SoK: Blockchain Light Clients

Panagiotis Chatzigiannis[1(✉)], Foteini Baldimtsi[1], and Konstantinos Chalkias[2]

[1] George Mason University, Fairfax, VA, USA
{pchatzig,foteini}@gmu.edu
[2] Mysten Labs, Palo Alto, CA, USA
kostas@mystenlabs.com

Abstract. Blockchain systems, as append-only ledgers, are typically associated with linearly growing participation costs. Therefore, for a blockchain client to interact with the system (query or submit a transaction), it can either pay these costs by downloading, storing and verifying the blockchain history, or forfeit blockchain security guarantees and place its trust on third party intermediary servers.

With this problem becoming apparent from early works in the blockchain space, the concept of a *light client* has been proposed, where a resource-constrained client such as a browser or mobile device can participate in the system by querying and/or submitting transactions without holding the full blockchain but while still inheriting the blockchain's security guarantees. A plethora of blockchain systems with different light client frameworks and implementations have been proposed, each with different functionalities, assumptions and efficiencies. In this work we provide a systematization of such light client designs. We unify the space by providing a set of definitions on their properties in terms of provided functionality, efficiency and security, and provide future research directions based on our findings.

Keywords: Blockchain · Light clients · Consensus · Long range attacks

1 Introduction

Blockchain-based, systems such as Bitcoin and Ethereum, typically include three types of participants: *consensus nodes* (also known as miners or validators), who run a consensus protocol to reach a common agreement on the current blockchain state, *full nodes* who store and communicate blockchain data, and *clients* which submit queries or transactions. Full nodes are considered to have relatively sufficient resources to perform their tasks, which involve communicating with each other through a gossip protocol in a peer-to-peer fashion, storing and communicating unconfirmed transactions, maintaining the entire blockchain history and

Panagiotis Chatzigiannis did part of this work during an internship at Novi Financial/Facebook Research. Konstantinos Chalkias did part of this work at Novi Financial/Facebook Research.

I. Eyal and J. Garay (Eds.): FC 2022, LNCS 13411, pp. 615–641, 2022.
https://doi.org/10.1007/978-3-031-18283-9_31

replying to queries. To perform transactions (e.g. in cryptocurrencies such as Bitcoin and Ethereum), clients first need to verify that the underlying blockchain is valid. Naively, this implies downloading and verifying all blocks, an operation that could take hours or days, and require gigabytes of bandwidth and storage. Therefore, the only remaining option for resource-constrained clients (such as mobile devices or browsers) is to place their trust on full nodes which will serve as intermediary servers, provide clients a view of the blockchain based on client queries, and forward submitted transactions on the client's behalf.

Nevertheless, in the early days of Bitcoin, the three roles mentioned were not necessarily distinct. For example, the Bitcoin core software [3] served as a common frontend to solve the Proof of Work puzzle as part of the consensus protocol, run a full node and submit queries and transactions. However, it quickly became necessary to decouple the client functionality to ensure less powerful clients can interact with the system while preserving as many security guarantees possible, which was mainly done through the Simplified Payment Verification (SPV) protocol [86]. Interestingly, while SPV required much less resources compared to a full node, it was still not lightweight enough to support resource-constrained environments with very low computational, storage and communication capabilities such as a mobile or browser-based client, while the introduction of more complex blockchain systems such as Ethereum made this gap even wider. In addition, SPV introduced additional trust assumptions and attack vectors, as in many implementations all communication and queries are executed through a small set of servers.

More recently, several implementations and academic works were proposed as "light clients" or "light-client friendly", either tailored to specific blockchain systems, or even as entirely new systems. However, every proposal provides different properties, definitions and goals for a light client, either implicitly or explicitly, while there are still many different interpretations for a "light client" in the blockchain space, even after a decade of evolution of cryptocurrencies. As a result, existing implementations approach the problem from a different angle, and no complete solution exists that makes a mobile client possible while maintaining all of the strong security guarantees of the underlying blockchain system.

Our Contributions. In this paper, we unify the diverse conception of light clients in the blockchain world by providing definitions for light client properties in terms of functionalities, efficiency and security, and provide a common list of assumptions for such clients. Then, we provide a systematization of prominent existing works based on our defined properties. Finally, through our systematization, we provide a series of insights and gaps serving as exciting future research directions, including considerations regarding long range attacks due to validator re-configurations, and light clients for privacy preserving blockchains or as smart contracts to allow for native interoperability between independent ledgers.

1.1 What is a Light Client?

We begin by providing an informal definition of a standard (non-light) client, which is the generic protocol that directly interacts with the blockchain system. This interaction includes at least one of the following functionalities:

- Perform queries (e.g. the balance of an account or the state of a transaction, with a specific time or block number as optional parameters). Such queries are typically accompanied by proofs verifiable by the client protocol (created by other entities in the system such as consensus participants or full nodes), in order to preserve security and prevent the client from being manipulated by malicious actors.
- Hold secret information (e.g. account private keys) and submit transactions to the blockchain system. This functionality is often referred to as a *wallet*.

Note that the terms *clients* and *wallets* are often considered equivalent and used interchangeably in the blockchain space, with the term "wallet" typically associated with a specific software implementation. However, based on the above informal definition, we make an explicit distinction between these terms: In a nutshell, a *wallet* is the software implementation of the *client* protocol that holds secret information used to submit transactions to the blockchain. As an example, Bitcoin Core [3] is Bitcoin's standard client which includes both wallet *and* full node functionalities, as discussed previously.

Starting again from the cryptocurrency community, a "light client" mostly refers to the wallet software running a "more" lightweight client implementation in its back-end compared to the standard client. This software usually interacts with the blockchain through a fully synchronized node, which in turn submits the transactions on the client's behalf (e.g. by placing them on a "mempool" and broadcasting them to other nodes and miners through a gossip protocol). The goal of such a client is to be more compatible with resource-constrained environments such as mobile devices or browsers, where the system's fully-fledged client might be prohibitive to work. Also, another goal of the light client might be to reduce the costs of the initial joining process, without requiring to download the full blockchain history (which for a standard client is typically in the order of gigabytes). The trade-off however for the efficiency of such clients is usually security; for instance they might need to trust the full node they are interacting with, they do not verify the consensus process, do not store and communicate ledger information themselves, and therefore do not contribute to decentralization, one of the blockchain's main goals.

However, in some implementations (e.g. Ethereum or Polkadot), a "light client" refers to a "lighter" version of a full node (i.e. with faster setup and synchronization time and lower computational/storage requirements), which only stores block headers but still directly interacts with the blockchain network in a peer-to-peer fashion, and therefore does not need to introduce all of the trust assumptions discussed above [29]. However this type of client is still not suitable to run in very constrained environments such as mobile devices, and is still above the bar in terms of such requirements.

Towards the "light client" goal, some systems have adopted additional cryptographic primitives or techniques, for instance succinct proofs to maintain a "compact" representation of the blockchain with fast verification [40].

Based on the above, we can envision an "ideal" light client as a client having very low computation, storage, communication and initial setup requirements

(making such a client feasible even in mobile devices or browsers). However, the light client should retain the security guarantees without introducing additional trust assumptions. Therefore, it still needs to act as the verifier of *efficient* cryptographic proofs, which will convince the client on the received query replies (e.g. on an account's balance or the state of a transaction). These proofs would be created by entities in the blockchain in the *prover* role (e.g. miners or full nodes), ideally without introducing a significant overhead. In Sect. 3.1, we provide informal definitions of the above desired properties that we consider in our work.

1.2 Light Client Implementations in Major Blockchain Systems

We now overview how a light client is perceived and implemented in prominent blockchain systems.

Bitcoin: As discussed previously, the earliest and most well known concept of light client is the Simplified Payment Verification (SPV) client in Bitcoin [86]. An SPV light client only verifies the chain of Proof of Work solutions through the block headers, and requests Merkle proofs on-demand from a full node to verify if a specific transaction is valid (e.g. for transactions that are associated with a wallet address). This approach, while popular even by today's wallets, is not consistent with "decentralization", and introduces additional security assumptions as well as privacy concerns. Satoshi's whitepaper proposed "pruning" as a method to downsize the blockchain (and therefore make it practical for light clients) by discarding spent transaction outputs in each block. However, this method a) requires clients to make a full synchronization even before performing pruning, and b) as of today, it has not been implemented because of security concerns. [4,5,22,28]. We also note some early proposals to store Bitcoin's UTXO set in a Merkle tree for fast bootstrapping [26].

Ethereum: Being an account-based system, Ethereum has the following three types of nodes: a) *full* nodes (most common), which cryptographically verify all account states at all times, but can *prune* account state tries older than 1024 blocks to save space [14], b) *archival* nodes, which always keep the full blockchain history without pruning, and c) *light* nodes which only store block headers to reduce resource requirements. Note that pruning can potentially hurt past transaction or account state querying (and therefore auditability) if there are no archival nodes available to provide a query reply along with a proof (e.g. a Merkle path). Also, Ethereum node software implementations include client and wallet functionalities, therefore the terms clients and nodes are used interchangeably [15].

In contrast with Bitcoin, there is no single node/client software implementation but several different open-source clients written in different programming languages. Geth, written in Go, is the most commonly used [17], and recently introduced a new "snapshot" functionality for full node synchronization in order

to improve read disk access speeds, by including a "flattened" version of all account states as well. However, no Ethereum node/client is light-client friendly even in light mode [1,11]. In practice, considering a Raspberry Pi 4 as a "light client" platform (which is still more powerful compared to mobile devices, especially in terms of energy resources), a geth full node with the new snapshot features can barely run on in, as it still needs a great amount of fast read-write disk storage (i.e. at least 1TB SSD). A geth light node comes without that storage requirement but it still requires a slow, communication-intensive setup phase, which is also required when the node desynchronizes (e.g. in periods of power-off, sleep or disconnections) and is prone to database corruptions.

All Ethereum node types rely on an initial peer discovery algorithm based on the Kademlia Distributed Hash Table (DHT) protocol to connect to other nodes. This is in contrast with Bitcoin core software (the official standard node/client for Bitcoin), which relies on a hardcoded DNS list feed. Lastly, Ethereum plans to implement light clients in its Proof of Stake version (Ethereum 2.0) by introducing "sync committees" to help minimize bootstrapping costs [13], however at the time of writing, details for these committees have not yet been released.

Algorand: Implementing an SPV client in a Proof of Stake blockchain such as Algorand is not straightforward, since block headers are not enough to securely verify the chain [27] (i.e. the client also needs the voters' balances for each block, also discussed in Sect. 4.1). Vault [80], a recent work approach based on Algorand's Proof of Stake protocol, "skips" blocks in each verification step, essentially compressing the block history, while also compressing the voter certificates themselves by using a smaller committee size, but requires a larger percentage of the committee members to vote in order to preserve the validity of the certificate. Vault is discussed in detail in the next section.

Diem: Clients in Diem interact with the blockchain through a full node's JSON-RPC endpoint [9,23], however the client API at the time of writing simply provides answers to queries, without accompanying proofs to provide the client verification capabilities. A client with full verifying functionalities is work in progress [10], and a recent work includes a framework to make client implementations in Diem lightweight [51].

Mina - Coda: Mina inherently supports light clients (full-nodes) through recursive SNARK compositions, which enable maintaining a constant-sized (20KB) blockchain that can be efficiently verified by a client with limited resources. It utilizes a variant of Ouroboros proof-of-stake algorithm to preserve consensus security properties. However Mina, while being light-client oriented, still requires a heavy amount of work for the Block producers, who are in the prover role [20,21,40] (its testnet has a 8-core processor and 16 GB of RAM as minimum requirements).

ZCash: ZIP 221 [16,19] implements Flyclient [45], an efficient block header verification method for light clients. Based on Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [75], it compresses blockchain transaction histories for light clients by only needing to download a small subset of all block headers, which correspond to blocks with higher difficulty target. We discuss both NIPoPoWs and Flyclient in the next section and consider them in our systematization.

Cardano: Although Cardano currently has naive light client implementations that need to place their trust on a full node, it plans to utilize recent work (Mithril) [48] to enable secure and fast boostrapping of light clients in Proof of Stake using a novel primitive, "stake-based threshold multisignatures".

Cosmos - InterBlockchain Communication (IBC): Using the Tendermint BFT Proof of Stake consensus [44], Cosmos' InterBlockchain Communication (IBC) [62] proposes a decentralized protocol for making blockchains communicate with each other, even when these ledgers have fundamentally different underlying architectures. IBC has explicit light client support tailored to its consensus algorithm [42], which only requires to download block headers after a trusted period, which contain sufficient validator signatures proving correctness of validator evolution up to that period. State proofs are then provided to light clients through a full node.

Binance: A light client in Binance chain [24], which uses a Proof of Stake consensus variant (Proof of Authority) [8], is simply implemented by querying a full node, seemingly with a trust model that resembles SPV.

1.3 Related Systematization of Knowledge Works

A recent work [71] provides a taxonomy for cryptocurrency wallets, however its scope is more narrow, focusing on existing wallet implementations (recall the distiction we provided in Sect. 1.1). Still, this work provides some brief insights on (super-)light clients, as well as definitions for the "light" property and its security compared to a full client.

[67] provided a survey on existing blockchain scalability solutions. These include sharding approaches such as OmniLedger [78], layer-2 blockchain protocols [66] or other direct modifications to the blockchain protocol such as increasing block size or replacing the chain structure entirely. At first glance, such scalability solutions might seem related to the light client problem. However, their end goal is different, which is to increase the blockchain's transaction throughput and latency, and not necessary to better support light clients.

2 Cryptographic Building Blocks

In this section we briefly discuss common cryptographic building blocks used by light clients.

2.1 Succinct Set Representation and Proofs

Cryptographic Accumulators enable a succinct and binding representation of a set of elements S and support constant-size proofs of membership (or non-membership) on S. An accumulator typically consists of algorithms to add an element x to it, create a membership proof π that x is contained in the accumulator, verify π, and later update a proof to π' after an element x' has been added to the accumulator. Sub-categories of accumulators are defined if an accumulator manager is needed, if trapdoor information exists and if it supports additional operations like removing elements or creating proofs of non-membership. We point the reader to [35] for formal accumulator definitions and properties. **Merkle Trees** [85] are a specific construction of accumulators, where each element x is represented in a tree of hashes.

Vector commitments [47] enable committing to a vector of elements $[x_i]_{i=1}^{n}$, and later open the commitment at any position i of the vector. While a VC might not be necessarily hiding as a standard commitment, it needs to be *position binding* instead of just binding.

SNARKs (succinct non-interactive arguments of knowledge) are proof systems that are succinct (i.e. have very small proof size compared to that of the statement or the witness) and do not require interaction between the prover and the verifier. zk-SNARKs are a special type of SNARKS augmented with the zero-knowledge property, i.e. constructing a verifiable proof without revealing any information about the witness [88]. In addition, zk-SNARK verification typically requires much less computation than constructing the proof itself. We refer the reader to [65,88] for relevant definitions and sample constructions.

2.2 Hash Functions and Signatures

Aggregate signatures are a special type of digital signatures, where from a set of users U with each user having a signing keypair (pk_u, sk_u) and a subset of signatures $[\sigma_u]$ and corresponding messages $[m_u]$, an aggregator can combine them into a single aggregate signature σ [38,39,74].

Threshold signatures [48,91] enable a subset of k out of n valid signers to generate a signature, but does not allow to create a valid signature with fewer than k of those signers.

Chameleon hashes [79] are collision-resistant hash functions, that have additional properties associated with public-private key pairs compared to standard hash functions. While anyone can compute the output of the chameleon hash function using the public key, the private key serves as trapdoor information to easily find collisions for a specific input.

3 Definitions

3.1 Light Client Properties

Given the plethora of light-client definitions and implementations that exist in the blockchain space, there is a need to unify and standardize their functional,

efficiency and security properties. We informally discuss these properties below, assuming a blockchain B which contains transactions tx and accounts acc, with participating light clients C, consensus participants CN and full nodes N. By B_1 we define the genesis block which we assume that holds all the system parameters and will be used for verifiable bootstrapping.

Functional Properties. As discussed in Sect. 1.1, the system needs to support the following protocols which all run between a client C and a set of full nodes N who always keep B as an input and serve as intermediaries:

- $\mathsf{Init}(\mathsf{B}_1) \rightarrow (\mathsf{st}, \pi)$: The client on input the genesis block B_1, bootstraps/initializes its state st by running an interactive protocol with a full node and receives a proof π of correct initialization.
- $\mathsf{Upd}(\mathsf{st}) \rightarrow (\mathsf{st}', \pi)$: The client updates its state from st to st' to reflect the newest view of B via an interactive protocol with a full node.
- $\mathsf{VrfySt}(\mathsf{st}, \mathsf{st}', \pi) \rightarrow b$: The client verifies π that st' is a correct transition from st (or B_1) and outputs $b \in \{0, 1\}$.
- $\mathsf{Q}(\mathsf{st}, data) \rightarrow (\mathsf{r}, \pi)$: The client makes a query for $data$ where $data = \mathsf{tx}$ (e.g. timestamp or block height) or $data = \mathsf{acc}$ (e.g. an account's address). We also assume that $data$ includes the type of query, i.e. current balance of an account, sender/receiver/value of a transaction, etc. The client receives a reply r and a proof π. If $data \notin \mathsf{B}$, Q typically returns error $\perp$, however *optionally*, it can still provide a proof of non-existence as $(\perp, \pi)$.
- $\mathsf{Vrfy}(\mathsf{st}, r, \pi) \rightarrow b$ Client verifies π for r and outputs $b \in \{0, 1\}$.
- $\mathsf{S}(\mathsf{st}, \mathsf{tx}, \mathsf{acc}, \mathsf{sk}) \rightarrow (\mathsf{st}')$ (optional wallet functionality): Submit a transaction tx to B on behalf of acc with secret information sk.

Security Properties. We list the required security properties that correspond to threats relevant to the operation of the light client.

- *Secure bootstrapping and synchronizing:* This property implies that given a publicly known genesis block B_1, an adversarial full node $\mathcal{A}$ should not be able to convince an honest client C to accept a forged blockchain state B^* (for any B^*) and therefore accept queries on it.

$$\Pr\left[\begin{array}{l}\mathsf{B}_1; \\ \mathcal{A}(\mathsf{B}^*) \text{ and } \mathsf{C} \text{ run } \mathsf{Init}(\mathsf{B}_1), \mathsf{Upd}(\mathsf{st}) : \\ (\mathsf{B}^* \neq \mathsf{B}) \wedge \mathsf{VrfySt}(\mathsf{B}_1, \mathsf{st}, \pi) \rightarrow 1\end{array}\right] \leq negl(\lambda)$$

- *Secure querying:* After bootstrapping, a malicious adversary $\mathcal{A}$ should not be able to convince a light client C to accept a forged transaction or account state. For instance, the adversary should not be able to convince the client that an unverified or forged transaction exists in the blockchain or accept an incorrect account balance. Secure querying also includes the case where $\mathcal{A}$ falsely convinces C the that an accepted transaction or existing account is not part of the blockchain history (i.e. forged proof of non-existence), which

is omitted for brevity from our definition.

$$\Pr\left[\begin{array}{l}\mathsf{B}_1; \\ \mathcal{A}, \mathsf{Init}(), \mathsf{Upd}(), \mathsf{Q}(), \mathsf{S}() : \\ \exists data \notin \mathsf{B} \wedge \mathsf{Q}(data, \mathsf{st}) \rightarrow (\mathsf{r}, \pi) \wedge \mathsf{Vrfy}(r, \pi) \rightarrow 1\end{array}\right] \leq negl(\lambda)$$

Efficiency Properties. We identify the following efficiency properties in terms of storage, computation and communication costs ($|\mathsf{B}|$ denotes blockchain size, or number of blocks). We focus on the operations that happen on the light client side.

- *Efficient bootstrapping and synchronizing:* $\mathsf{Init}()$ and $\mathsf{Upd}()$ computation and communication are sublinear to $|\mathsf{B}|$.
- *Efficient storage:* storage costs (i.e. state size) for light clients, is sublinear to $|\mathsf{B}|$.
- *Efficient communication:* $\mathsf{Q}()$ and $\mathsf{S}()$ (if applicable) require communication costs sublinear to $|\mathsf{B}|$, where communication happens between C and N.
- *Efficient client computation:* $\mathsf{Q}()$ and $\mathsf{S}()$ (if applicable) require client computation costs sublinear to $|\mathsf{B}|$.
- $\mathsf{Vrfy}()$ requires computational costs sublinear to $|\mathsf{B}|$.

Overall, the overhead for B, CN and N in order to support light clients should be minimal compared to the equivalent system that does not provide such support. That said, the full nodes supporting the light clients, might already perform work linear to B.

3.2 Underlying Assumptions

While the variety of light clients operate under different threat models and assumptions depending on the underlying system properties (i.e. PoW or PoS based consensus), we identify a set of common assumptions that we list below.

Basic Light Client Assumptions. To the best of our knowledge, all light client designs implicitly make the following assumptions:

- Trusted genesis block (note that [59] discusses the presence of adversarial pre-computed genesis blocks).
- Reliable consensus (i.e. safety and liveness).
- Secure underlying cryptographic primitives.
- Weak synchrony, i.e. no long network partitions. We do not consider Eclipse network level attacks.

Additional Assumptions. Depending on their design, some systems impose additional assumptions.

- Trusted setup phase for the underlying cryptographic primitives (i.e. zk-SNARKs setup).

- Network-level assumptions: we assume that a client receives and relays information in a peer-to-peer fashion (i.e. distributed networking). This is generally preferred over communicating with a single full node which could act maliciously by relaying a forged view of B to C or prevent it from completing $\mathsf{Init}()$ or $\mathsf{Upd}()$ (i.e. DoS attack).
- Game-theoretic assumptions, i.e. that participants behave in a rational model.
- Special assumptions e.g. fixed Proof of Work difficulty or certain blockchain participants performing specific operations (e.g. accounts needing to restore other accounts not included in the bootstrapped state).

4 Generic Techniques to Build Light Clients

In this section we provide an overview of several generic techniques and protocols that can be used towards designing blockchain light clients and list examples of light client implementations that are based on each technique.

4.1 Header Verification and Consensus Evolution

A common approach when designing bootstrapping and synchronizing for light clients is to only have them verify the block headers and skip verification of transactions or account states (as opposed to standard clients who verify the full blockchain history). This popular technique is adopted by SPV [86], Ethereum [17] and many others.

In Proof of Work consensus, block header verification is straightforward, as the client only needs to verify the proofs of work based on block hashes and nonces. However, additional considerations must be made in Proof of Stake or BFT consensus blockchains to preserve security. For instance, in Proof of Stake, normally the client also needs to verify account states and balances in the whole blockchain history, or consider the risk of long range attacks [6]. In short, the client needs to be convinced that the blockchain consensus has evolved correctly and honestly throughout the history, and no malicious majority was ever present. For BFT-consensus, there is an additional challenge: BFT validators can join and leave, and a client needs to verify the consensus evolution through all validator signatures. A common technique to shorten the client's work is by storing intermediate checkpoints [30] so that clients are not referring to the genesis block each time they verify the current validator set. On the other hand, validator set re-configurations, known as "epochs", present additional considerations as we discuss later in our paper.

4.2 Compressing the State

Being append-only immutable ledgers, the issue of ever-growing storage requirements in blockchains was implied even in the original Bitcoin whitepaper [86], which considered pruning old, spent transaction information (although never

adopted from the community due to security concerns). However, securely pruning "obsolete" data from a blockchain is a direct step towards client efficient bootstrapping and synchronizing as previously discussed in Sect. 3.1. As an example, Ethanos [77] uses a form of "temporary" pruning in the account-based model.

We note that *redacting* is a relevant but stronger notion, with the main goal being to make the blockchain conditionally mutable rather than just reclaiming storage [31,32,61]. This "mutable" blockchain approach mainly relies on the *chameleon hash* primitive discussed in Sect. 2.

As another method of compressing the state, aggregate signatures, such as Schnorr and pairing-based BLS signatures [38,39], can compress many signatures (even under different keys) into a single signature, which in case of BLS, is constant-sized. However in the blockchain setting, aggregate signatures are vulnerable to "rogue key" attacks, where an adversary can produce an aggregated signature for arbitrary public keys, and typically requires a zero-knowledge proof (ZKP) of correct public key computation. Non-interactive EdDSA half aggregation [49] provides ways of compressing multiple Schnorr/EdDSA signatures to a single signature with half the size of the original signatures. One could also consider aggregating signatures using zero knowledge proofs [74]. Overall, aggregate signatures, already used by Plumo [58], is a promising primitive towards light client implementations, as it is estimated to save a significant portion of the needed bandwidth and storage. Another potential option is for the validators to engage into some interactive protocol in advance as part of the consensus committee protocol, using threshold signatures [46,60].

In Appendix A we briefly mention some additional proposals and works whose main goal is to compress the blockchain state. Although these works are not standalone light client implementations, they can serve as examples towards implementing light clients. However, we do not explicitly consider them in our systematization in the later Sections.

4.3 Removing the State

Taking it one step further, *stateless* blockchains aim to only keep a succinct and verifiable representation of the entire state at all times. Compacting a blockchain in this manner is light-client friendly[1], as the bootstrapping and syncing costs would be minimal, and the "stateless" blockchain approach used by Coda-Mina [40], Edrax [54] and others, is also becoming popular. However this can potentially hurt security guarantees, for example the consensus algorithm should be able to securely handle forks, which can happen at any point; there is either a significant share of malicious consensus participants, or simply a network partition. Some works [41] claim that stateless Proof of Stake blockchains are impossible, while others [34,52] introduce special consensus considerations to maintain security.

[1] This approach is sometimes referenced in the literature as "extremely light clients".

Several works point towards the stateless blockchain direction. For instance, Vector Commitments and Subvector Commitments [93] (a special category of Vector commitments), can be used to build a stateless cryptocurrency by committing to key-value maps. Pointproofs [63] further improved this idea by enabling aggregation of individual subvector commitment proofs into a single proof by anyone, as well as cross-commitment proof aggregation (i.e. from multiple subvector commitments) while also ensuring the hiding property (which vector commitments do not necessarily guarantee). Hyperproofs [92] are tree-based data structures that are aggregateable and homomorphic, which are very useful properties for implementing stateless blockchains, and have polynomial commitments [72] as their underlying primitive. Although efficient in their aggregation and update operations, hyperproofs require a trusted setup and have a public parameter size linear to the number of the proofs (i.e. the tree leaves). Finally, SNARKs seem to be a natural tool for implementing stateless or succinct blockchains, while also requiring very low computation for verification; however to be practical, ZKP friendly cryptographic primitives are recommended.

4.4 Leveraging Game-Theoretic Assumptions

In a unique approach as shown by [81], light clients can be built on top of a smart contract interacting with the client and a set of full nodes, thus offloading all blockchain queries and replies to those nodes, with the client themselves performing minimal computational work. In this setting, all participants (i.e. client and full nodes) need to lock funds in an "arbiter" contract as collateral to discourage dishonest behavior. Therefore, rational full nodes are incentivized to provide correct replies to the client's queries or risk being penalized. Such an approach naturally requires a blockchain that is augmented with smart-contract capabilities, but is otherwise agnostic to its other properties.

5 Systematization Methodology

The design of light clients has always been a vibrant topic of discussion in the community. A number of proposals have been given ranging from simple forum or blog posts to rigorous theoretical works and actual deployed systems. In our systematization, we only consider works that represent a distinct light client proposal (i.e. not generic techniques as discussed in Sect. 4), and include at least some form of security discussion. Our systematization is performed over the axes corresponding to the light client properties provided in Sect. 3.1.

In particular, we first consider the **functional and basic operation** axis, where we categorize light client proposals based on their functional properties. These include their compatibility on existing systems (which is preferred), if they require modifications or if they propose a new standalone system. We also note if they are designed for a specific consensus algorithm, and the cryptographic primitives they use. Table 1 shows our findings. We observe that verifiable queries of non-existence are neglected by light client protocols and therefore omitted

from the table. Also note that while clients should always be able to make verifiable queries, wallet functionality is not always included in each one of them. However, we omit a reference to this functionality from our table, as adding it to an existing client protocol or implementation is usually trivial.

The **efficiency** axis, includes several aspects of light client efficiency characteristics, in line with the properties discussed in Sect. 3.1. Note that our systematization is not meant to be used as a direct asymptotic comparison between different light client proposals and protocols. Such a comparison is impossible as the clients operate on top of different underlying schemes. In Table 2, we provide a coarse categorization based on their performance in each efficiency category, indicated with a "good" or "bad" practice icon (thumbs up and down icons respectively). In general, a sublinear cost with respect to the number of blocks is treated as good practice, however, in some cases we deviate from this rule to take concrete costs into account - we mark those with a "*" in the Table. For storage efficiency, we consider both the prover and verifier, where a thumbs up icon denotes good practice for both. Communication efficiency denotes the requirements for proof size, while bootstrapping efficiency denotes the initial cost of client joining the system as well as the syncing maintenance cost.

Finally we consider **security** as the third systematization axis and present our findings in Table 3. We start by listing any required assumptions (i.e. beyond the Basic Assumptions listed in Sect. 3.2) that each light client proposal needs, "-" means that no additional assumptions are made. Then, for each required security property (secure bootstrapping and querying), we indicate whether the light client scheme satisfies the property (✔) or a known vulnerability exists (✘)[2]. In cases where a security guarantee of a light client has not been proven via a security (or sketch) of proof, we denote this by the exclamation mark symbol "!". In Table 3, we also consider the network-level assumption separately, as it is more secure for the light client to communicate with the blockchain in a distributed fashion. Therefore we mark schemes with ✔ that communicate independently (e.g. peer-to-peer) with the blockchain system, while schemes marked with ✘ rely on a centralized server or full node.

In all of our Tables, we group the schemes into two main categories based on their design. The first group follows the "stateless blockchain" approach for constructing efficient light clients, while the second group follows the "efficient bootstrapping - synchronization" approach. We keep the game theoretic-based work as a third separate category.

6 Existing Light Client Constructions: Insights and Gaps

In this section we discuss the works listed in our Tables in more detail, and present a series of interesting insights and gaps. We organize our discussion in a similar way to our scheme grouping for each table, by first analyzing schemes

[2] To mark that a system satisfies a property, we do not necessarily require a formal security proof, but we do require at least some relevant informal discussion.

Table 1. Light client functional properties overview.

System - client	Consensus	Compatibility	Crypto primitives
Mina [40]	PoS	New system	SNARKs
Plumo [58]	BFT	Modification	SNARKs, BLS signatures
PoNW [73]	PoW	New system or Modification	SNARKs
Chen et al. [52]	Not specified	Modification	SNARKs (trusted or universal)
Batched accumulators [37]	Not specified	New system or Modification	Batched RSA accumulator
Edrax [54]	Not specified	New system	Sparse MT, Distributed VC, zkSNARKs
SPV [86]	Any	Yes	
Geth light mode [17]	PoW	Yes	
Vault [80]	PoS	New system	Stamping certificates
Ethanos [77]	PoW	Modification	
NiPoPoW [75]	PoW	Modification	NiPoPoWs [75]
Flyclient [45]	PoW	Modification	MMR commitments
Diem [10]	BFT	Yes	
Cosmos IBC [62]	BFT - PoS	New system	
Binance [24]	PoS variant	New system	
Cardano [48]	PoS	Modification	Stake-based threshold multisignatures
Lu et al. [81]	Any	Yes	

Table 2. Light client efficiency overview.

System - client	Bootstrapping	Storage	Communication	Prover Computation*	Client Computation
Mina [40]	👍	👍	👍	👎	👍
Plumo [58]	👍*	👎 (prover)	👍	👎 (long intervals)	👍
PoNW [73]	👍	👎 (prover)	👍	👍 (embedded in PoW puzzle)	👍
Chen et al. [52]	👍	👍	👍	👎	👍
Batched accumulators [37]	👎	👍	👍	👎	👎
Edrax [54]	👎	👍	👍	👎	👍
SPV [86]	👍*	👍	👍	👍	👍
Geth light mode [17]	👎	👎*	👎*	👍	👍
Vault [80]	👍	👎	👎	👎	👎*
Ethanos [77]	👍*	👎	👎	👍	👍*
NiPoPoW [75]	👍	👍	👍	👍	👍
Flyclient [45]	👍	👍	👍	👎	👍
Diem (verifying) [10]	👍	👍	👍	👍	👍
Cosmos IBC [62]	👍	👍	👍	👎	👍
Binance [24]	👍	👍	👎*	👍	👍
Cardano [48]	👍	👍	👍	👍	👍
Lu et al. [81]	👍	👍	👍	👍	👍

Table 3. Light client schemes security properties.

System - client	Assumptions	Bootstrapping	Querying	Distributed networking
Mina [40]	Trusted setup	✔	✔	✔
Plumo [58]	Trusted setup	✔	✔	✘
PoNW [73]	Trusted setup	!	!	✘
Chen et al. [52]	-	!	!	✘
Batched accumulators [37]	Trusted setup or class groups	✔	✔	!
Edrax [54]	-	✔	✔	!
SPV [86]	-	✘	✘	✘
Geth [17]	-	✔	✔	✔
Vault [80]	Weak synchrony	✔	✔	✔
Ethanos [77]	Active account availability	✔	✔	✔
NiPoPoW [75]	Fixed difficulty	✔	✔	✘
Flyclient [45]	-	✔	✔	✘
Diem [10]	-	✔	✔	✘
Cosmos IBC [62]	-	!	!	✘
Binance [24]	-	!	!	✘
Cardano [48]	-	✔	!	✘
Lu et al. [81]	Rational behavior	!	✔	!

that follow the stateless blockchain approach, then schemes which have efficient bootstrapping and synchronization as their main goal.

6.1 Stateless Blockchains for Light Clients

Here we consider schemes that enable a stateless blockchain design, namely a blockchain with a succinct and verifiable representation of its entire state, as previously discussed in Sect. 4.3.

SNARKs are an effective tool for implementing a stateless blockchain, with Coda-Mina [40] using them in an recursive fashion, chaining them together, eventually having a single SNARK to verify the whole blockchain state. As discussed in Sect. 1.2, it utilizes a variant of the Ouroboros Genesis Proof of Stake algorithm [34] to preserve consensus properties in a stateless setting. Essentially, SNARKS are used as a tool to implement "incremental" verification of recursively-composed proofs, and follow-up works [52,73] improved this paradigm. However, SNARKs typically imply a significant burden on the prover. Plumo [58] uses SNARKs for proving transitions in the consensus committee, enabling fast synchronization of light clients through "checkpoints", thus only needing to fetch data after the most recent checkpoint. These checkpoints also

include periodic proofs of BFT consensus evolution to preserve consensus properties, efficiently verifiable by light clients such as resource-constrained mobile phones. [73] also uses SNARKs and incremental verification, in addition to a Proof-or-Work variant (Proof of Necessary Work) to take advantage of the computation performed by the consensus layer, while Chen et al. [52] in a more extensive study of incremental verification in blockchains, provide a framework to make an existing system incrementally verifiable using a "compatible" consensus algorithm. This work is also the first to provide directions for implementing this paradigm in the context of privacy-preserving blockchains like Zcash [36] by applying incremental verification combined with ZKPs on the public state of the system (which for the case of Zcash is the set of serial numbers and coin commitments). Still, it leaves many questions open, such as which entities will be responsible for providing the proofs, or the overhead on the system which is already not among the most efficient ones.

Gap 1. *Is a complete and efficient light-client scheme possible that is compatible with privacy-preserving systems?*

We should also mention that zk-SNARKs were used in zk-rollups [18]: a layer-2 scalability solution to move data and computation off-chain. However, except for [52], none of the SNARK-based approaches seem to consider the prover's substantial overhead, which in a blockchain system would be the consensus participants or the full nodes. Beyond the prover costs, most SNARK approaches come with additional assumptions such as a trusted setup phase. That leads us to the following Gap:

Gap 2. *Can we design a light-client scheme that satisfies all the security properties while being efficient and practical for the client with a minimal overhead to the consensus participants or full nodes?*

As an intermediate solution, additional financial incentives for entities producing such proofs could alleviate the extra computational requirements, however this is only applicable to blockchains that implement or contain a cryptocurrency.

Improving on the Vector Commitment approach discussed in Sect. 4.3, Boneh et al. [37] introduced techniques for efficiently batching various operations in RSA accumulators (e.g. additions, deletions and witness creation), all of which can potentially utilized for implementing stateless blockchains (e.g. committing to the UTXO set as an accumulator state). RSA accumulators are used by MiniLedger [50] as an alternative model to Merkle trees discussed above. Since RSA Accumulators involve a trusted setup (or novel but more expensive class groups), hash-based accumulators were proposed by [57], however with a different goal, to reduces storage for a fully validating node. An additional concern in the RSA accumulator approach is the extra overhead of maintaining the accumulator (which depending on the implementation, would be paid either by consensus participants or full nodes).

Edrax [54] proposed a cryptocurrency where validators only need to verify a commitment of the most recent state. Edrax implemented this approach in the UTXO model by utilizing sparse Merkle trees to represent the UTXO set, and also in account model by utilizing distributed vector commitments. In the UTXO-based case, validators first verify if a transaction's input belongs in the set, and then simply remove that input and add the output in the set. In the account-based case, they utilize distributed vector commitments to still make transactions possible without requiring interaction between the sender and receiver. However, clients need to constantly synchronize their local proofs with respect to those commitments, and will have to pay a significant synchronization cost after an offline period. Although Edrax proposes an additional untrusted entity to provide synchronization proofs on behalf of the client, this nevertheless introduces a significant overhead overall in the system.

Insight 1. *Redactable blockchains have not been explored as a solution towards implementing light clients.*

Blockchain redaction, discussed in Sect. 4.2, has the potential to be utilized in several ways, for instance, a series of blocks can be replaced by a single block containing compressed information. An interesting direction might be to execute redaction operations at the consensus layer.

6.2 Reducing Bootstrapping and Synchronization Costs

An important property of light clients is the requirement for an efficient way to initialize itself and join the system; downloading gigabytes of data and performing heavy verification operations on millions of transactions is prohibitive for a mobile or browser-based client. This is also important if the client is disconnected for some periods of time and needs to reconnect, or even just to maintain a synchronization with the current state of the blockchain.

Gap 3. *No light client approach or implementation explicitly considers frequent offline phases, where the client needs to re-sync with the current system state.*

As discussed in Sect. 1.2, SPV follows the Header verification approach, which while generally efficient for a light client, suffers from potential security issues (especially in Proof of Stake and BFT consnensus), and relies on the availability and honesty of a small set of servers, while also exposing its privacy to the chosen server(s) from that set [7,12]. Ethereum's native light client also follows this paradigm without relying on a chosen server or full node, however its concrete bootstrapping, storage and communication requirements are practically prohibitive for a light client implementation.

Vault [80] is a prominent example of a standalone system designed for significantly decreasing bootstrapping and participation costs. It is based on Algorand's proof of stake protocol, however it works in an account based model using sparse Merkle trees similar to Ethereum. Vault introduces techniques such

as decoupling double-spend detection from account balances by making transactions valid only for a parameterized block window, while also pruning accounts with no balance, sharding the account state tree across participants, and using additional "stamping" certificates to convince new joining clients on block validity, which have reduced size by trading off liveness while still preserving safety. Although Vault (as a standalone cryptocurrency) was not designed with light clients in mind (e.g. a client needs to constantly perform an update operation while its transaction is pending), its techniques which seem to decrease bootstrapping costs by one or two orders of magnitude, can serve as a guideline for implementing light clients on top of existing systems.

In another approach, Ethanos [77] chooses to reduce the bootstrapping costs on Ethereum by not downloading "inactive" accounts, and invoking a "restore" transaction when such an account needs to reactivate itself. This special transaction type has the inherent limitation of needing to be submitted by another "active" address, and is essentially a Merkle proof of the last known account state (or checkpoint), along with void proofs that no more recent checkpoint exists (paired with a Bloom filter for space efficiency). In this manner, Ethanos reduces bootstrapping costs by a constant factor of 2.

Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) [75] further improve the notion of SPV client by introducing a new primitive under the same name. This primitive, designed for Proof of Work blockchains, constructs a multi-layer chain of blocks from the basic chain, where each layer is essentially a skip list of blocks that satisfy a lower target (i.e. higher difficulty) in the PoW puzzle. In this way, a new client can avoid fetching the entire chain of block headers as in SPV, which translates to logarithmic asymptotic costs (or a few hundred kilobytes proof) making an even more efficient light client. While NIPoPoWs assumed static difficulty across the chain, Flyclient [45] uses an efficiently-updatable Merkle tree variant (Merkle Mountain Range commitments) as underlying primitive for compatibility with variable-difficulty PoW chains.

We also mention some works further improving NIPoPoWs and FlyClient. Kiayias et al. [76] discuss how to securely implement them on top of existing systems through a "velvet" fork, i.e. without requiring a soft or hard fork but only through a minority of the miners. TxChain [96] extends NIPoPoWs and FlyClient to efficiently handle a large number of transaction verifications distributed across several blocks, by introducing a new transaction type ("contingent" transaction), serving as a single reference to other transactions and replacing the need to provide transaction and block inclusion proofs for the skipped blocks (which potentially can be more expensive even than a naive SPV client).

Diem's verifying light client [10] (as discussed in Sect. 1.2) fully relies on a full node to receive query replies and proofs (in contrast, Binance light client [24] which also relies on a full node, does not explicitly verifies any proofs). As Diem utilizes a BFT consensus, it also needs to receive "epoch proofs", which prove to the client correctness of evolution of validator signatures, which is the approach discussed in Sect. 4.1. In addition, recent work [51] suggest to further compress epoch proofs by an *epoch skipping* technique, without however addressing long

range attacks. Also as discussed in Sect. 1.2, Tendermint [42] (used in Cosmos IBC) proposes a similar technique based on the latest block height which ensures that at least one validator is honest based on validator intersection and the byzantine threshold. Plumo's proofs of BFT consensus evolution [58] also aim to reduce client synchronization load as discussed previously.

Insight 2. *Light clients in BFT-based consensus blockchains can be implemented through full nodes, where clients make queries and full nodes provide verifiable proofs alongside with epoch proofs.*

Insight 3. *In BFT-based consensus blockchains, aggregate signatures (e.g. BLS signatures or ZK-friendly signatures) can be used to compress not only transactions, but also validator signatures, leading to further reduced bootstrapping and synchronization costs for light clients.*

An alternative approach to Diem and Plumo is used by Dfinity's Internet computer [68], a blockchain-based protocol that creates a network of decentralized data centers running smart contracts, inspired by Ethereum. Dfinity utilizes key re-sharing within a threshold signature scheme to accommodate validators joining or leaving, aiming at circumventing the need for tracking their key evolution by a client [64]. However, it is unclear whether this approach guarantees BFT security at all times, as it assumes that validators will delete their old shares afterwards. For instance, suppose the consensus system has 7 honest validators from a quorum of 10 validators, which guarantees the $2f + 1$ consensus security properties. Still, if 12 validators join afterwards, which now implies a tolerance of 7 Byzantine validators, this can potentially compromise consensus, as the previous 7 "honest" validators might not have deleted their key shares. In addition, Aumasson and Shlomovits [33] highlighted the possibility of an adversary corrupting the key re-sharing process in some threshold signature schemes, which could potentially hurt consensus liveness.

Insight 4. *For blockchains based on BFT consensus, frequent validator reconfigurations (e.g. joining, leaving or key rotations) usually imply additional work for clients.*

While the insight above is not applicable to off-chain reconfiguration approaches such as Dfinity [64], such approaches are typically prone to long range attacks as we discussed previously.

Gap 4. *A light client of a BFT-based consensus blockchain normally needs to verify the evolution of validator signatures using "epoch proofs" to prevent long range attacks. Is it possible to design a secure protocol for BFT consensus that either compress these proofs or circumvents this requirement entirely?*

More recently, Chaidos and Kiayias [48] proposed a new primitive, called stake-based threshold multisignatures. This primitive enables a client's bootstrapping through header verification in Proof of Stake systems like Cardano, in a similar way to SPV, without however needing to verify the participant's stake history (as discussed in Sect. 4.1) and without the need of any modifications to the Proof of Stake consensus as in Mina [40].

6.3 Smart-Contract Based Approaches and Blockchain Interoperability

We briefly discuss implementing light clients by querying full nodes though a smart contract, and assuming "rational" behavior from the client and the full nodes after the required collateral deposits to participate, similar to the work by Lu et al. [81]. This approach can potentially address many of the previously discussed gaps, as the rational behavior assumption can circumvent technical difficulties or limitations which rise from complex cryptographic primitives. For instance, as [81] showed, a light client can make a query of non-existence, and assuming full node rational behavior, will get a correct reply (i.e. inclusion proof if queried data exists, or a negative reply in case such data does not exist, which can be challenged if another node presents an inclusion proof thus penalizing a false non-existence claim). However there are several caveats to such an approach: First it naturally requires a smart-contract, which implies a time delay until it received the reply to its query, incompatibility with blockchains without a smart contract, and additional monetary costs for the contract's "gas" fees which can be potentially very high. Also, the client might merely receive an answer to its query (e.g. a simple "∄" reply if answer to query does not exist) without a cryptographic proof (as defined in Sect. 3.1 as an optional functional property), which leaves this problem still open. Finally, the game-theoretic model might not capture cases where the client is considered a "high value target", where a full node (or a coalition of them) might choose to actually behave "irrationally" and intentionally risk being penalized in hope for other (not necessary monetary) gains.

Gap 5. *Can we design a light client protocol compatible with queries of transaction or account state non-existence proofs?*

From the above approach we observe however that it is trivial to implement an efficient light client that makes and receives queries to a "trusted oracle" (which in the above case were the rational full nodes following the protocol), without needing to make verifications, even if such an oracle is decentralized. This implies that such a client would be possible to exist even in extremely resource-constrained environments such as a smart contract *itself*:

Insight 5. *Interoperability: Ideally, light clients should be implemented as a smart contract without the use of trusted oracles. This would allow for verifying transactions of a blockchain A inside a contract of blockchain B.*

Gap 6. *Implementing reasonably efficient light clients inside smart contracts might be impractical for many non zero-knowledge proof friendly blockchains or ledgers without succinct fraud proof in optimistic settings [18].*

Although Cosmos makes a first step towards building a light client compatible with several blockchain systems (including those with smart contracts), it is still not known if we can also utilize previous techniques or primitives to implement such clients in pure smart-contract based blockchains, e.g. Ethereum.

7 Conclusion

The blockchain community is witnessing a continuous effort towards implementing efficient light clients, suitable for resource-constrained devices or environments like browsers or mobile phones, while maintaining the underlying blockchain's security guarantees, and without introducing additional trust assumptions. As we observe different perceptions of light client properties across blockchain systems, we first provide a categorization of the most important light client properties. Then, we present a systematization of proposed light clients across three axes derived from our property categorization. Our systemization helps to identify a number of exciting open problems on implementing light clients which we summarize below.

We first observe that light clients satisfying our properties, and compatible with privacy preserving systems have not yet been implemented (Gap 1), with recent works providing preliminary directions [52]. In addition, no current scheme seems to satisfy all of our functional, efficiency and security properties together (Gap 2). Also, existing works seem to neglect the case of frequent light client offline phases, which might be inefficient even for clients with efficient bootstrapping protocols (Gap 3). Distributing prover's work among the main blockchain participants (consensus layer or full nodes) along with providing incentives are possible directions.

Furthermore, it is not yet known if light clients can be efficient enough, such that they can be run from smart contracts across different blockchains (Gap 6). SNARKs seem to be a promising primitive towards this, although this still need to be shown in practice. Also there seems to be room for improvement for light clients implemented on BFT-consensus blockchains (Gap 4) by leveraging primitives such as key re-sharing and threshold signatures in off-chain protocols, while however considering Byzantine nodes in special cases. Finally, proofs of non-existence, a desired property in blockchain systems, is still missing from all current light client implementations (Gap 5). We hope our work will provide research directions for the community towards usable and secure light clients for blockchain systems.

Acknowledgements. Foteini Baldimtsi and Panagiotis Chatzigiannis were supported by NSF #1717067, NSA #204761 and a Facebook Research Award. Panagiotis Chatzigiannis was partially supported by Harmony through the Research DAO. The authors would like to thank Matthew Zipkin for the constructive feedback.

A Towards the Light Client Goal

A number of works and proposals exist towards improving efficiency in state representation. Merkle trees were initially proposed to store Bitcoin's UTXO set (which represents the blockchain state) for fast bootstrapping [26], with a $O(\lg n)$ algorithm for updating and re-balancing the tree across blocks (i.e. updating values, insertions and deletions of accounts). Then [89] further optimized the re-balancing algorithm using AVL trees. MiniLedger [50] also used Merkle trees

to represent the history of transactions per participant. Meanwhile, Ethereum used tries as a more efficient method to represent the account-balance state [95].

In addition, Karakostas et al. [70] proposed a modification of storing the UTXO set which represents the blockchain state in UTXO-model cryptocurrencies by incentivizing constructing "state-friendly" transactions, while [94] proposes a modification on Bitcoin to represent transactions with a trie-based authenticated data structure to enable efficient membership and non-membership proofs. Stateless clients have also been considered in Ethereum using asynchronous accumulators [25,90].

Aiming exclusively for faster client bootstrapping, [2] suggested to distribute the state through external file sharing protocols (e.g. Bittorrent). Then [53] proposed a modification designed for Proof-of-Work blockchains that stores a constant number of state snapshots, in a similar fashion to Ethereum. Similarly, [83] proposes a state-based synchronization based on Bitcoin (i.e. snapshot-based approach), forming a side-chain linked to the main chain, and claiming to reduce blockchain size by 93%. Which however required modifications to Bitcoin, since blocks with invalid attached states should be rejected.

Works that include blockchain pruning include [43], which replaces a UTXO set with an account tree that is cryptographically tied to each mined block, and [87], which proposes a pruning algorithm for permissioned blockchains, executed by each participant separately, using predicate functions to remove spent transactions. Matzutt et al. [84] proposed a pruning scheme for Bitcoin that makes snapshots of the Bitcoin state for efficient bootstrapping of new clients, and also includes a qualitative comparison of related work to pruning and efficient bootstrapping. In addition, Corda [69] can aggregate (and then prune) previous transactions into a single new, reissued transaction.

In the context of blockchain redaction, in addition to preliminary works as [32], we mention [82] designed for "execute-order-validate blockchains" such as Hyperledger Fabric, however with a goal to improve privacy rather efficiency. Also [56] and [55] consider "policy-based" blockchain redaction, which can also serve as a useful tool towards light client implementations.

References

1. Ask about geth: snapshot acceleration. https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration/
2. Bitcoin blockchain data torrent. https://bitcointalk.org/index.php?topic=145386.0
3. Bitcoin core client. https://bitcoin.org/en/bitcoin-core/
4. Bitcoin wiki - clients. https://en.bitcoin.it/wiki/Clients
5. Bitcoin wiki - scalability. https://en.bitcoin.it/wiki/Scalability#Simplified_payment_verification
6. Blockchain light client. https://medium.com/codechain/blockchain-light-client-1171dfa1269a
7. Breadwallet SPV bitcoin C library. https://github.com/breadwallet/breadwallet-core

8. Consensus engine of binance smart chain. https://docs.binance.org/smart-chain/guides/concepts/consensus.html
9. Diem client SDKs. https://github.com/diem/client-sdks
10. Diem verifying client. https://github.com/diem/diem/blob/main/sdk/client/src/verifying_client.rs
11. Dodging a bullet: Ethereum state problems. https://blog.ethereum.org/2021/05/18/eth_state_problems/
12. Electrum docs - frequently asked questions. https://electrum.readthedocs.io/en/latest/faq.html
13. Eth 2.0 specs - minimal light client. https://github.com/ethereum/eth2.0-specs/blob/dev/specs/altair/sync-protocol.md
14. The ethereum-blockchain size will not exceed 1TB anytime soon. https://dev.to/5chdn/the-ethereum-blockchain-size-will-not-exceed-1tb-anytime-soon-58a
15. Ethereum nodes and clients. https://ethereum.org/en/developers/docs/nodes-and-clients/
16. Explaining flyclient. https://electriccoin.co/blog/explaining-flyclient/
17. How to run a light node with geth. https://ethereum.org/en/developers/tutorials/run-light-node-geth/
18. An incomplete guide to rollups. https://vitalik.ca/general/2021/01/05/rollup.html
19. Introducing heartwood. https://electriccoin.co/blog/introducing-heartwood/
20. Mina documentation. https://docs.minaprotocol.com/en
21. Mina protocol - a succinct blockchain. https://masked.medium.com/the-coda-protocol-bbcb4b212b13
22. Nakamoto: a new bitcoin light-client. https://cloudhead.io/nakamoto/
23. The official diem client SDK for python. https://github.com/diem/client-sdk-python
24. Run a light client to join binance chain. https://docs.binance.org/light-client.html
25. The stateless client concept. https://ethresear.ch/t/the-stateless-client-concept/172
26. Storing UTXOs in a balanced Merkle tree. https://bitcointalk.org/index.php?topic=101734.msg1117428
27. A suggestion for a light-client wallet (like the BTC SPV wallet with Merkle tree). https://forum.algorand.org/t/a-suggestion-for-a-light-client-wallet-like-the-btc-spv-wallet-with-merkle-tree/1092/4
28. Ultimate blockchain compression w/ trust-free lite nodes. https://bitcointalk.org/index.php?topic=88208.0/
29. What is a light client and why you should care? https://www.parity.io/blog/what-is-a-light-client/
30. Amsden, Z., et al.: The libra blockchain (2019). https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf
31. Ashritha, K., Sindhu, M., Lakshmy, K.: Redactable blockchain using enhanced chameleon hash function. In: 2019 5th International Conference on Advanced Computing Communication Systems (ICACCS), pp. 323–328 (2019). https://doi.org/10.1109/ICACCS.2019.8728524
32. Ateniese, G., Magri, B., Venturi, D., Andrade, E.R.: Redactable blockchain - or - rewriting history in bitcoin and friends. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, 26–28 April 2017, pp. 111–126. IEEE (2017). https://doi.org/10.1109/EuroSP.2017.37
33. Aumasson, J.P., Shlomovits, O.: Attacking threshold wallets. Cryptology ePrint Archive, Report 2020/1052 (2020). https://eprint.iacr.org/2020/1052

34. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: composable proof-of-stake blockchains with dynamic availability. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, pp. 913–930. ACM Press, October 2018. https://doi.org/10.1145/3243734.3243848
35. Baldimtsi, F., et al.: Accumulators with applications to anonymity-preserving revocation. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, 26–28 April 2017, pp. 301–315. IEEE (2017). https://doi.org/10.1109/EuroSP.2017.13
36. Ben-Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, May 2014. https://doi.org/10.1109/SP.2014.36
37. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 561–586. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_20
38. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 416–432. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_26
39. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45682-1_30
40. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352 (2020). https://eprint.iacr.org/2020/352
41. Bonnet, F., Bramas, Q., Défago, X.: Stateless distributed ledgers. In: Georgiou, C., Majumdar, R. (eds.) NETYS 2020. LNCS, vol. 12129, pp. 349–354. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67087-0_22
42. Braithwaite, S., et al.: A tendermint light client. CoRR abs/2010.07031 (2020). https://arxiv.org/abs/2010.07031
43. Bruce, J.: The mini-blockchain scheme (2017). https://cryptonite.info/files/mbc-scheme-rev3.pdf
44. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. CoRR abs/1807.04938 (2018). https://arxiv.org/abs/1807.04938
45. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: FlyClient: super-light clients for cryptocurrencies. In: 2020 IEEE Symposium on Security and Privacy, pp. 928–946. IEEE Computer Society Press, May 2020. https://doi.org/10.1109/SP40000.2020.00049
46. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1769–1787 (2020)
47. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_5
48. Chaidos, P., Kiayias, A.: Mithril: stake-based threshold multisignatures. Cryptology ePrint Archive, Report 2021/916 (2021). https://ia.cr/2021/916
49. Chalkias, K., Garillot, F., Kondi, Y., Nikolaenko, V.: Non-interactive half-aggregation of EdDSA and variants of schnorr signatures. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 577–608. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75539-3_24

50. Chatzigiannis, P., Baldimtsi, F.: Miniledger: compact-sized anonymous and auditable distributed payments. Cryptology ePrint Archive, Report 2021/869 (2021). https://eprint.iacr.org/2021/869
51. Chatzigiannis, P., Chalkias, K.: Proof of assets in the diem blockchain. Cryptology ePrint Archive, Report 2021/598 (2021). https://eprint.iacr.org/2021/598
52. Chen, W., Chiesa, A., Dauterman, E., Ward, N.P.: Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Report 2020/1522 (2020). https://ia.cr/2020/1522
53. Chepurnoy, A., Larangeira, M., Ojiganov, A.: Rollerchain, a blockchain with safely pruneable full blocks (2016)
54. Chepurnoy, A., Papamanthou, C., Zhang, Y.: Edrax: a cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968 (2018). https://eprint.iacr.org/2018/968
55. Derler, D., Samelin, K., Slamanig, D., Striecks, C.: Fine-grained and controlled rewriting in blockchains: chameleon-hashing gone attribute-based. In: NDSS 2019. The Internet Society, February 2019
56. Deuber, D., Magri, B., Thyagarajan, S.A.K.: Redactable blockchain in the permissionless setting. In: 2019 IEEE Symposium on Security and Privacy, pp. 124–138. IEEE Computer Society Press, May 2019. https://doi.org/10.1109/SP.2019.00039
57. Dryja, T.: Utreexo: a dynamic hash-based accumulator optimized for the bitcoin UTXO set. Cryptology ePrint Archive, Report 2019/611 (2019). https://eprint.iacr.org/2019/611
58. Gabizon, A., et al.: Plumo: towards scalable interoperable blockchains using ultra light validation systems (2020)
59. Garay, J.A., Kiayias, A., Leonardos, N., Panagiotakos, G.: Bootstrapping the blockchain, with applications to consensus and fast PKI setup. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part II. LNCS, vol. 10770, pp. 465–495. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-76581-5_16
60. Garillot, F., Kondi, Y., Mohassel, P., Nikolaenko, V.: Threshold schnorr with stateless deterministic signing from standard assumptions. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12825, pp. 127–156. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84242-0_6
61. Gligor, V.D., Woo, S.L.M.: Establishing software root of trust unconditionally. In: NDSS 2019. The Internet Society, February 2019
62. Goes, C.: The interblockchain communication protocol: an overview. CoRR abs/2006.15918 (2020). https://arxiv.org/abs/2006.15918
63. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: aggregating proofs for multiple vector commitments. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020, pp. 2007–2023. ACM Press, November 2020. https://doi.org/10.1145/3372297.3417244
64. Groth, J.: Introducing noninteractive distributed key generation. https://medium.com/dfinity/applied-crypto-one-public-key-for-the-internet-computer-ni-dkg-4af800db869d
65. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
66. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: SoK: layer-two blockchain protocols. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 201–226. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_12

67. Hafid, A., Hafid, A.S., Samih, M.: Scaling blockchains: a comprehensive survey. IEEE Access **8**, 125244–125262 (2020). https://doi.org/10.1109/ACCESS.2020.3007251
68. Hanke, T., Movahedi, M., Williams, D.: Dfinity technology overview series, consensus system (2018)
69. Hearn, M., Brown, R.G.: Corda: a distributed ledger (2019). https://www.corda.net/wp-content/uploads/2019/08/corda-technical-whitepaper-August-29-2019.pdf
70. Karakostas, D., Karayannidis, N., Kiayias, A.: Efficient state management in distributed ledgers. Cryptology ePrint Archive, Report 2021/183 (2021). https://eprint.iacr.org/2021/183
71. Karantias, K.: SoK: a taxonomy of cryptocurrency wallets. Cryptology ePrint Archive, Report 2020/868 (2020). https://eprint.iacr.org/2020/868
72. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_11
73. Kattis, A., Bonneau, J.: Proof of necessary work: succinct state verification with fairness guarantees. Cryptology ePrint Archive, Report 2020/190 (2020). https://eprint.iacr.org/2020/190
74. Khaburzaniya, I., Chalkias, K., Lewi, K., Malvai, H.: Aggregating hash-based signatures using starks. Cryptology ePrint Archive, Report 2021/1048 (2021). https://ia.cr/2021/1048
75. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 505–522. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_27
76. Kiayias, A., Polydouri, A., Zindros, D.: The velvet path to superlight blockchain clients. Cryptology ePrint Archive, Report 2020/1122 (2020). https://eprint.iacr.org/2020/1122
77. Kim, J., Lee, J., Koo, Y., Park, S., Moon, S.: Ethanos: efficient bootstrapping for full nodes on account-based blockchain. In: Barbalace, A., Bhatotia, P., Alvisi, L., Cadar, C. (eds.) EuroSys 2021: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, 26–28 April 2021, pp. 99–113. ACM (2021). https://doi.org/10.1145/3447786.3456231
78. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: a secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy, pp. 583–598. IEEE Computer Society Press, May 2018. https://doi.org/10.1109/SP.2018.000-5
79. Krawczyk, H., Rabin, T.: Chameleon hashing and signatures. Cryptology ePrint Archive, Report 1998/010 (1998). https://eprint.iacr.org/1998/010
80. Leung, D., Suhl, A., Gilad, Y., Zeldovich, N.: Vault: fast bootstrapping for the algorand cryptocurrency. In: NDSS 2019. The Internet Society, February 2019
81. Lu, Y., Tang, Q., Wang, G.: Generic superlight client for permissionless blockchains. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020, Part II. LNCS, vol. 12309, pp. 713–733. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59013-0_35
82. Manevich, Y., Barger, A., Assa, G.: Redacting transactions from execute-order-validate blockchains. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2021, Sydney, Australia, 3–6 May 2021, pp. 1–9. IEEE (2021). https://doi.org/10.1109/ICBC51069.2021.9461093

83. Marsalek, A., Zefferer, T., Fasllija, E., Ziegler, D.: Tackling data inefficiency: compressing the bitcoin blockchain. In: 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), pp. 626–633 (2019). https://doi.org/10.1109/TrustCom/BigDataSE.2019.00089
84. Matzutt, R., Kalde, B., Pennekamp, J., Drichel, A., Henze, M., Wehrle, K.: How to securely prune bitcoin's blockchain. In: 2020 IFIP Networking Conference, Networking 2020, Paris, France, 22–26 June 2020, pp. 298–306. IEEE (2020). https://ieeexplore.ieee.org/document/9142720
85. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
86. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2009). https://bitcoin.org/bitcoin.pdf
87. Palm, E., Schelén, O., Bodin, U.: Selective blockchain transaction pruning and state derivability. In: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, 20–22 June 2018, pp. 31–40. IEEE (2018). https://doi.org/10.1109/CVCBT.2018.00009
88. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society Press, May 2013. https://doi.org/10.1109/SP.2013.47
89. Reyzin, L., Meshkov, D., Chepurnoy, A., Ivanov, S.: Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 376–392. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_21
90. Reyzin, L., Yakoubov, S.: Efficient asynchronous accumulators for distributed PKI. In: Zikas, V., De Prisco, R. (eds.) SCN 2016. LNCS, vol. 9841, pp. 292–309. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44618-9_16
91. Shoup, V.: Practical threshold signatures. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 207–220. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_15
92. Srinivasan, S., Chepurnoy, A., Papamanthou, C., Tomescu, A., Zhang, Y.: Hyperproofs: aggregating and maintaining proofs in vector commitments. Cryptology ePrint Archive, Report 2021/599 (2021). https://eprint.iacr.org/2021/599
93. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 45–64. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_3
94. White, B.: A theory for lightweight cryptocurrency ledgers (2015). https://raw.githubusercontent.com/input-output-hk/qeditas-ledgertheory/master/lightcrypto.pdf
95. Wood, G.: Ethereum: A secure decentralized generalised transaction ledger (2021). https://ethereum.github.io/yellowpaper/paper.pdf. Accessed 14 Feb 2021
96. Zamyatin, A., Avarikioti, Z., Perez, D., Knottenbelt, W.J.: TxChain: efficient cryptocurrency light clients via contingent transaction aggregation. Cryptology ePrint Archive, Report 2020/580 (2020). https://eprint.iacr.org/2020/580