CoVer: Collaborative Light-Node-Only Verification and Data Availability for Blockchains

Steven Cao, Swanand Kadhe, Kannan Ramchandran
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{*stevencao, swanand.kadhe, kannanr*}*@berkeley.edu*

***Abstract*—Validating a blockchain incurs heavy computation, communication, and storage costs. As a result, clients with limited resources, called *light nodes*, cannot verify transactions independently and must trust *full nodes*, making them vulnerable to security attacks. Motivated by this problem, we ask a fundamental question: can light nodes securely validate without any full nodes? We answer affirmatively by proposing CoVer, a decentralized protocol that allows a group of light nodes to collaboratively verify blocks even under a dishonest majority, achieving the same level of security for block validation as full nodes while only requiring a fraction of the work. In particular, work per node scales down proportionally with the number of participants (up to a log factor), resulting in computation, communication, and storage requirements that are *sublinear in block size*. Our main contributions are light-node-only protocols for fraud proofs and data availability.**

1. Introduction

1.1. Motivation

Blockchain participants that independently validate and store all the blocks are called *full nodes* and are vital to the security of the network. However, as adoption of cryptocurrencies grows, the burden of running a full node will become infeasible for most users due to higher throughput, causing more participants to operate as *light nodes*. Light nodes or light clients, based on the simplified payment verification (SPV) protocol proposed by Nakamoto [1], store only block headers and do not validate transactions. As a result, they cannot contribute to the security of the network and are vulnerable to several security and privacy attacks (see, e.g., [2, Chapter 6]). In particular, light nodes accept the longest header chain, assuming that any *invalid* chain, or a chain containing fraudulent transactions, will not be mined upon. Therefore, in the status quo, their security depends on the longest chain being valid, requiring the strong assumption that the majority of miners are honest.

In [3], Al-Bassam et al. propose protocols for *fraud proofs* and *data availability*, allowing light nodes to reject headers of invalid blocks by receiving compact proofs of their invalidity from full nodes. While block proposal protocols still require an honest majority (e.g. to prevent forking-based double spending attacks), they solve the verification problem with an honest minority of full nodes. However, the protocols still require each light node to be in communication with an honest full node, increasing the burden on full nodes and limiting the protocol's scalability.

Motivated by these challenges, we ask a fundamental question: *can light nodes securely validate without any full nodes?* Formally, we characterize a light node as one that bears computation, communication, and storage costs *sublinear in block size*. We answer affirmatively by proposing a decentralized and permissionless light-node-only **Co**llaborative **Ver**ification protocol, called **CoVer**, which enables a group of light nodes to collaboratively validate blocks even under a dishonest majority, resulting in the same level of security as full nodes.[1] Furthermore, the protocol is flexible in that participants can perform as much or as little validation as they wish, with more active participants contributing more to the security of the protocol. The result is a blockchain network that accommodates and utilizes the computing resources of participants of all sizes.

1.2. Overview

The key ingredients in CoVer are light-node-only protocols for fraud proofs and data availability (see Fig. 1 for an overview). At a high level, each node verifies a small part of each block and broadcasts a *fraud proof*, i.e. a proof of block invalidity, for any invalid transaction. A node rejects the block if it receives a valid fraud proof and accepts it otherwise. Then, the protocol is secure as long as each transaction is validated by at least one honest participant.

To realize this idea, we propose protocols to solve the following three challenges. The first challenge is that light nodes should be able to validate individual transactions without needing to process the whole block or store the entire state. We overcome this challenge by proposing a *new block structure and fraud proof protocol* (Section 2).

This work is supported in part by the National Science Foundation grant SaTC-1937357.

1. Like [3] and [4], we restrict our attention to block verification and not block proposal. Forking-based double spending attacks are still possible with a dishonest majority of miners.

978-0-7381-0495-9/20/$31.00 ©2020 IEEE
DOI 10.1109/Blockchain50366.2020.00014

The second challenge is to guarantee *data availability* [3]: if a miner includes an invalid transaction in the Merkle root of the block but does not make this transaction available for download, then no node can produce a fraud proof for it. Therefore, light nodes should accept a header only if all of its transactions are available, and they must check availability while only downloading a small amount of data. To solve this problem, the protocol in [3] requires miners to encode the block data using an erasure code. Then, to hide any single transaction, a miner must prevent decoding by either (1) hiding at least a constant fraction of the block, in which case a light node can catch unavailability with high probability by randomly sampling a constant number of shares, or (2) constructing the coded data incorrectly, in which case full nodes can produce proofs of coding fraud. The bottleneck in this approach is decoding the block, which is at least linear in block size and therefore infeasible for light nodes. We address this challenge by proposing a *secure collaborative decoding protocol* on top of the recently proposed LDPC-coded Merkle tree [4] (Section 3).

The third challenge is that light nodes cannot download the entire block. To address this challenge, we propose and analyze a simple modification of the gossip protocol, deemed *selective broadcast*, in which nodes only gossip the data that they need (Section 4).

Together, these protocols empower a group of light nodes to collaboratively validate a block with each node performing only a small amount of *work*, i.e. computation, communication, and storage. Specifically, if each block has L transactions and there are N_h honest light nodes, each light node performs $\tilde{O}(L/N_h)$ work, where $\tilde{O}$ denotes equivalence up to log factors. Due to the connectivity required for the selective broadcast communication scheme, this savings factor is capped at $\sqrt{L}$, resulting in $\tilde{O}(\sqrt{L})$ work per node for large validation pools.[2] These protocols are also decentralized and permissionless while requiring only an honest minority. Table 1 summarizes the properties of CoVer while comparing it to other approaches.

1.3. Problem Setup and Objectives

System Model: We consider two types of nodes:

1) *Miners*: nodes that produce and broadcast new blocks.
2) *Validators*: nodes that validate new blocks. We also refer to validators as *participants*.

We assume that all participants are *light nodes*. Specifically, we model a light node as a node having computation, communication, and storage capabilities sublinear in the size of each block. We note, however, that nodes with higher capacity can also participate in the protocol. For example, a node with twice the capability of a normal light node can act as two light nodes and perform twice the work.

We assume bounded network delay such that if a node sends a message, it can be received by all other nodes connected to the sender within some maximum delay Δ. For

2. This ceiling can be bypassed by alternate communication schemes that may be possible given stronger assumptions.

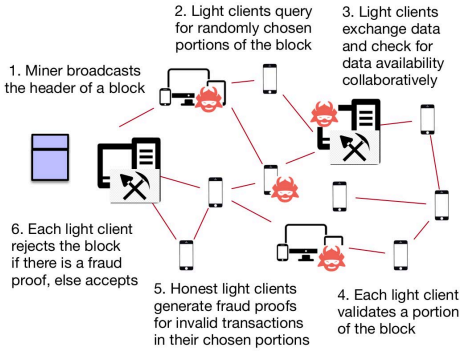

Figure 1. An overview of CoVer. We consider a network with miners, i.e. block producers, and light node validators, which perform computation, communication and storage sublinear in block size. The key ingredients are light-node-only protocols for data availability and fraud proofs.

analysis, we model the connectivity between honest nodes as an Erdős-Rényi random graph, where each pair of nodes is connected independently with some probability p [5]. This connection probability is a parameter of the protocol, and we analyze the required connectivity in Section 5. We make no assumptions on the connectivity of dishonest nodes.

Threat Model: We assume that any nodes can be dishonest and act adversarially. Dishonest nodes may deviate from the protocol in arbitrary manner and may collude with each other. Further, we assume that dishonest nodes can perform any polynomial-time computations but cannot invert hash functions. We require an honest minority, meaning that for the protocol to function, there must be honest participants doing enough work such that every transaction in a block is validated by at least one participant. The required minimum number is analyzed in Section 5. Like [3], we only solve verification. Forking-based double spending attacks are still possible with a dishonest majority of miners.

Objectives: In every round, a miner produces a block and broadcasts the block along with its header. A malicious miner may only broadcast subset of its data. Then, each participant executes the protocol and chooses to either accept or reject the block. The protocol should be

1) *Correct*: all participants accept the block if and only if
   a) The block is *valid* (contains only valid transactions).
   b) The block is *available*, meaning that the miner broadcasted enough of the block data such that all of the data can be decoded (see Section 3).
2) *Light-node-only*: each participant performs computation, communication, and storage sublinear in the number of transactions in the incoming block.
3) *Permissionless*: nodes can freely join and leave.
4) *Decentralized*: the requirements of the protocol should be uniform over all participants.
5) *Secure under honest minority*: for block verification, the number of honest participants required for the protocol to be correct should not depend on the number of total participants.

We require correctness only with high probability, rather

TABLE 1. SUMMARY OF APPROACHES FOR LIGHT NODE SECURITY

	Full node computation	Full node communication	Light client computation	Light client communication	Light client storage	Block size	Light client security dependence
Status quo [1]	$O(L)$	$O(L)$	O(1)	O(1)	$O(T)$	$O(L)$	Miners (honest majority)
Fraud proofs [3] + 2D Reed Solomon [3]$^{\alpha}$	$O(L \log A)$ + $O(L^{1.5})$	$O(L)$	$O(\log(AL))$ + $O(\sqrt{L} \log L)$	$O(\log(AL))$ + $O(\sqrt{L} \log L)$	$O(T)$ + $O(\sqrt{L}T)$	$O(L)$	Full nodes (honest minority)
Fraud proofs [3] + Coded Merkle Tree [4]	$O(L \log A)$	$O(L)$	$O(\log(AL))$	$O(\log(AL))$	$O(T)$	$O(L)$	Full nodes (honest minority)
CoVer (this work)	N/A	N/A	$O\left(\frac{1}{k} L \log L\right)$	$O(k \log N_h)$	$O\left(T + \frac{1}{k} L \log L\right)$	$O(L \log L)$	Each other (honest minority)

L: transactions per block, N_h: number of honest participants, A: total number of accounts, T: number of rounds, k: division of work parameter, capped at $\frac{\log N_h}{N_h}$
We compare to the original fraud proofs with two versions of data availability, 2D Reed Solomon [3] and the coded Merkle tree [4].
$^{\alpha}$For fraud proofs + 2D Reed Solomon, we split the costs into those due to validation of transactions (upper row) and data availability (lower row).

than in the worst case. We will use λ to denote a security parameter such that correctness holds with probability at least $(1 - e^{-\lambda})$. The specific high probability expressions are calculated in the correctness proofs (Section 5).

2. Collaborative Verification

2.1. Proposed Block Structure

To enable fraud proofs, Al-Bassam et al. [3] modify the block to include the root of a sparse Merkle tree that contains the state (UTXOs or account balances). However, this approach does not accommodate light clients because the state cannot be divided. Therefore, we propose an alternate block structure and fraud proof protocol. First, as usual, the header of the i-th block, denoted $\texttt{header}(i)$, contains:

- `prevHash`: hash of the previous block.
- `root`: root of the Merkle tree containing transactions.
- `len`: number of transactions.
- `other`: other data (like the nonce for proof-of-work).

We will denote the Merkle proof of the transaction `txn` in block i as $\texttt{proof}(\texttt{txn} \rightarrow \texttt{header}(i).\texttt{root})$. Next, as usual, each transaction `txn` in block i contains:

- `txid`: transaction id (e.g., 32 byte transaction hash)
- `sender`: the id (e.g., public key) of the sender.
- `signature`: the sender's signature.
- `outputs`: a list of transaction outputs, numbered $1, 2, ...$, which specify the recipient and the amount.

To enable fraud proofs (see Section 2.2), we require that each transaction reference and provide Merkle proofs for the UTXOs (unspent transaction outputs) that fund it:

- `inputs`: a list of past TXOs $(\texttt{txid}, j)$, i.e. the jth output of $\texttt{txn}_{\texttt{txid}}$, where the sender received money.
- `inputProofs`: for each $(\texttt{txid}, j)$ in `inputs`, a Merkle proof linking each input to the `root` in some past $\texttt{header}(k)$ in the chain, where $k < i$.

Given this block structure, a transaction is valid if

1) The signature is valid.
2) The sum of inputs equals the sum of outputs.[3]
3) The input Merkle proofs are valid.
4) The corresponding TXOs are unspent.

Any node can check (1)-(3) with just the header chain, but checking (4) requires some knowledge of the current *state*, i.e. some representation of the transaction history. For our protocol, validators store the state as a hash table `spentTXOs` mapping the IDs of spent TXOs to the transaction that spent it, along with a Merkle proof of that transaction to enable fraud proofs (see Section 2.2).

Assuming that each transaction has a constant number of outputs, after T blocks with L transactions per block, the size of this table will be $O(TL \log L)$. We will reduce the size of this table by removing the dependence on T in Section 2.4, and we will further reduce to $O((L/k) \log L)$ by dividing the work among the participants in Section 2.3.

Then, a node can validate `txn` in block i as follows:

$\texttt{is\_valid\_txn}(\texttt{txn}, \texttt{spentTXOs}) \in \{\texttt{True}, \texttt{False}\}$

1: Check if the `signature` is valid.
2: Check that sum of `inputs` equals sum of `outputs`.
3: Check that the `inputProofs` are valid.
4: Check for double payment: check that each TXO in `inputs` is not in the table `spentTXOs`.
5: If any check fails, return `False` and broadcast a `fraudProof`, as described in the following section.
6: If all checks succeed, return `True`.

If the block is valid, then for each `txn`, a node can update its state storage by including the spent TXOs as follows:

$\texttt{update\_state}(\texttt{txn}, \texttt{spentTXOs})$

1: For each $(\texttt{txid}, j)$ in `inputs`, insert the entry $(\texttt{txid}, j) : (\texttt{txn}, \texttt{proof}(\texttt{txn} \rightarrow \texttt{header}(i).\texttt{root}))$ into the hash table `spentTXOs`.

2.2. CoVer Fraud Proofs

If a transaction in the block is invalid, then a validator can produce a fraud proof, which proves that the block contains an invalid transaction and can be checked by any node storing the header chain. A `fraudProof` contains:

- `invalidTxn`: the invalid transaction.
- `invalidTxnProof`: the Merkle proof to header i.

3. If the inputs are greater, the sender can send herself the change. For simplicity, we do not consider transaction fees.

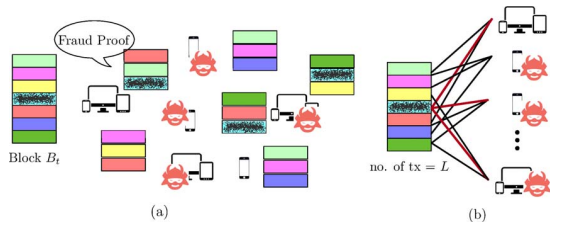

Figure 2. Collaborative verification using fraud proofs. Each light node will verify roughly $1/k$ fraction of each block on average. Honest light nodes generate fraud proofs for any invalid transactions that they verify. Every invalid transaction will be caught with a fraud proof as long as each of the k sections is *covered* (i.e., validated) by at least one honest participant.

- `pastTxn`: if the transaction is invalid due to double spending, the proof contains a past transaction from the same `spender` with at least one colliding input.
- `pastTxnProof`: the Merkle proof to some past header k, where $k < i$.

Then, any recipient can check the fraud proof as follows:

`is_valid_fraudProof(fraudProof)` $\in \{$`T`,`F`$\}$

1: Check `invalidTxnProof` and `pastTxnProof`. If either proof is invalid, return `False`.
2: Perform the first three checks of `is_valid_txn(invalidTxn,` •`)`. If any check fails, then the transaction is invalid, so return `True`.
3: Check for double spending: if the intersection `invalidTxn.inputs` $\cap$ `pastTxn.inputs` is non-empty, then return `True`.
4: Otherwise, return `False`.

Assuming that the number of inputs and outputs is constant and hash table lookup is constant time, the heaviest computation is checking a constant number of Merkle proofs, so the runtimes of `is_valid_fraudProof` and `is_valid_txn` are both $O(\log L)$.

2.3. Collaborative Validation

Note that a node can verify a transaction as long as it has the history of its sender, so a node can choose a segment $[a, b]$ of account numbers and only validate transactions sent by those accounts.[4] We will divide the account numbers into k equal-sized sections and each node will choose one section at random, resulting in the following protocol (see Fig. 2):

1) Download the `header`.
2) Download the transactions `txn` where `txn.sender` is in the node's chosen account section. The selective downloading scheme is described in Section 4.
3) Validate the transactions and broadcast a `fraudProof` if any is invalid.
4) Wait some fixed delay. Reject the block if you receive a valid `fraudProof`, otherwise accept it.
5) If the block is accepted, update `spentTXOs` to include the newly spent `inputs` for each `txn`.

4. To facilitate partial validation, the transactions in a block should be sorted by the sender's account number. A "sorting fraud proof" would contain two transactions that are out of order, plus their Merkle proofs.

Note that malicious nodes cannot generate fraud proofs for valid transactions, and a single valid fraud proof is sufficient to discard the block. Then, the scheme is secure as long as each of the k sections is covered by at least one honest participant. We analyze the probability of coverage in Section 5 and show that we need $N_h > k(\log k + \lambda)$ honest nodes, each validating $1/k$th of the block, to cover the block with probability at least $1 - e^{-\lambda}$. Therefore, the fraction $1/k$ of work required scales down proportionally with the number N_h of honest nodes, up to a log factor. Honest nodes with more resources can verify more than one section, counting as multiple nodes in the calculation above.

2.4. Reducing State Storage and Switching Sections

One shortcoming of the above approach is that the amount of storage grows linearly with the length of the blockchain. Specifically, each node stores $O(L/k)$ transactions per block on average, each of which is size $O(\log L)$, resulting in $O(T\frac{L}{k}\log L)$ storage after T rounds. To reduce this storage, we add an *expiration time* inspired by [6], where each transaction can only be spent up to τ blocks after it is mined (note that the TXO owner can pay herself to avoid expiration). Then, storage is reduced to $O(\tau\frac{L}{k}\log L)$.

While each validator can choose to stick to one account section forever, it may be desirable from a security perspective for validators to periodically switch sections, e.g. in the case of slowly adaptive adversaries. Because TXOs are only valid up to τ blocks after they are mined, two transactions whose inputs intersect must be within $\tau - 1$ blocks of each other. Then, a validator can join a new section as follows:

1) For time steps $t, t+1, ..., t+(\tau-1)$, download the transactions for the new section.
2) At time step $t+\tau$, the validator now has enough history to validate the new section.

3. Collaborative Data Availability Coding

3.1. Coding Structure

As described in Section 1.2, a fraud proof cannot be produced for a hidden transaction, so light nodes must be assured that the entire block is available *without downloading the entire block*. To solve this problem, known as *data availability*, the high-level idea in [3] is to require miners to encode their block with an erasure code. This idea is improved in [4] via a coding scheme that combines Low-Density Parity-Check (LDPC) codes with Merkle trees.

First, we review the LDPC erasure code [7] [8]. Starting with n data symbols, a rate-$1/2$ LDPC code produces n additional coded symbols, resulting in $2n$ symbols total. The code is specified by a set of parity equations, which signify that some subset of the symbols sum to zero, and can be represented by a bipartite graph with n left vertices (symbols) and $n - k$ right nodes (parity equations); see Fig. 4. The code has the following properties:

1) Within the subclass of left- and right-regular LDPC codes, each parity equation is connected to a constant

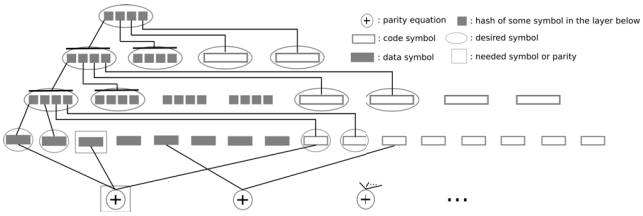

Figure 3. The structure of a coded Merkle tree [4]. Each row is encoded using a rate-$1/2$ code. Then, symbols are divided into groups of 4, and their hashes are taken to produce the next level. The arrows denote a sub-tree, starting from the 4 symbols sampled on the bottom layer. One possible sampled subtree is denoted by the circled symbols, and some example parities and symbols needed to decode this sampled subtree are boxed.

number d_R of symbols, and each symbol is connected to a constant number d_L of parity equations.

2) The code can decoded via *peeling*, which has time complexity linear in the number of symbols:
   a) Starting with some set of known symbols, our goal is to recover the unknown symbols.
   b) Find a *singleton*, or a parity equation containing only one unknown symbol. Compute the value of this symbol using the parity equation, and *peel* this symbol by subtracting and removing it from each parity equation it participates in.
   c) Repeat until all the unknown symbols are decoded (success), or there are no more singletons (failure).
3) A *stopping set* denotes a set of symbols that, if removed, prevent peeling decoding from succeeding. It is possible to construct an LDPC code such that the size of the smallest stopping set is a constant fraction f of the total number of symbols.

Next, we review the LDPC-coded Merkle tree [4] (see Fig. 3). The lowest level contains the L transactions, which are coded into $2L$ symbols via a rate-$1/2$ LDPC code. Then, we take the hash of each symbol and group the hashes into groups of 4 to produce the $L/2$ data symbols in the next layer. We then code these $L/2$ data symbols into L symbols using an LDPC code, and so on. As in a typical Merkle tree, the Merkle proof for a symbol consists of its path to the root. A key property of this structure is that after decoding the ℓ-th layer, the decoder has the hashes for the $(\ell+1)$-th layer, which will be exploited in the protocols below.

3.2. Checking Availability

To make any symbol in the layer unavailable, the miner must prevent peeling decoding from succeeding by hiding a number of symbols that is at least the size of a stopping set, which by the LDPC code is at least a constant fraction f of the data. Leveraging this idea, a light node can check for availability by randomly sampling c symbols from each layer. If the miner has made a layer unavailable, then the node will sample an unavailable symbol with probability at least $1-(1-f)^c$, which decays quickly in c. Therefore, each node samples c symbols for each of the $\log L$ layers, along with their proofs to the root, and it accepts the data as available if and only if it receives all of its sampled data, resulting in $O(c(\log L)^2)$ total data downloaded.

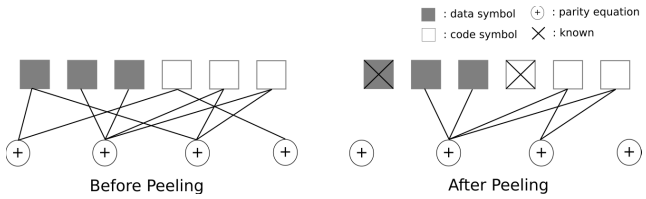

Figure 4. A graph depiction of an LDPC code with 3 data symbols and 3 coded symbols, represented by gray and white boxes. Each $+$ circle denotes a parity equation, which signifies that the attached symbols sum to some known value, initially zero before peeling. After recursively peeling singletons (right), the first and fourth symbols are known.

Using an interleaving approach in the tree structure, the amount of downloaded data can be reduced to $O(c\log L)$ [4]. Specifically, a node first chooses c symbols on the bottom layer. Then, it also downloads their paths to the root, as well as any siblings of symbols on these paths. The arrows in Figure 3 denote one such sampled subtree. This subtree contains c randomly chosen symbols from each layer, achieving the same sampling guarantees as above. Furthermore, because the sampled symbols form a subtree, each symbol's path to the root and therefore its Merkle proof is contained in the sampled data.

3.3. Classical Decoding

For the protocol above to function, however, the validators must have the capability to decode the block, otherwise the miner can hide small parts of the block without the fear that it will be decoded. We first review non-collaborative decoding, as described in [4]. First, the miner revealing some subset of the symbols along with Merkle proofs. Using these revealed symbols, the node attempts to decode the remaining symbols in the tree. We will decode layer by layer via peeling, starting from the top of the tree. For each layer, the decoding protocol should have the following properties:

1) If the set of hidden symbols contains a stopping set, then peeling decoding will fail and the node will reject the block as unavailable.
2) Otherwise, if there is coding fraud (at least one of the parity equations does not sum to zero), then the decoding procedure will produce a coding fraud proof.
3) Otherwise, decoding is successful.

The layered structure of the coded Merkle tree allows one to catch coding fraud. Suppose that the codes for layers $1,...,\ell-1$ are valid, but there is coding fraud in layer ℓ. Because the node already decoded layer $\ell-1$, it knows the hash of each unknown symbol in layer ℓ. Then, if the node uses an invalid parity equation to decode a new symbol, the hash of this new decoded symbol will not match the known hash in layer $\ell-1$, allowing it to catch coding fraud. Using this idea, a node can decode a layer as follows [4]:

`decode_layer(knownSymbols, hashes)`

`1:` While there are still unknown symbols,

- Look for a degree-one parity equation, i.e. an equation where all but one symbol are known.
- The unknown symbol is now known. Check that its hash matches the known hash in `hashes`. If the hash matches, add the symbol to `knownSymbols`.
- Otherwise, stop and produce a coding fraud proof.
- If there are no degree-one parity equations, stop and reject the block as unavailable.

2: Once everything is decoded, check the parity equations and make sure they sum to zero. If any equation does not, stop and produce a coding fraud proof.
3: Otherwise, this layer is valid and has been decoded.

Then, a node can decode the entire tree as follows:

`decode_tree()`

1: For each layer ℓ, keep track `knownSymbols`(ℓ) and their hashes `hashes`(ℓ). Download as many symbols as possible, and add each symbol with a valid Merkle proof to `knownSymbols`(ℓ).
2: Initialize `hashes`(1) to the `root` of the Merkle tree.
3: For layer $\ell = 1, 2, ..., \lceil \log L \rceil$:
- Call `decode_layer(knownSymbols`(ℓ), `hashes`$(\ell))$, which produces `hashes`$(\ell + 1)$.

3.4. Coding Fraud Proofs

Suppose that after peeling some degree-one parity equation using symbols $s_1, ...s_{d-1}$, the decoded symbol $\hat{s}_d$ has an incorrect hash. Because this parity equation was the first to fail, the $d-1$ previously known symbols and the dth hash are all valid, known, and have Merkle proofs to the root. Then, a `codingFraudProof` contains the following:

- $\hat{s}_d$: the newly decoded incorrect symbol.
- h_d: the committed hash of the d-th symbol.
- `hashProof`: the Merkle proof of the hash h_d.
- $s_1, s_2, ..., s_{d-1}$: the $d-1$ known symbols corresponding to the parity-check equation for $\hat{s}_d$.
- `symbolProofs`: Merkle proofs of $s_1, s_2, ..., s_{d-1}$.

`is_valid_codingFraudProof(codingFraudProof)` $\in \{$`True`, `False`$\}$

1: The `hashProof` and `symbolProofs` are correct.
2: There is a parity equation involving these d symbols.
3: Decoding this parity equation results in the $\hat{s}_d$ such that $0 = \hat{s}_d + \sum_{i=1}^{d-1} s_i$.
4: The commitment is invalid, or `hash`$(\hat{s}_d) \neq h_d$.
5: If any of the above checks are false, return `False`. Otherwise, return `True`.

The coding fraud proof is size $O(d_R \log L)$ from the d_R Merkle proofs, where d_R (LDPC right degree) is constant.

3.5. Collaborative Decoding

To enable this protocol for light nodes, we propose a collaborative decoding scheme. At a high level, decoding follows the steps described above, but each node decodes only a subtree of the entire tree. Every time a node decodes a new symbol, it broadcasts it, allowing other nodes to use it. Each broadcasted symbol must be accompanied by a Merkle proof so that malicious nodes cannot broadcast fake symbols. Decoding succeeds as long as together, the honest nodes cover the entire tree.

First, we describe the subtree version of `decode_layer`, assuming the previous layer in the subtree has already been decoded. Decoding layer ℓ involves the following sets:

- `desiredSymbols`(ℓ): the layer ℓ symbols in the subtree. Note that their Merkle proofs are contained in previous layers of the subtree.
- `knownDesiredSymbols`(ℓ): the subset of the desired symbols that have been successfully decoded.
- `hashes`(ℓ): the hashes of the desired symbols, which are in `knownDesiredSymbols`$(\ell - 1)$, or the previous layer of the subtree.
- `neededParities`(ℓ): any parity equations attached to at least one unknown desired symbol.
- `neededSymbols`(ℓ): any symbol attached to a needed parity equation.[5]
- `knownNeededSymbols`$_\ell$: the subset of needed symbols that are known, along with their Merkle proofs.

At a high level, each node's job is to securely decode the symbols in `desiredSymbols`(ℓ), and it has stored their hashes after decoding the previous layer. To accomplish this job, the node needs to decode parities, so it downloads the symbols in `neededSymbols`(ℓ), depending on other nodes to decode them. An example is shown in Figure 3. To make the protocol secure, any downloaded symbols must be accompanied by a Merkle proof, and the node also broadcasts any symbol it decodes with an Merkle proof. Specifically, decoding a layer proceeds as follows:

`collab_decode_layer(hashes, knownDesiredSymbols, knownNeededSymbols)`

1: While there are still unknown desired symbols,
- Look for a degree-one needed parity equation (all but one symbol are in `knownDesiredSymbols` or `knownNeededSymbols`).
- The newly decoded symbol is a desired symbol. Check that its hash matches the known hash in `hashes`. If the hash matches, add the symbol to `knownDesiredSymbols` and broadcast it with a Merkle proof. Otherwise, stop and produce a `codingFraudProof` (see Section 3.4).
- Listen for any broadcasts of `neededSymbols`. Check their Merkle proofs. If the Merkle proof is valid, add the symbol and its proof to `knownNeededSymbols`. Store the Merkle proof in case it is needed for a `codingFraudProof`.

5. Note that because the left and right degree of the LDPC code are at most d_L and d_R, we have $|$`neededSymbols`$| \leq d_R d_L$ $|$`desiredSymbols`$|$, so the size of this set is not too big.

- If there are no degree-one parity equations, wait some fixed delay. If there were no additional needed symbols broadcasted in this delay, stop and reject the block as unavailable.

2: Once everything is decoded, check degree-zero needed parity equations and make sure they sum to zero. If any equation does not, produce a coding fraud proof.

3: Otherwise, this layer is valid and has been decoded.

We can decode an entire subtree by decoding each layer:

`decode_subtree()`

1: Choose a subtree of $O(c \log L)$ symbols, as described in Section 3.2 and Figure 3. This subtree defines the $\texttt{desiredSymbols}(\ell)$, $\texttt{neededParities}(\ell)$, and $\texttt{neededSymbols}(\ell)$ for each layer ℓ.

2: Attempt to download any desired or needed symbols from the network, along with their Merkle proofs. Add each symbol with a valid proof to $\texttt{knownDesiredSymbols}(\ell)$ or $\texttt{knownNeededSymbols}(\ell)$, where ℓ is the layer that the symbol belongs to.

3: For layer $\ell = 1, 2, ..., \lceil \log L \rceil$,

- Decode using

$$\texttt{collab\_decode\_layer}(\texttt{knownDesiredSymbols}_{\ell}, \texttt{knownDesiredSymbols}_{\ell-1}, \texttt{knownNeededSymbols}_{\ell}).$$

- If decoding was unsuccessful, stop.
- Also listen for any `codingFraudProof`. If any is valid, stop and reject the block.

Using similar calculations as Section 5, if there are at least

$$N_h > \frac{L}{c}(\log L + \lambda)$$

honest nodes, then the entire tree is covered with probability at least $(1 - e^{-\lambda})$. Note that, as for validation, the fraction $\frac{c}{L}$ of work required scales down proportionally with N_h up to a log factor. For simplicity, we can set $c = L/k$, such that $1/k$-th of the block is sampled for validation and another $1/k$-th is sampled for availability.

4. Selective Broadcast

Finally, because light nodes cannot download the entire block, we adapt the conventional gossip protocol to enable selective downloading. The conventional gossip protocol, which we use for block headers and any fraud proofs, proceeds as follows (see [9] for more details):

1) The starting node sends the message to all neighbors.
2) Each neighboring node does the following: if the node has not already seen the message, it first validates the message (e.g., checks if the fraud proof is valid). If the message is valid, the node forwards the message to all of its neighbors. If the message is invalid or has been seen, the node ignores it.

Note that fake communication from malicious nodes are mitigated by honest nodes only gossiping valid messages.

Other broadcasts, like ones involving symbols in the tree, are only of interest to a subset of the nodes. To reduce communication, we propose and analyze a simple modification of the gossip protocol, called *selective broadcast*:

1) Each node chooses the symbols it is interested in and informs its neighbors of these symbols.
2) A node only gossips each received symbol to neighbors that are also interested in that symbol.

Then, the communication complexity per node is simply the total size of the data it wishes to download, plus the initial step of informing neighbors of desired symbols, which has complexity linear in the number of neighbors. The protocol is successful, meaning that every node receives the data it wishes to download, as long as for each symbol, the subgraph of nodes interested in that symbol is connected.

This property always holds if the network graph is fully connected. While nodes would need many neighbors, the only communication that scales with number of neighbors is the initial information step. Because these messages are relatively small, this communication is small compared to the work needed to download, decode, and validate block data, especially if the number of participants is small. Therefore, it is reasonable with a small group of light nodes.

If the number of participants is large, then we can reduce the connectivity. Under the simplifying assumption that the graph is Erdős-Rényi [5], we analyze the required connectivity in Section 5 and find that each node should have at least $O(k \log N_h)$ neighbors, where $1/k$ is the fraction of each block that each node downloads. In the case where there are malicious nodes, these nodes can choose to ignore all messages, increasing the amount of connectivity needed. In this case, each honest node will need $O\left(\frac{1}{1-\alpha} k \log N_h\right)$ neighbors, where α is the fraction of malicious nodes. This term, which represents the amount of communication during the information step, is linear in k, which is exactly the factor of work reduced. Therefore, choosing a savings factor of $k = O(\sqrt{L})$ results in $O(\sqrt{L}(\log L + \log N_h))$ computation, bandwidth, and storage.

5. Analysis of CoVer

In this section, we formally analyze CoVer; please see the appendix of our arXiv paper[6] for full proofs. Let each of the N_h honest nodes choose a section out of k at random to validate, and a random $1/k$th to sample for availability. Let λ denote the security parameter such that the protocol succeeds with probability $\geq 1 - e^{-O(\lambda)}$. First, we calculate the number of nodes needed to cover every section with probability $\geq 1 - e^{-\lambda}$ and find that we need $N_h \geq k(\log k + \lambda)$. Next, given that the block is unavailable, all honest nodes sample at least one unavailable symbol with probability $\geq 1 - N_h(1-f)^{L/k}$. Finally, we calculate connectivity required for selective broadcast and find that if each node

6. https://arxiv.org/abs/2010.00217

has $O(k\lambda \log N_h)$ neighbors, the success probability is at least $\geq 1-4L\left(\frac{N_h}{8k}\right)^{1-\lambda}-4L\exp\left(-\frac{N_h}{8(4k-1)}\right)$. Putting these results together via the union bound, we analyze correctness and show that with high probability, all light nodes accept the block if and only if all of its data is available for download, and every included transaction is valid.

Theorem 5.1. *CoVer satisfies the following:*

1) *If the block is valid and available, then all honest participants will accept the block with probability* $\geq 1-e^{-\lambda}-4L\left(\frac{N_h}{8k}\right)^{1-\lambda}-4L\exp\left(-\frac{N_h}{8(4k-1)}\right)$.
2) *If the block is unavailable, then all honest participants will reject with probability* $\geq 1-N_h(1-f)^{L/k}$.
3) *If the block is invalid but available, then all honest participants will reject the block with probability* $\geq 1-e^{-\lambda}-4L\left(\frac{N_h}{8k}\right)^{1-\lambda}-4L\exp\left(-\frac{N_h}{8(4k-1)}\right)$.

6. Related Work

To the best of our knowledge, CoVer is the first light-node-only protocol for block validation. First, we briefly survey works related to light node protocols. Light nodes, or simplified payment verification (SPV) clients, were proposed by Nakamoto in the original Bitcoin paper [1]. These nodes only store block headers and perform no validation, relying on full nodes for transaction membership proofs. As a result, they are vulnerable to several privacy and security attacks [2], and several works are aimed at improving their privacy and security; see, e.g., [10]–[12]. Rather than improving security, another line of work instead reduces the light node computational burden even further by enabling them to download only a sublinear (in the length of the blockchain) number of block headers [13]–[15].

Several works focus on reducing the costs associated with running a full node, especially storage costs [16]–[18] and communication cost during bootstrap [6]. For sharded blockchains, Polyshard [19] proposes a protocol for verification functions that can be represented as polynomials.

Our work directly builds on top of recent work on fraud proofs and data availability. In [3], Al-Bassam et al. propose the ideas of fraud proofs and data availability, allowing light nodes to receive compact proofs of block invalidity from full nodes. While Al-Bassam et al. use 2D Reed Solomon codes for data availability, Yu et al. [4] propose a new coding scheme called the coded Merkle tree to further reduce the coding fraud proof sizes and decoding complexity. While these works require light nodes to rely on full nodes, our main contribution is to remove this requirement through light-node-only protocols.

7. Conclusion

We propose CoVer, a decentralized and permissionless protocol that enables light nodes to collaboratively validate blocks even under a dishonest majority, achieving the same level of security as full nodes with a fraction of the work. CoVer allows nodes of all sizes to contribute to network security, bringing us closer to fully scalable blockchains and enabling a new world of possibilities.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] G. Karame and E. Audroulaki, *Bitcoin and Blockchain Security*. Norwood, MA, USA: Artech House, Inc., 2016.

[3] M. Al-Bassam, A. Sonnino, and V. Buterin, "Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities," *CoRR*, vol. abs/1809.09044, 2018. [Online]. Available: http://arxiv.org/abs/1809.09044

[4] M. Yu, S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded merkle tree: Solving data availability attacks in blockchains," in *Financial Cryptography and Data Security (FC)*, 2020. [Online]. Available: https://eprint.iacr.org/2019/1139.pdf

[5] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960.

[6] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast bootstrapping for the algorand cryptocurrency," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[7] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[8] T. Richardson and R. Urbanke, *Modern Coding Theory*. New York, NY, USA: Cambridge University Press, 2008.

[9] K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.*, p. 8–13, Oct. 2007.

[10] M. Hearn and M. Corallo, "Connection bloom filtering," Bitcoin Improvement Proposal 37, 2012, https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki.

[11] O. Osuntokun, A. Akselrod, and J. Posen, "Client side bloom filtering," Bitcoin Improvement Proposal 157, 2017, https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki.

[12] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "BITE: Bitcoin lightweight client privacy using trusted execution," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019.

[13] A. Kiayias, N. Lamprou, and A.-P. Stouka, "Proofs of proofs of work with sublinear complexity," in *Financial Cryptography Workshops*, 2016.

[14] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work," in *Financial Cryptography and Data Security (FC)*, 2020. [Online]. Available: https://eprint.iacr.org/2017/963.pdf

[15] B. Buüñz, L. Kiffer, L. Luu, and M. Zamani, "FlyClient: Super-light clients for cryptocurrencies," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[16] R. K. Raman and L. R. Varshney, "Dynamic distributed storage for scaling blockchains," *CoRR*, vol. abs/1711.07617, 2017. [Online]. Available: http://arxiv.org/abs/1711.07617

[17] D. Perard, J. Lacan, Y. Bachy, and J. Detchart, "Erasure code-based low storage blockchain node," *CoRR*, vol. abs/1805.00860, 2018. [Online]. Available: http://arxiv.org/abs/1805.00860

[18] S. Kadhe, J. Chung, and K. Ramchandran, "SeF: A secure fountain architecture for slashing storage costs in blockchains," in *Scaling Bitcoin*, Sep. 2019. [Online]. Available: https://arxiv.org/pdf/1906.12140

[19] S. Li, M. Yu, C. Yang, A. S. Avestimehr, S. Kannan, and P. Viswanath, "Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 249–261, 2020.