.1109/SP61157.2025.00014.

[21] S. Dziembowski, S. Ebrahimi, P. Hassanizadeh, VIMz: Verifiable image manipulation using folding-based zkSNARKs, Cryptology ePrint Archive, Paper 2024/1063, accepted at Privacy Enhancing Technologies Symposium (PETS) 2025. (2024).
URL https://eprint.iacr.org/2024/1063

[22] P. Zero, Plonky2: Fast recursive arguments with plonk and fri, GitHub repository (2022).
URL https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf

[23] A. Gabizon, Z. J. Williamson, O. Ciobotaru, PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, Cryptology ePrint Archive, Report 2019/953, https://eprint.iacr.org/2019/953 (2019).

[24] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev, Fast reed-solomon interactive oracle proofs of proximity, in: I. Chatzigiannakis, C. Kaklamanis, D. Marx, D. Sannella (Eds.), ICALP 2018, Vol. 107 of LIPIcs, Schloss Dagstuhl, 2018, pp. 14:1–14:17. doi:10.4230/LIPIcs.ICALP.2018.14.

[25] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger, Poseidon: A new hash function for zero-knowledge proof systems, in: M. Bailey, R. Greenstadt (Eds.), USENIX Security 2021, USENIX Association, 2021, pp. 519–535.

Appendix A. Solidity Smart Contract: On-chain Voting Mechanism

We provide the pseudo-code of a Solidity smart contract that implements the on-chain voting mechanism described in Section 5.2.

```
contract Voting {
    struct Candidate {
        string name;
        uint voteCount;
    }

    mapping(uint => Candidate) public candidates;
    uint public candidatesCount;

    constructor(string[] memory candidateNames) {
        for (uint i = 0; i < candidateNames.length; i++) {
            candidates[i] = Candidate(candidateNames[i], 0);
            candidatesCount++;
        }
    }

    function vote(uint candidateId) public {
        require(candidateId < candidatesCount, "Invalid candidate");
        candidates[candidateId].voteCount++;
    }

    function getVotes(uint candidateId) public view returns (uint) {
        require(candidateId < candidatesCount, "Invalid candidate");
        return candidates[candidateId].voteCount;
    }
}
```

Listing 1: Voting Smart Contract in Solidity

The contract defines a simple voting system where candidates are stored in a mapping, each identified by an index. Each candidate has a name and a vote count. The constructor initializes the contract with a predefined list of candidates, each starting with zero votes.

To allow users to vote, the contract provides a `vote` function that takes a candidate ID as input and increments the corresponding vote count. A `require` statement ensures that the candidate ID is valid before updating the count. Additionally, a `getVotes` function allows users to retrieve the number of votes a candidate has received.

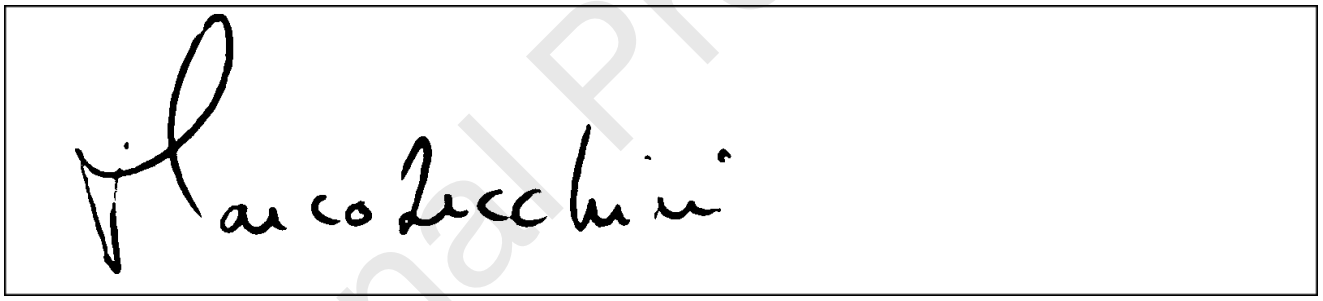
This implementation provides a minimal and functional voting mechanism but does not include security features such as preventing double voting, authentication, or encryption of votes. However, we can use it to show the incremental counter per candidate as the crucial state parameter for a SSLC. In particular, the counter represents the number of vote transactions emitted per candidate and can be used by a light client in the SSLC protocol to infer computation completeness of the map-reduce query.

Declaration of Interest Statement

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for this journal and was not involved in the editorial review or the decision to publish this article.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:



Stefano De Angelis: Investigation, Software, Data curation, Writing- Original draft:**Ivan Visconti**:Conceptualization, Writing - Review & Editing,Supervision:**Andrea Vitaletti**:Conceptualization, Writing - Review & Editing, Supervision:**Marco Zecchini**: Investigation, Methodology, Writing- Original draf,Validation: