Fraud Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities

Mustafa Al-Bassam[1], Alberto Sonnino[1], and Vitalik Buterin[2]

[1] University College London
{m.albassam,a.sonnino}@cs.ucl.ac.uk
[2] Ethereum Research
vitalik@ethereum.org

Abstract. Light clients, also known as Simple Payment Verification (SPV) clients, are nodes which only download a small portion of the data in a blockchain, and use indirect means to verify that a given chain is valid. Typically, instead of validating block data, they assume that the chain favoured by the blockchain's consensus algorithm only contains valid blocks, and that the majority of block producers are honest. By allowing such clients to receive fraud proofs generated by fully validating nodes that show that a block violates the protocol rules, and combining this with probabilistic sampling techniques to verify that all of the data in a block actually is available to be downloaded, we can eliminate the honest-majority assumption, and instead make much weaker assumptions about a minimum number of honest nodes that rebroadcast data. Fraud and data availability proofs are key to enabling on-chain scaling of blockchains (*e.g.,* via sharding or bigger blocks) while maintaining a strong assurance that on-chain data is available and valid. We present, implement, and evaluate a novel fraud and data availability proof system.

1 Introduction and Motivation

As cryptocurrencies and smart contract platforms have gained wider adoption, the scalability limitations of existing blockchains have been observed in practice. Popular services have stopped accepting Bitcoin [25] payments due to transactions fees rising as high as $20 [18,27], and Ethereum's [6] popular CryptoKitties smart contract caused the pending transactions backlog to increase six-fold [39]. Users pay higher fees as they compete to get their transactions included on the blockchain, due to on-chain space being limited, *e.g.,* by Bitcoin's block size limit [2] or Ethereum's block gas limit [40].

While increasing on-chain capacity limits would yield higher transaction throughput, there are concerns that this would decrease decentralisation and security, because it would increase the resources required to fully download and validate the blockchain, and thus fewer users would be able to afford to run full nodes that independently validate the blockchain, requiring users to instead run

light clients that assume that the chain favoured by the blockchain's consensus algorithm abides by the protocol rules [22]. Light clients operate well under normal circumstances, but have weaker assurances when the majority of the consensus (*e.g.,* miners or block producers) is dishonest; for example, whereas a dishonest majority in the Bitcoin or Ethereum network can at present only censor, reverse or reorder transactions, if all clients are using light nodes, a majority of the consensus would be able to collude together to generate blocks that contain transactions that create money out of thin air, and light nodes would not be able to detect this. On the other hand, full nodes would reject those invalid blocks immediately.

As a result, various scalability efforts have focused on off-chain scaling techniques such as payment channels [30], where participants sign transactions off-blockchain, and settle the final balance on-chain. Payment channels have also been generalised to state channels [24]. However, as opening and settling channels involves on-chain transactions, on-chain scaling is still necessary for widespread adoption of payment and state channels.[3]

In this paper, we decrease the on-chain capacity vs. security trade-off by making it possible for light clients to receive and verify fraud proofs of invalid blocks from full nodes, so that they too can reject them, assuming that there is at least one honest full node willing to generate fraud proofs to be propagated within a maximum network delay. We also design a data availability proof system, a necessary complement to fraud proofs, so that light clients have assurance that the block data required for full nodes to generate fraud proofs from is available, given that there is a minimum number of honest light clients to reconstruct missing data from blocks. We implement and evaluate the security and efficiency of our overall design.

Our work also plays a key role in efforts to scale blockchains with sharding [1, 7, 19], as in a sharded system no single node in the network is expected to download and validate the state of all shards, and thus fraud proofs are necessary to detect invalid blocks from malicious shards.

2 Background

2.1 Blockchain Models

Briefly, the data structure of a blockchain consists of (literally) a chain of blocks. Each block contains two components: a header and a list of transactions. In addition to other metadata, the header stores at minimum the hash of the previous

[3] Suppose a setting where all users used channels and channels only needed to be opened once and maintained with on-chain transactions once per year per used. To support a userbase equal in size to Facebook's ($\approx$ 2.2 billion [26]), one would need 2.2 billion transactions per year, or $\approx$ 70 transactions per second, significantly higher than supported by the Bitcoin or Ethereum blockchains [9, 43]. This does not take into account usages that require "going on-chain" more frequently, users requiring multiple channels, or the possibility of attacks on channels requiring more transactions to process.

block (thus enabling the chain property), and the root of the Merkle tree that consists of all transactions in the block.

Blockchain networks have a consensus algorithm [3] to determine which chain should be favoured in the event of a fork, *e.g.,* if proof-of-work [25] is used, then the chain with the most accumulated work is favoured. They also have a set of protocol rules that dictate which transactions are valid, and thus blocks that contain invalid transactions will never be favoured by the consensus algorthim and should in fact always be rejected.

Full nodes are nodes which download block headers as well as the list of transactions, verifying that the transactions are valid according to some protocol rules. Light clients only download block headers, and assume that the list of transactions are valid according to the protocol rules. Light clients verify blocks against the consensus rules, but not the protocol rules, and thus assume that the consensus is honest. Light clients can receive Merkle proofs from full nodes that a specific transaction or state object is included in a block header.

There are two major types of blockchain transaction models: Unspent Transaction Output (UTXO)-based, and account-based. Transactions in UTXO-based blockchains (*e.g.,* Bitcoin) contain references to previous transactions whose coins they wish to 'spend'. As a single transaction may send coins to multiple addresses, a transaction has many 'outputs', and thus new transactions contain references to these specific outputs. Each output can only be spent once.

On the other hand, account-based blockchains (*e.g.,* Ethereum), are somewhat simpler to work with (though sometimes more complex to apply parallelisation techniques to), as each transaction simply specifies a balance transfer from one address to another, without reference to previous transactions. In Ethereum, the block header also contains a root to a Merkle tree containing the state, which is the 'current' information that is required to verify the next block; in Ethereum this consists of the balance, code and permanent storage of all of the accounts and contracts in the system.

2.2 Merkle Trees and Sparse Merkle Trees

A Merkle tree [23] is a binary tree where every non-leaf node is labelled with the cryptographic hash of the concatenation of its children nodes. The root of a Merkle tree is thus a commitment to all of the items in its leaf nodes. This allows for Merkle proofs, which given some Merkle root, are proofs that a leaf is a part of the tree committed to by the root. A Merkle proof for some leaf consists of all of the ancestor and ancestor's sibling intermediate nodes for that leaf, up to the root of the tree, thus forming a sub-tree whose Merkle root can be recomputed to verify that the Merkle proof is valid. The size and verification time of a Merkle proof for a tree with n leaves is $O(\log(n))$, as it is a tree.

A sparse Merkle tree [11, 20] is a Merkle tree with n leaves where n is extremely large (*e.g.,* $n = 2^{256}$), but where almost all of the nodes have the same default value (*e.g.,* 0). If k nodes are non-zero, then at each intermediate level of the tree there will be a maximum of k non-zero values, and all other values will be the same default value for that level: 0 at the bottom level, $L_1 = H(0, 0)$

at the first intermediate level, $L_2 = H(L_1, L_1)$ at the second intermediate level, and so on. Hence, despite the exponentially large number of nodes in the tree, the root of the tree can be calculated in $O(k \times log(n))$ time. A sparse Merkle tree allows for commitments to key-value maps, where values can be updated, inserted or deleted trivially in $O(\log(n))$ time. Merkle proofs of specific key-values entries are of size $\log(n)$ if constructed naively but can be compressed to size $\log(k)$ as intermediate nodes whose sibling have the default value do not need to explicitly be shown.

Systems such as Ripple and Ethereum at present use Patricia trees instead of sparse Merkle trees [34, 40]; we use sparse Merkle trees in this paper because of their greater simplicity.

2.3 Erasure Codes and Reed-Solomon Codes

Erasure codes are error-correcting codes [13, 29] working under the assumption of bit erasures rather than bit errors; in particular, the users knows which bits have to be reconstructed. Error-correcting codes transform a message of length k into a longer message of length $n > k$ such that the original message can be recovered from a subset of the n symbols.

Reed-Solomon (RS) codes [38] have various applications and are among the most studied error-correcting codes. A Reed-Solomon code encodes data by treating a length-k message as a list of elements $x_0, x_1, ..., x_{k-1}$ in some finite field (prime fields and binary fields are most frequently used), interpolating the polynomial $P(x)$ where $P(i) = x_i$ for all $0 \leq i < k$, and then extending the list with $x_k, x_{k+1}, ..., x_{n-1}$ where $x_i = P(i)$. The polynomial P can be recovered from any k symbols from this longer list using techniques such as Lagrange interpolation, or more optimized and advanced techniques involving tools such as Fast Fourier transforms, and knowing P one can then recover the original message. Reed-Solomon codes can detect and correct any combination of up to $\frac{n-k}{2}$ errors, or combinations of errors and erasures. RS codes have been generalised to multidimensional codes [12, 35] in various ways [33, 36, 41]. In a d-dimensional code, the message is encoded into a square or cube or hybercube of size $k \times k \times ... \times k$, and a multidimensional polynomial $P(x_1, x_2, ..., x_d)$ is interpolated where $P(i_1, i_2, ..., i_n) = x_{i_1, i_2 ..., i_n}$, and this polynomial is extended to a larger $n \times n \times ... \times n$ square or cube or hypercube.

3 Assumptions and Threat Model

We present the network and threat model under which our fraud proofs (Section 4) and data availability proofs (Section 5) apply.

3.1 Preliminaries

We present some primitives that we use in the rest of the paper.

- $\mathsf{hash}(x)$ is a cryptographically secure hash function that returns the digest of x (*e.g.,* SHA-256).
- $\mathsf{root}(L)$ returns the Merkle root for a list of items L.
- $\{e \rightarrow r\}$ denotes a Merkle proof that an element e is a member of the Merkle tree committed by root r.
- $\mathsf{VerifyMerkleProof}(e, \{e \rightarrow r\}, r, n, i)$ returns $true$ if the Merkle proof is valid, otherwise $false$, where n additionally denotes the total number of elements in the underlying tree and i is the index of e in the tree. This verifies that e is at index i, as well as its membership.
- $\{k, v \rightarrow r\}$ denotes a Merkle proof that a key-value pair k, v is a member of the Sparse Merkle tree committed by root r.

3.2 Blockchain Model

We assume a generalised blockchain architecture, where the blockchain consists of a hash-based chain of block headers $H = (h_0, h_1, ..., h_n)$. Each block header h_i contains a Merkle root txRoot_i of a list of transactions T_i, such that $\mathsf{root}(T_i) = \mathsf{txRoot}_i$. Given a node that downloads the list of transactions N_i from the network, a block header h_i is considered to be valid if (*i*) $\mathsf{root}(N_i) = r_i$ and (*ii*) given some validity function

$$\mathsf{valid}(T, S) \in \{true, false\}$$

where T is a list of transactions and S is the state of the blockchain, then $\mathsf{valid}(T_i, S_{i-1})$ must return $true$, where S_i is the state of the blockchain after applying all of the transactions in T_i. We assume that $\mathsf{valid}(T, S)$ takes $O(n)$ time to execute, where n is the number of transactions in T.

In terms of transactions, we assume that given a list of transactions $T_i = (t_i^0, t_i^1, ..., t_i^n)$, where t_i^j denotes a transaction j at block i, there exists a state transition function $\mathsf{transition}$ that returns the post-state S' of executing a transaction on a particular pre-state S, or an error if the transition is illegal:

$$\mathsf{transition}(S, t) \in \{S', err\}$$

$$\mathsf{transition}(err, t) = err$$

Thus given the intermediate post-states after applying every transaction one at a time, $I_i^j = \mathsf{transition}(I_i^{j-1}, t_i^j)$, and the base case $I_i^{-1} = S_{i-1}$, then $S_i = I_i^n$. Hence, I_i^j denotes the intermediate state of the blockchain at block i after applying transactions $t_i^0, t_i^1, ..., t_i^j$.

Therefore, $\mathsf{valid}(T_i, S_{i-1}) = true \iff I_i^n \neq err$.

In Section 4.2, we explain how both a UTXO-based (*e.g.,* Bitcoin) and an account-based (*e.g.,* Ethereum) blockchain can be represented by this model.

Aim. Our aim is to prove to clients that for a given block header h_i, $\mathsf{valid}(T_i, S_{i-i})$ returns $false$ in less than $O(n)$ time and less than $O(n)$ space, relying on as few security assumptions as possible.

3.3 Network Model

We assume a network that consists of two types of nodes:

- **Full nodes.** These are nodes which download and verify the entire blockchain. Honest full nodes store and rebroadcast valid blocks that they download to other full nodes, and broadcast block headers associated with valid blocks to light clients. Some of these nodes may participate in consensus (*i.e.,* by producing blocks).
- **Light clients.** These are nodes with computational capacity and network bandwidth that is too low to download and verify the entire blockchain. They receive block headers from full nodes, and on request, Merkle proofs that some transaction or state is a part of the block header.

We assume a network topology as shown in Figure 1; full nodes communicate with each other, and light clients communicate with full nodes, but light clients do not communicate with each other. Additionally, we assume a maximum network delay δ; such that if one honest node can connect to the network and download some data (*e.g.,* a block) at time T, then it is guaranteed that any other honest node will be able to do the same at time $T' \leq T + \delta$.

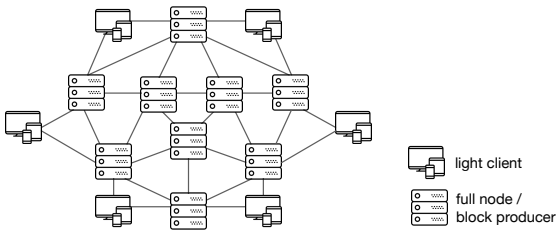


Fig. 1: Network model—full nodes communicate with each other, and light clients communicate only with full nodes.

3.4 Threat Model

We make the following assumptions in our threat model:

- **Blocks and consensus.** Block headers may be created by adversarial actors, and thus may be invalid, and there is no honest majority of consensus-participating nodes that we can rely on.
- **Full nodes.** Full nodes may be dishonest, *e.g.,* they may not relay information (*e.g.,* fraud proofs), or they may relay invalid blocks. However, we assume that there is at least one honest full node that is connected to the network (*i.e.,* it is online, willing to generate and distribute fraud proofs, and is not under an eclipse attack [17]).

– **Light clients.** We assume that each light client is connected to at least one honest full node. For data availability proofs, we assume a minimum number of honest light clients to allow for a block to be reconstructed. The specific number depends on the parameters of the system, and is analysed in Section 5.6.

4 Fraud Proofs

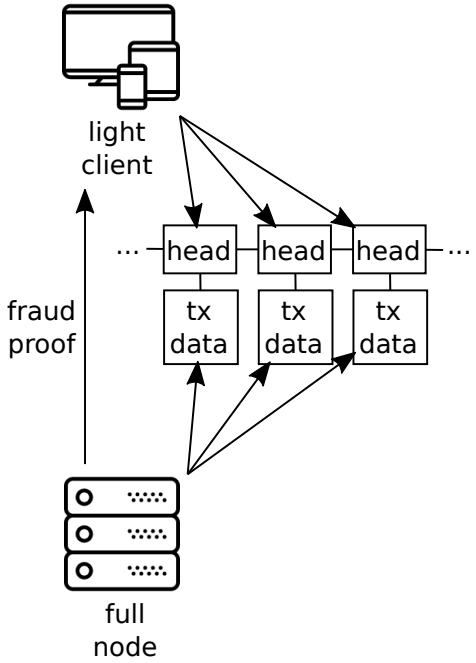


Fig. 2: Overview of the architecture of a fraud proof system at a network level.

4.1 Block Structure

In order to support efficient fraud proofs, it is necessary to design a blockchain data structure that supports fraud proof generation by design. Extending the model described in Section 3.2, a block header h_i at height i contains the following elements:

$\mathsf{prevHash}_i$ The hash of the previous block header in the chain.
$\mathsf{dataRoot}_i$ The root of the Merkle tree of the data (*e.g.,* transactions) included in the block.

$\mathsf{dataLength}_i$ The number of leaves represented by $\mathsf{dataRoot}_i$.

$\mathsf{stateRoot}_i$ The root of a sparse Merkle tree of the state of the blockchain (to be described in Section 4.2).

$\mathsf{additionalData}_i$ Additional arbitrary data that may be required by the network (*e.g.,* in proof-of-work, this may include a nonce and the target difficulty threshold).

Additionally, the hash of each block header $\mathsf{blockHash}_i = \mathsf{hash}(h_i)$ is also stored by clients and nodes.

Note that typically blockchains have the Merkle root of transactions included in headers. We have abstracted this to a 'Merkle root of data' called $\mathsf{dataRoot}_i$, because as we shall see in Section 4.3, as well as including transactions in the block data, we also need to include intermediate state roots.

4.2 State Root and Execution Trace Construction

To instantiate a blockchain based on the state-based model described in Section 3.2, we make use of sparse Merkle trees, and represent the state as a key-value map. We explain how both a UTXO-based and an account-based blockchain can be instantiated atop such a model:

- **UTXO-based.** The keys in the map are transaction output identifiers *e.g.,* $\mathsf{hash}(\mathsf{hash}(d)||i)$ where d is the data of the transaction and i is the index of the output being referred to in d. The value of each key is the state of each transaction output identifier: either *unspent* (1) or *nonexistent* (0, the default value).
- **Account-based.** This is already a key-value map, where the key is the account or storage variable, and the value is the balance of the account or the value of the variable.

The state would need to keep track of all data that is relevant to block processing, including for example the cumulative transaction fees paid to the creator of the current block after each transaction.

We now define a variation of the function $\mathsf{transition}$ defined in Section 3.2, called $\mathsf{rootTransition}$, that performs transitions without requiring the whole state tree, but only the state root and Merkle proofs of parts of the state tree that the transaction reads or modifies (which we call "witness", or w for short). These Merkle proofs are effectively expressed as a sub-tree of the same state tree with a common root.

$$\mathsf{rootTransition}(\mathsf{stateRoot}, t, w) \in \{\mathsf{stateRoot}', err\}$$

A witness w consists of a set of key-value pairs and their associated Sparse Merkle proofs in the state tree, $w = \{(k_1, v_1, \{k_1, v_1 \rightarrow stateRoot\}), (k_2, v_2, \{k_2, v_2 \rightarrow stateRoot\}), ..., (k_n, v_n, \{k_n, v_n \rightarrow stateRoot\})\}$.

After executing t on the parts of the state shown by w, if t modifies any of the state, then the new resulting $\mathsf{stateRoot}'$ can be generated by computing the

root of the new sub-tree with the modified leafs. Not that if w is invalid and does not contain all of the parts of the state required by t during execution, then err is returned.

Let us denote, for the list of transactions $T_i = (t_i^0, t_i^1, ..., t_i^n)$, where t_i^j denotes a transaction j at block i, then w_i^j is the witness for transaction w_i^j for $\mathsf{stateRoot}_i$.

Thus given the intermediate state roots after applying every transaction one at a time, $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$, and the base case $\mathsf{interRoot}_i^{-1} = \mathsf{stateRoot}_{i-1}$, then $\mathsf{stateRoot}_i = \mathsf{interRoot}_i^n$. Hence, $\mathsf{interRoot}_i^j$ denotes the intermediate state root at block i after applying transactions $t_i^0, t_i^1, ..., t_i^j$.

4.3 Data Root and Periods

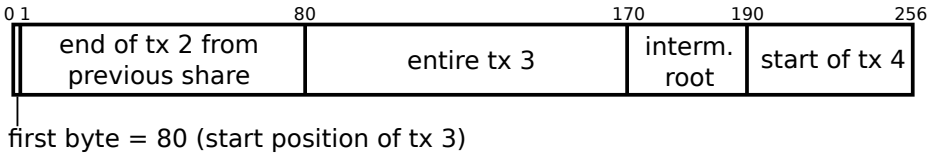


Fig. 3: Example of a 256-byte share.

The data represented by the $\mathsf{dataRoot}_i$ of a block contains transactions arranged into fixed-size chunks of data called 'shares', interspersed with intermediate state roots called 'traces' between transactions. We denote $trace_i^j$ as the jth intermediate state root in block i. It is necessary to arrange data into fixed-size shares to allow for data availability proofs as we shall see in Section 5.

As a share may not contain entire transactions but only parts of transactions as shown in Figure 3, we may reserve the first byte in each share to be the starting position of the first transaction that starts in the share, or 0 if no transaction starts in the share. This allows a protocol message parser to establish the message boundaries without needing every transaction in the block.

Given a list of shares $(sh_0, sh_1, ..., sh_n)$ we define a function $\mathsf{parseShares}$ which parses these shares and outputs an ordered list of t messages $(m_0, m_1, ..., m_t)$, which are either transactions or intermediate state roots. For example, $\mathsf{parseShares}$ on some shares in the middle of some block i may return $(trace_i^1, t_i^4, t_i^5, t_i^6, trace_i^2)$.

$$\mathsf{parseShares}((sh_0, sh_1, ..., sh_n)) = (m_0, m_1, ..., m_t)$$

Note that as the block data does not necessarily contain an intermediate state root after every transaction, we assume a 'period criterion', a protocol rule that defines how often an intermediate state root should be included in the block's data. For example, the rule could be at least once every p transactions, or b bytes or g gas (*i.e.,* in Ethereum [40]).

We thus define a function $\mathsf{parsePeriod}$ which parses a list of messages, and returns a pre-state intermediate root $trace_x$, a post-state intermediate root

$trace_{x+1}$, and a list of transaction $(t_i^g, t_i^{g+1}, ..., t_i^{g+h})$ such that applying these transactions on $trace_x$ is expected to return $trace_{x+1}$. If the list of messages violate the period criterion, then the function may return err, for example if there too many transactions in the messages to constitute a period.

$$\mathsf{parsePeriod}((m_0, m_1, ..., m_t)) = \{(trace_x, trace_{x+1}, (t_i^g, t_i^{g+1}, ..., t_i^{g+h})), err\}$$

Note that $trace_x$ may be nil if no pre-state root was parsed, as this may be the case if the first messages in the block are being parsed, and thus the pre-state root is the state root of the previous block $\mathsf{stateRoot}_{i-i}$. Likewise, $trace_{x+1}$ may be nil if no post-state root was parsed *i.e.,* if the last messages in the block are being parsed, as the post-state root would be $\mathsf{stateRoot}_i$.

4.4 Proof of Invalid State Transition

A faulty or malicious miner may provide an incorrect $\mathsf{stateRoot}_i$. We can use the execution trace provided in $\mathsf{dataRoot}_i$ to prove that some part of the execution trace was invalid.

We define a function VerifyTransitionFraudProof and its parameters which verifies fraud proofs received from full nodes. If the fraud proof is valid, then the block that the fraud proof is for is permanently rejected by the client. In summary, the fraud proof verifier checks if applying the transactions in a period of the block's data on the intermediate pre-state root results in the intermediate post-state root specified the block data. If it does not, then the fraud proof is valid.

We denote d_i^j as share number j in block i.

$$\begin{aligned}
&\mathsf{VerifyTransitionFraudProof}(\mathsf{blockHash}_i,\\
&\quad (d_i^y, d_i^{y+1}, ..., d_i^{y+m}), y, && \text{(shares)}\\
&\quad (\{d_i^y \rightarrow \mathsf{dataRoot}_i\}, \{d_i^{y+1} \rightarrow \mathsf{dataRoot}_i\}}, ..., \{d_i^{y+m} \rightarrow \mathsf{dataRoot}_i\}\}),\\
&\quad (w_i^y, w_i^{y+1}, ..., w_i^{y+m}), && \text{(tx witnesses)}\\
&) \in \{true, false\}
\end{aligned}$$

VerifyTransitionFraudProof returns $true$ if all of the following conditions are met, otherwise $false$ is returned:

1. $blockHash_i$ corresponds to a block header h_i that the client has downloaded and stored.
2. For each share d_i^{y+a} in the proof, $\mathsf{VerifyMerkleProof}(d_i^{y+a}, \{d_i^{y+a} \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, \mathsf{dataLength}_i, y+a)$ returns $true$.
3. Given $\mathsf{parsePeriod}(\mathsf{parseShares}((d_i^y, d_i^{y+1}, ..., d_i^{y+m}))) \in \{(trace_x, trace_{x+1}, (t_i^g, t_i^{g+1}, ..., t_i^{g+h})), err\}$, the result must not be err. If $trace_x$ is nil, then $y = 0$ is true, and if $trace_{x+1}$ is nil, then $y + m = dataLength$ is true.

4. Check that applying $(t_i^g, t_i^{g+1}, ..., t_i^{g+h})$ on $trace_x$ results in $trace_{x+1}$. Formally, let the intermediate state roots after applying every transaction in the proof one at a time be $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$. If trace_x is not nil, then the base case is $\mathsf{interRoot}_i^y = \mathsf{trace}_x$, otherwise $\mathsf{interRoot}_i^y = \mathsf{stateRoot}_{i-1}$. If trace_{x+1} is not nil, $\mathsf{trace}_{x+1} = \mathsf{interRoot}_i^{g+h}$ is true, otherwise $\mathsf{stateRoot}_i = \mathsf{interRoot}_i^{y+m}$ is true.

4.5 Transaction Fees

As discussed in Section 4.2, the state would need to keep track of all data that is relevant to block processing. A block producer may attempt to collect more transaction fees than is afforded to them by the transactions in the block. In order to make this detectable by a fraud proof as part of the model we have described, we can introduce a special key in the state tree called $\_\_fees\_\_$, which represents the cumulative fees in the block after applying each transaction, and is reset to 0 after applying the transaction where the block producer collects the fees.

5 Data Availability Proofs

A malicious block producer could prevent full nodes from generating fraud proofs by withholding the data needed to recompute $\mathsf{dataRoot}_i$ and only releasing the block header to the network. The block producer could then only release the data—which may contain invalid transactions or state transitions—long after the block has been published, and make the block invalid. This would cause a rollback of transactions on the ledger of future blocks. It is therefore necessary for light clients to have a level of assurance that the data matching $\mathsf{dataRoot}_i$ is indeed available to the network.

We propose a data availability scheme based on Reed-Solomon erasure coding, where light clients request random shares of data to get high probability guarantees that all the data associated with the root of a Merkle tree is available. The scheme assumes there is a sufficient number of honest light clients making the same requests such that the network can recover the data, as light clients upload these shares to full nodes, if a full node who does not have the complete data requests it. It is fundamental for light clients to have assurance that all the transaction data is available, because it is only necessary to withhold a few bytes to hide an invalid transaction in a block.

We define below *soundness* and *agreement* and analyse them in **??**.

Definition 1 (Soundness). *If an honest light client accepts a block as available, then at least one honest full node has the full block data or will have the full block data within some known maximum delay $k * \delta$ where δ is the maximum network delay.*

Definition 2 (Agreement). *If an honest light client accepts a block as available, then all other honest light clients will accept that block as available within some known maximum delay $k * \delta$ where δ is the maximum network delay.*

5.1 Strawman 1D Reed-Solomon Availability Scheme

To provide some intuition, we first describe a strawman data availability scheme, based on standard Reed-Solomon coding.

A block producer compiles a block of data consisting of k shares, extends the data to $2k$ shares using Reed-Solomon encoding, and computes a Merkle root (the $\mathsf{dataRoot}_i$) over the extended data, where each leaf corresponds to one share.

When light clients receive a block header with this $\mathsf{dataRoot}_i$, they randomly sample shares from the Merkle tree that $\mathsf{dataRoot}_i$ represents, and only accept a block once it has received all of the shares requested. If an adversarial block producer makes more than 50% of the shares unavailable to make the full data unrecoverable (recall in recalled in Section 2.3 that Reed-Solomon codes allow recovery of $2t$ shares from any t shares), there is a 50% chance that a client will randomly sample an unavailable share in the first draw, a 25% chance after two draws, a 12.5% chance after three draws, and so on, if they draw with replacement. (In the full scheme, they will draw without replacement, and so the probability will be even lower.)

Note that for this scheme to work, there must be enough light clients in the network sampling enough shares so that block producers will be required to release more than 50% of the shares in order to pass the sampling challenge of all light clients, and so that the full block can be recovered. An in-depth probability and security analysis is provided in Section 5.6.

The problem with this scheme is that an adversarial block producer may incorrectly construct the extended data, and thus the incomplete block is unrecoverable from the extended data even if more than 50% of the data is available. With standard Reed-Solomon encoding, the fraud proof that the extended data is invalid is the original data itself, as clients would have to re-encode all data locally to verify the mismatch with the given extended data, and thus it requires $O(n)$ data with respect to the size of the block. Therefore, we instead use multi-dimensional encoding, as described in Section 5.2, so that proofs of incorrectly generated codes are limited to a specific axis—rather than the entire data—reducing proof size to $O(\sqrt[d]{n})$ where d is the number of dimensions of the encoding. For simplicity, we will only consider two-dimensional Reed-Solomon encoding in this paper, but our scheme can be generalised to higher dimensions.

We note in Section 7.2 that succinct proofs of computation could be an alternative future solution to this problem instead of multi-dimensional encoding.

5.2 2D Reed-Solomon Encoded Merkle Tree Construction

A 2D Reed-Solomon Encoded Merkle tree can be constructed as follows from a block of data:

1. Split the raw data into shares of size $shareSize$ each, and arrange them into a $k \times k$ matrix; apply padding if the last share is not exactly of size $shareSize$, or if there are not enough shares to complete the matrix.

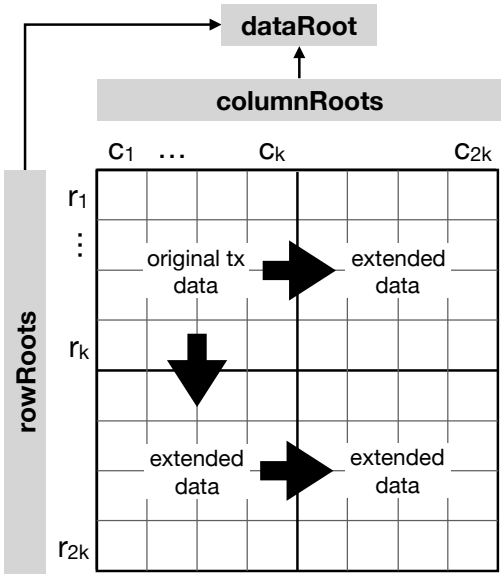


Fig. 4: Diagram showing a 2D Reed-Solomon encoding. The original data is initially arranged in a $k \times k$ matrix, which is then 'extended' to a $2k \times 2k$ matrix applying multiple times Reed-Solomon encoding.

2. Apply Reed-Solomon encoding on each row and column of the $k \times k$ matrix to extend the data horizontally and vertically; *i.e.,* encode each row and each column. Then apply a third time a Reed-Solomon encoding horizontally, on the vertically extended portion of the matrix to create a $2k \times 2k$ matrix, as shown in Figure 4. This results in an extended matrix M_i for block i.
3. Compute the root of the Merkle tree for each row and column in the $2k \times 2k$ matrix, where each leaf is a share. We have $\mathsf{rowRoot}_i^j = \mathsf{root}((M_i^{j,1}, M_i^{j,2}, ..., M_i^{j,2k}))$ and $\mathsf{columnRoot}_i^j = \mathsf{root}((M_i^{1,j}, M_i^{2,j}, ..., M_i^{2k,j}))$, where $M_i^{x,y}$ represents the share in row x, column y in the matrix.
4. Compute the root of the Merkle tree of the roots computed in step 3 and use this as $\mathsf{dataRoot}_i$. We have $\mathsf{dataRoot}_i = \mathsf{root}((\mathsf{rowRoot}_i^1, \mathsf{rowRoot}_i^2, ..., \mathsf{rowRoot}_i^{2k}, \mathsf{columnRoot}_i^1, \mathsf{columnRoot}_i^2, ..., \mathsf{columnRoot}_i^{2k}))$.

The resulting tree of $\mathsf{dataRoot}_i$ has $\mathsf{dataLength}_i = 2 \times (2k)^2$ elements, where the first $\frac{1}{2}\mathsf{dataLength}_i$ elements are in leaves via the row roots, and the latter half are in leaves via the column roots.

Note that although it is possible to present a Merkle proof from $\mathsf{dataRoot}_i$ to an individual share, it is important to note that a Merkle tree has 2^x leaves, and the Merkle sub-trees for the row and column roots are constructed independently from $\mathsf{dataRoot}_i$. Therefore it is necessary to have a wrapper function around VerifyMerkleProof called VerifyShareMerkleProof with the same parameters which takes into account how the underlying Merkle tree deals with an unbalanced

number of leaves; this may involve calling VerifyMerkleProof twice for different portions of the path, or offsetting the index.[4]

The width of the matrix can be derived as $\mathsf{matrixWidth}_i = \sqrt{\frac{1}{2}dataLength_i}$. If we are only interested in the row and column roots of $\mathsf{dataRoot}_i$, rather than the actual shares, then we can assume that $\mathsf{dataRoot}_i$ has $2 \times \mathsf{matrixWidth}_i$ leaves when verifying a Merkle proof of a row or column root.

A light client or full node is able to reconstruct $\mathsf{dataRoot}_i$ from all the row and column roots by recomputing step 4. In order to gain data availability assurances, all light clients should at minimum download all the row and column roots needed to reconstruct $\mathsf{dataRoot}_i$ and check that step 4 was computed correctly, because as we shall see in Section 5.5, they are necessary to generate fraud proofs of incorrectly generated extended data.

We nevertheless represent all of the row and column roots as a a single $\mathsf{dataRoot}_i$ to allow 'super-light' clients which do not download the row and column roots, but these clients cannot be assured of data availability and thus do not fully benefit from the increased security of allowing fraud proofs.

5.3 Random Sampling and Network Block Recovery

In order for any share in the 2D Reed-Solomon matrix to be unrecoverable, then at least $(k+1)^2$ out of $2k$ shares must be unavailable (see Theorem 1). Thus when light clients receive a new block header from the network, they should randomly sample $0 < s < (k+1)^2$ distinct shares from the extended matrix, and only accept the block if they receive all shares. Additionally, light clients gossip shares that they have received to the network, so that the full block can be recovered by honest full nodes.

The protocol between a light client and the full nodes that it is connected to works as follows:

1. The light client receives a new block header h_i from one of the full nodes it is connected to, and a set of row and column roots $R = (\mathsf{rowRoot}_i^1, \mathsf{rowRoot}_i^2, ..., \mathsf{rowRoot}_i^{2k}, \mathsf{columnRoot}_i^1, \mathsf{columnRoot}_i^2, ..., \mathsf{columnRoot}_i^{2k})$. If the check $\mathsf{root}(R) = \mathsf{dataRoot}_i$ is false, then the light client rejects the header.
2. The light client randomly chooses a set of unique (x, y) coordinates $S = \{(x_0, y_0)(x_1, y_1), ..., (x_n, y_n)\}$ where $0 < x < \frac{1}{2}dataLength$ and $0 < y < \frac{1}{2}dataLength$, corresponding to points on the extended matrix, and sends them to one or more of the full nodes it is connected to.
3. If a full node has all of the shares corresponding to the coordinates in S and their associated Merkle proofs, then for each coordinate (x_a, y_b) the full node responds with $M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{rowRoot}_i^a\}$ or $M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{columnRoot}_i^b\}$. Note that there are two possible Merkle proofs for each share; one from the row roots, and one from the column roots, and thus the full

[4] For example, if the underlying tree simply repeats the last leaves to pad the tree to 2^x leaves, then the wrapper function may be $\mathsf{VerifyShareMerkleProof}(e, \{e \rightarrow r\}, r, n, i) = \mathsf{VerifyMerkleProof}(e, \{e \rightarrow r\}, r, n, i + \lfloor i \,/\, \sqrt{\frac{1}{2}i} \rfloor \times (2^{\lceil \log_2(\sqrt{\frac{1}{2}i}) \rceil} - \sqrt{\frac{1}{2}i}))$.

node must also specify for each Merkle proof if it is associated with a row or column root.

4. For each share $M_i^{x_a,y_b}$ that the light client has received, the light client checks that $VerifyMerkleProof(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{rowRoot}_i^a\}, \mathsf{rowRoot}_i^a, \mathsf{matrixWidth}_i, b)$ is *true* if the proof is from a row root, otherwise if the proof is from a column root then $VerifyMerkleProof(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{columnRoot}_i^b\}, \mathsf{columnRoot}_i^b, \mathsf{matrixWidth}_i, a)$ is *true*.
5. Each share and valid Merkle proof that is received by the light client is gossiped to all the full nodes that the light client is connected to if the full nodes do not have them, and those full nodes gossip it to all of the full nodes that they are connected to.
6. If all the proofs in step 4 succeeded, and no shares are missing from the sample made in step 2, then the block is accepted as available if within $2 \times \delta$ no fraud proofs for the block's erasure code is received (Section 5.5).

5.4 Selective Share Disclosure

If a block producer selectively releases shares as light clients ask for them, up to $(k+1)^2$ shares, they can violate the soundness property (Definition 1) of the clients that ask for the first $(k+1)^2$ out of $2k$ shares, as they will accept the blocks as available despite them being unrecoverable.

This can be alleviated if one assumes an enhanced network model where a sufficient number of honest light clients make requests such that more than $(k+1)^2$ shares will be sampled, and that each sample request for each share is anonymous (*i.e.,* sample requests cannot be linked to the same client) and the distribution in which every sample request is received is uniformly random, for example by using a mix net [8]. As the network would not be able to link different per-share sample requests to the same clients, shares cannot be selectively released on a per-client basis.

We thus assume two network connection models that sample requests can be made under, which we will analyse in the security analysis:

- **Standard model.** Sample requests are linkable to the clients that made them, and the order that they are received is predictable (*e.g.,* they are received in the order that they were sent).
- **Enhanced model.** Different sample requests cannot be linked to the same client, and the order that they are received by the network is uniformly random with respect to other requests.

5.5 Fraud Proofs of Incorrectly Generated Extended Data

If a full node has enough shares to recover a particular row or column, and after doing so detects that recovered data does not match its respective row or column root, then it must distribute a fraud proof consisting of enough shares in that row or column to be able to recover it, and a Merkle proof for each share. In summary, the fraud proof verifier checks that (*i*) all of the shares given by the

prover are in the same row or column and (*ii*) that the recovered row or column does not match the row or column root in the block.

We define a function VerifyCodecFraudProof that verifies these fraud proofs, where $axisRoot_i^j \in \{rowRoot_i^j, columnRoot_i^j\}$. These proofs can also be verified by 'super-light' clients as they do not assume any knowledge of the row and column roots. We denote $axis$ and ax_j as row or column boolean indicators; 0 for rows and 1 for columns.

$$
\begin{aligned}
&\mathsf{VerifyCodecFraudProof}(\mathsf{blockHash}_i,\\
&\quad axisRoot_i^j, \{axisRoot_i^j \rightarrow \mathsf{dataRoot}_i\}, j, && \text{(row or column root)}\\
&\quad axis, && \text{(row or column indicator)}\\
&\quad ((sh_0, pos_0, ax_0), (sh_1, pos_1, ax_1), ..., (sh_k, pos_k, ax_k)), && \text{(shares)}\\
&\quad (\{sh_0 \rightarrow dataRoot_i\}, \{sh_1 \rightarrow dataRoot_i\}\}, ..., \{sh_k \rightarrow dataRoot_i\}\})\\
&) \in \{true, false\}
\end{aligned}
$$

Let *recover* be a function that takes a list of shares and their positions in the row or column $((sh_0, pos_0), (sh_1, pos_1), ..., (sh_k, pos_k))$, and the length of the extended row or column $2k$. The function outputs the full recovered shares $(sh_0, sh_1, ..., sh_{2k})$ or *err* if the shares are unrecoverable.

$$recover(((sh_0, pos_0), (sh_1, pos_1), ..., (sh_k, pos_k)), 2k) \in \{(sh_0, sh_1, ..., sh_{2k}), err\}$$

VerifyCodecFraudProof returns true if all of the following conditions are met:

1. $blockHash_i$ corresponds to a block header h_i that the client has downloaded and stored.
2. If $axis = 0$ (row root), $\mathsf{VerifyMerkleProof}(axisRoot_i^j, \{axisRoot_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, j)$ returns *true*.
3. If $axis = 1$ (col. root), $\mathsf{VerifyMerkleProof}(axisRoot_i^j, \{axisRoot_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, \frac{1}{2}\mathsf{dataLength}_i + j)$ returns *true*.
4. For each (sh_x, pos_x, ax_x), $\mathsf{VerifyShareMerkleProof}(sh_x, \{sh_x \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, \mathsf{dataLength}, index)$ returns true, where *index* is the expected index of the sh_x in the data tree based on pos_x assuming it is in the same row or column as $axisRoot_i^j$. See Appendix B for how *index* can be computed.
   Note that full nodes can specify Merkle proofs of shares in rows or columns from either the row or column roots *e.g.,* if a row is invalid but the full nodes only has Merkle proofs for the row's share from column roots. This also allows for full nodes to generate fraud proofs if there are inconsistencies in the data between rows and columns *e.g.,* if the same cell in the matrix has a different share in its row and column trees.
5. $root(recover(((sh_0, pos_0), (sh_1, pos_1), ..., (sh_k, pos_k)))) = axisRoot_i^j$ is false.

If $VerifyCodecFraudProof$ for $blockHash_i$ returns *true*, then the block header h_i is permanently rejected by the light client.

5.6 Sampling Security Analysis

We present how the data availability scheme presented in Section 5 can provide lights clients with a high level of assurance that block data is available to the network.

Minimum Unavailable Shares for Unrecoverability Theorem 1 states that data is unrecoverable if a malicious block proposer withholds $k+1$ shares of at least $k+1$ columns or rows; which makes a total of $(k+1)^2$ shares to withhold.

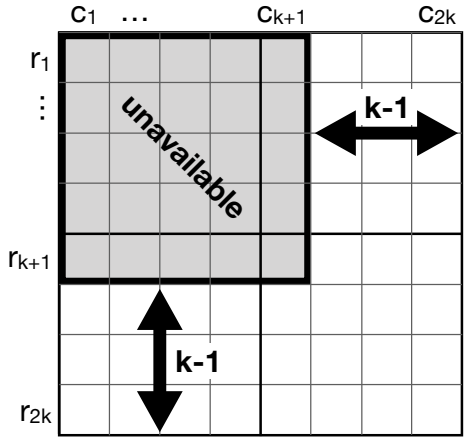


Fig. 5: Graphical interpretation of Theorem 1. Data is unrecoverable if at least $k+1$ columns (or rows) have each at least $k+1$ unavailable shares.

Theorem 1. *Given a $2k \times 2k$ matrix E as show in Figure 4, data is unrecoverable if at least $k+1$ columns or rows have each at least $k+1$ unavailable shares. In that case, the minimum number of shares that must be unrecoverable is $(k+1)^2$.*

Proof. Suppose a malicious block producer wants to make unrecoverable a share $E_{i,j}$ of the $2k \times 2k$ matrix E. Recall that Reed-Solomon encoding allow to recover all $2k$ shares from any k shares; the block producer will have to (i) make unrecoverable at least $k+1$ shares from the row $E_{i,*}$, and (ii) make unrecoverable at least $k+1$ shares from the column $E_{*,j}$.

Let us start from (i); the block producer withholds at least $k+1$ shares from row $E_{i,*}$. However, each of these $k+1$ withheld shares $(E_{i,c_1}, \ldots, E_{i,c_{k+1}}) \in E_{i,*}$ can be recovered from the available shares of their respective columns E_{*,c_1}, $E_{*,c_2} \ldots$, $E_{*,c_{k+1}}$. Therefore, the block producer will also have to withhold at least $k+1$ shares from each of these columns. This gives a total of $(k+1) * (k+1) = (k+1)^2$ shares to withhold. Note that at this point, there are not

enough shares left in the matrix to recover any of the $(k+1)^2$ shares of columns $(E_{*,c_1}, \ldots E_{*,c_{k+1}})$.

Let us now consider (*ii*); the block producer withholds at least $k+1$ shares from the column $E_{*,j}$ to make unrecoverable the share $E_{i,j}$. As before, each shares $(E_{r_1,j}, \ldots, E_{r_{k+1},j}) \in E_{*,j}$ can be recovered from the available shares of their respective row $E_{r_1,*}, E_{r_2,*}, \ldots, E_{r_{k+1},*}$. Therefore, the block producer will also have to withhold at least at least $k+1$ shares from each of these rows. As before, this also gives a total of $(k+1) * (k+1) = (k+1)^2$ shares to withhold.

However, (*i*) is equivalent to (*ii*) by the symmetry of the matrix, and are actually operating on the same shares; executing (*i*) on matrix E is equivalent to execute (*ii*) on the transposed of the matrix E.

Unrecoverable Block Detection Theorem 2 states the probability that a single light client will sample at least one unavailable share in a matrix with the minimum unavailable shares for unrecoverability, thus detecting that a block may be unrecoverable.

Theorem 2. *Given a $2k \times 2k$ matrix E as shown in Figure 4, where $(k+1)^2$ shares are unavailable. If one player randomly samples $0 < s < (k+1)^2$ shares from E, the probability of sampling at least one unavailable share is:*

$$p_1(X \geq 1) = 1 - \prod_{i=0}^{s-1} \left(1 - \frac{(k+1)^2}{4k^2 - i}\right) \tag{1}$$

Proof. We start by assuming that the $2k \times 2k$ matrix E contains q unavailable shares; If the player performs m trials ($0 < s < (k+1)^2$), the probability of finding exactly zero unavailable share is:

$$p_1(X = 0) = \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} \tag{2}$$

The numerator of Equation (2) computes the number of ways to pick s chunks among the set of unavailable shares $4k^2 - q$ (i.e., $\binom{4k^2-q}{s}$). The denominator computes the total number of ways to pick any s samples out of the total number of samples (i.e., $\binom{4k^2}{s}$).

Then, the probability $p_1(X \geq 1)$ of finding at least one unavailable share can be easily computed from Equation (2):

$$p_1(X \geq 1) = 1 - p_1(X = 0) \tag{3}$$

$$= 1 - \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} \tag{4}$$

$$= 1 - \prod_{i=0}^{s-1} \left(1 - \frac{q}{4k^2 - i}\right) \tag{5}$$

which can be re-written as Equation (1) by setting $q = (k+1)^2$.

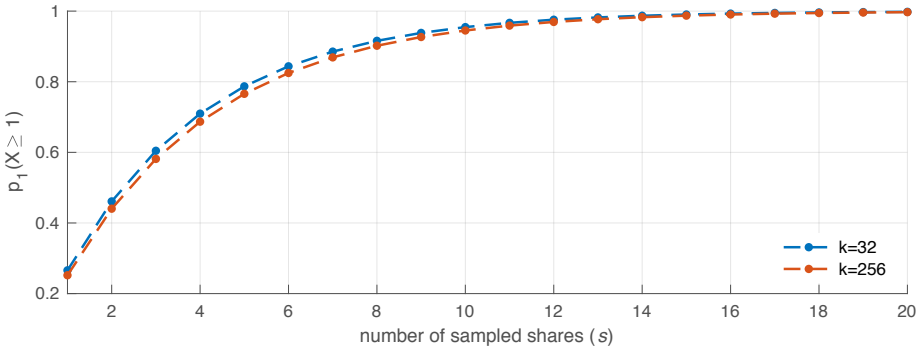


Fig. 6: Plot of Equation (1)—variation of the probability $p_1(X \geq 1)$ with the number of sampled shares (s) (computed for $k = 32$ and $k = 256$).

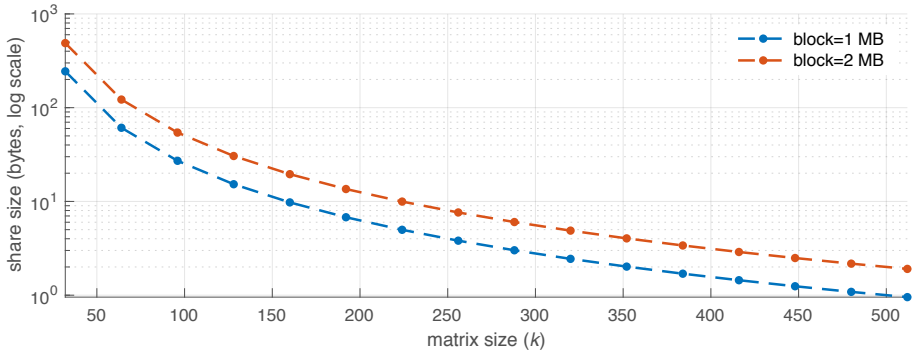


Fig. 7: Variation of the shares size with the size of the matrix (k).

Figure 6 shows how this probability varies with s samples for $k = 32$ and $k = 256$; each light client samples at least one unavailable share with about 60% probability after 3 samplings (*i.e.,* after querying respectively 0.07% of the block shares for $k = 32$ and 0.001% of the block shares for $k = 256$), and with more than 99% probability after 15 samplings (*i.e.,* after querying respectively 0.4% of the block shares for $k = 16$ and 0.005% of the block shares for $k = 256$). Figure 7 shows that light clients would have to download about 3.6 KB of shares to be able to detect incomplete blocks with more than 99% probability for $k = 32$, and about 57 bytes of shares for $k = 256$.

Equation (6) shows a noticeable result: the probability $p_1(X \geq 1)$ is almost independent of k for large values of k; it is therefore convenient to have a large matrix size (*i.e.,* $k \geq 128$) as this reduces the amount of data that light clients have to download.

$$\lim_{k\to\infty} p_1(X \geq 1) = \lim_{k\to\infty} \left(1 - \prod_{i=0}^{s-1} \left(1 - \frac{(k+1)^2}{4k^2 - i}\right)\right) = 1 - (3/4)^s \tag{6}$$

Under the enhanced model described in Section 5.4, a malicious block producer could statistically link light clients based on the shares they query; *i.e.,* assuming that a light client would never request twice the same share, a block producer can deduce that any request for the same share comes from a different client. To mitigate this problem, light clients could sample without replacement by performing the procedure for sampling with replacement multiple times, and only stop when they have sampled s unique values.

Multi-Client Unrecoverable Block Detection Theorem 3 captures the probability that more than $\hat{c}$ out of c light clients sample at least one unavailable share in a matrix with the minimum unavailable shares for unrecoverability.

Theorem 3. *Given a $2k \times 2k$ matrix E as shown in Figure 4, where $(k+1)^2$ shares are unavailable. If c players randomly sample $0 < s < (k+1)^2$ shares from E, the probability that more than $\hat{c}$ players sample at least one unavailable share is:*

$$p_c(Y > \hat{c}) = 1 - \sum_{j=1}^{\hat{c}} \binom{c}{j} \big(p_1(X \geq 1)\big)^j \big(1 - p_1(X \geq 1)\big)^{c-j} \tag{7}$$

where $p_1(X \geq 1)$ is given by Equation (1).

Proof. We start by computing the probability that exactly $\hat{c}$ players sample at least one unavailable share; this probability is given by the binomial probability mass function:

$$p_{s,\hat{c}}(Y = \hat{c}) = \binom{c}{\hat{c}} \big(p_1(X \geq 1)\big)^{\hat{c}} \big(1 - p_1(X \geq 1)\big)^{c-\hat{c}} \tag{8}$$

where $p_1(X \geq 1)$ is given by Equation (1). Equation (8) describes the probability that $\hat{c}$ players succeed to sample at least one unavailable share. This can be

viewed as the probability of observing $\hat{c}$ successes each happening with probability p_1, and $(c - \hat{c})$ failures each happening with probability $1 - p_1$; there are $\binom{c}{\hat{c}}$ possible ways of sequencing these successes and failures.

Equation (8) easily generalises to the binomial cumulative distribution function expressed in Equation (9)—the probability of observing at most $\hat{c}$ successes is the sum of the probabilities of observing j successes for $j = 1, \ldots, \hat{c}$.

$$p_c(Y \leq \hat{c}) = \sum_{j=1}^{\hat{c}} \binom{c}{j} \big(p_1(X \geq 1)\big)^j \big(1 - p_1(X \geq 1)\big)^{c-j} \tag{9}$$

Therefore the probability of observing more than $\hat{c}$ successes is given by Equation (10) below, which expands as Equation (7).

$$p_c(Y > \hat{c}) = 1 - p_c(Y \leq \hat{c}) \tag{10}$$

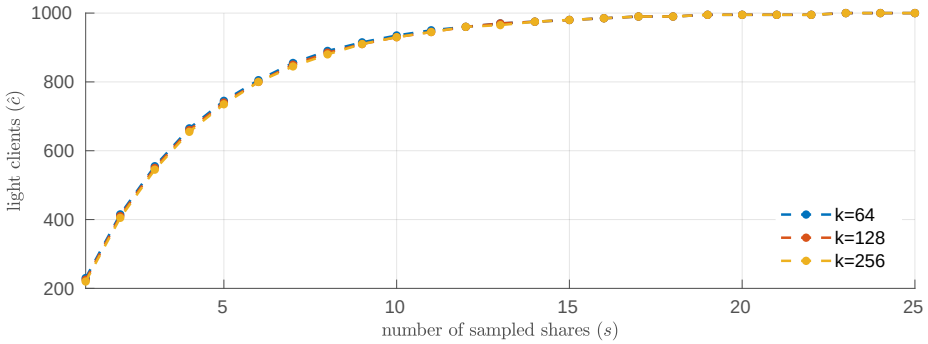


Fig. 8: Plot of Equation (7)—variation of the number of light clients $\hat{c}$ for which $p_c(Y > \hat{c}) \geq 0.99$ with the sampling size s. The total number of clients is fixed to $c = 1000$, and the matrix sizes are $k = 64, 128, 256$; Equation (7) is however almost independent of k, as indicated by Equation (6).

Figure 8 shows the variation of the number of light clients $\hat{c}$ for which $p_c(Y > \hat{c}) \geq 0.99$ with the sampling size s. The total number of clients is fixed to $c = 1000$, and the matrix sizes are $k = 64, 128, 256$; Equation (7) is however almost independent of k, as indicated by Equation (6). This figure can be used to determine the number of light clients that will detect incomplete matrices with high probability ($p_c(Y > \hat{c}) \geq 0.99$), and that there is little gain in increasing s over 15.

Recovery and Selective Share Disclosure Corollary 1 presents the probability that light clients collectively samples enough shares to recover every share of the $2k \times 2k$ matrix.

If the light clients collectively sample all but $(k+1)^2$ distinct shares, the block producer cannot release any more shares without allowing the network to recover the whole matrix; it follows from Theorem 1 that light clients need to collect at least:

$$\gamma = (2k)^2 - (k+1)^2 + 1 = k(3k-2)$$

distinct shares (randomly chosen) to have the certainty to be able to recover the $2k \times 2k$ matrix. We are therefore interested in the probability that light clients—each sampling s distinct shares—collectively samples at least γ distinct shares; this probability is expressed by Corollary 1.

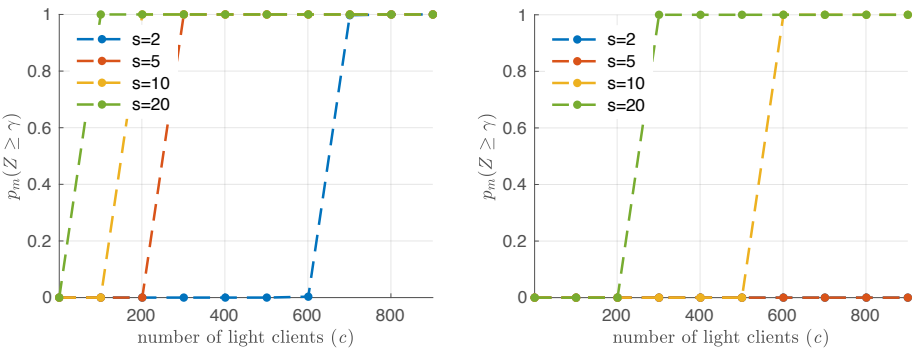
Theorem 4. *(Euler [14]) the probability that the number of distinct elements sampled from a set of n elements, after c drawings with replacement of s distinct elements each, is at least all but λ elements*[5]*:*

$$p_e(Z \geq n - \lambda) = 1 - \sum_{i=1}^{\infty} (-1)^i \binom{\lambda + i - 1}{\lambda} \binom{n}{\lambda + i} (W_i)^c \tag{11}$$

$$\textit{where} \quad W_i = \binom{n - \lambda - i}{s} \Big/ \binom{n}{s}$$

Corollary 1. *Given a $2k \times 2k$ matrix E as shown in Figure 4, where each of c players randomly samples s distinct shares from E. The probability that the players collectively sample at least $\gamma = k(3k-2)$ distinct shares is $p_e(Z \geq \gamma)$*

Proof. Corollary 1 can be easily proven by substituting $\lambda = n - \gamma$ and $n = (2k)^2$ into Theorem 4.



(a) Matrix size $k = 16$ (b) Matrix size $k = 32$

Fig. 9: Plot of Corollary 1—variation of the probability $p_e(Z \geq \gamma)$ with the number of clients (c) for different values of s and k.

Contrarily to Equation (7), Figure 9 shows that $p_e(Z \geq \gamma)$ depends on the matrix size k.

[5] This problem is also known as *the coupon collector's problem* with group drawing [15].

$p_e(Z \geq \gamma)$	$s = 2$	$s = 5$	$s = 10$	$s = 20$	$s = 50$
$k = 16$	692	277	138	69	28
$k = 32$	2805	1,122	561	280	112
$k = 64$	11,289	4,516	2,258	1,129	451
$k = 128$	>40,000	∼18,000	∼9,000	∼4,500	1,811

Table 1: Minimum number of light clients (c) required to achieve $p_e(Z \geq \gamma) > 0.99$ for various values of k and s. The approximate values have been approached numerically as evaluating Theorem 4 can be extremely resource-intensive for large values of k.

5.7 Properties Security Analysis

Standard Model

Corollary 2. *Under the standard model, a block producer cannot cause soundness (Definition 1) and agreement (Definition 2) to fail for more than c honest clients with a probability lower than $p_1(X \geq 1)$ per client, where c is determined by the probability distribution $p_e(Z \geq \gamma)$.*

Proof. Corollary 1 shows that with probability $p_e(Z \geq \gamma)$, c honest clients will sample enough shares to collectively recover the full block. Honest clients will gossip these shares to full nodes which then gossip them to each other, and within $k \times \delta$ at least one honest full node will then recover the full block data, thus satisfying soundness with a probability of $1 - p_1(X \geq 1)$ per client (the probability of the block producer not passing the client's random sampling challenge when all the block data is available).

If the data is available and no fraud proofs of incorrectly generated extended data was received by the client, then no other client will receive a fraud proof either, due to our assumption that there is at least one honest full node in the network and honest light clients are not under an eclipse attack, thus satisfying agreement with a probability of $1 - p_1(X \geq 1)$ per client.

Due to the selective share disclosure attack described in Section 5.4, this means that the block producer can violate soundness and agreement of the first c clients that make sample requests, as the block producer can stop releasing shares just before it is about to release the final shares to allow the block to be recoverable.

Enhanced Model

Corollary 3. *Under the enhanced model, a block producer cannot cause soundness (Definition 1) and agreement (Definition 2) to fail with a probability lower than $p_x(X \geq 1)$ per client,*

$$p_x(X \geq 1) = \sum_{i=1}^{d} \frac{\binom{s}{i}\binom{s(c-1)}{d-i}}{\binom{c \cdot s}{d}} \tag{12}$$

where c is the number of clients and d is the number of requests that the block producer must deny to prevent full nodes from recovering the data.

Proof. The proof of Corollary 3 starts as the proof of Corollary 2; honest light clients collectively samples enough shares to recover the full block data by gossiping these shares to full nodes; soundness is satisfied with probability $1 - p_1(X \geq 1)$ per client. None of the light clients receive fraud proofs if the full data is available and no valid fraud proofs are sent over the network, and all light clients eventually receive a valid fraud proof if one is sent, satisfying agreement with the same probability.

However, the enhanced model assumes that all sample requests come through a perfect mix network (*i.e.,* requests are unlinkable between each other), and defeats the selective shares disclosure attack presented in Section 5.4. The enhanced model removes the notion of 'first' clients described in Corollary 2 as block producers cannot distinguish which requests comes from which client (since requests are unlikable). Furthermore, if block producers randomly deny some requests, light clients would uniformly see some of their sample requests denied, and each light client would therefore consider the block invalid with equal probability.

Particularly, if c light clients each sample $0 < s < (k+1)^2$ shares, block producers observe a total of $(c \cdot s)$ indistinguishable requests. Let us assume that a malicious block producer must deny at least d request to prevent full nodes from recovering the block data. The probability that a light client observes at least one of its requests denied (and thus rejects the block) is given by $p_x(X \geq 1)$ in Equation (12). The numerator of Equation (12) computes the number of ways of picking i of the denied requests among the s requests sent by the light client (i.e., $\binom{s}{i}$), multiplied by the number of ways to pick the remaining $d-i$ requests among the set of requests sent by other light clients: $c \cdot s - s = s(c-1)$ (i.e., $\binom{s(c-1)}{d-i}$). The denominator computes the total number of ways to pick any d requests out of the total number of requests (i.e., $\binom{c \cdot s}{d}$). The probability that at least one of the denied requests comes from a particular client is the sum of the probabilities for $i = 1, \ldots, d$.

Like Equation (1), Equation (12) rapidly grows and shows that light clients reject the block if invalid (for appropriate values of d). The value of d can be approximated using Corollary 1, and depends on s and c. To provide a quick intuition, if we assume that the light clients collectively sample at least once every share of the block, a malicious block producer must deny at least $(k+1)^2$ requests on different shares to prevent full nodes from recovering the block data; since multiple requests can sample the same shares, $d \geq (k+1)^2$.

6 Performance and Implementation

We implemented the data availability proof scheme described in Section 5 and a prototype of the state transition fraud proof scheme described in Section 4 in

Object	Worst case space complexity
State fraud proof	$O(p + p\log(d) + w\log(s) + w)$
Availability fraud proof	$O(d^{0.5} + d^{0.5}\log(d^{0.5}))$
Single sample response	$O(shareSize + \log(d))$
Block header with axis roots	$O(d^{0.5})$
Block header excl. axis roots	$O(1)$

Table 2: Worst case space complexity for various objects. p represents the number of transactions in a period, w represents the number of witnesses for those transactions, d is short for `dataLength`, and s is the number of key-value pairs in the state tree.

2,683 lines of Go code and released the code as a series of free and open-source libraries.[6]

We first evaluate the space and time complexity of the scheme in Section 6.1 and then present the performance benchmarks of our implementation in Section 6.2. We perform the measurements on a laptop with an Intel Core i5 1.3GHz processor and 16GB of RAM, and use SHA-256 for hashing.

6.1 Space and Time Complexity

Table 2 shows the space complexity for different objects. We observe that the size of the state transition fraud proofs only grows logarithmically with the size of the block and state, whereas the availability fraud proofs (as well as block headers with the axis roots) grows at least in proportion to the square root of the size of the block.

Table 3 shows the time complexity for various actions. For generating and verifying fraud proofs, we note that generating and verifying Merkle proofs for witnesses is $O(1)$ as a sparse Merkle tree has a static depth. The most expensive operation is generating availability fraud proofs, as Lagrange interpolation takes $O(k^2)$ time to encode or decode a row/column with k shares, but this can be reduced to $O(k\log(k))$ time with algorithms based on Fast Fourier Transforms (FFT) [21, 32].

6.2 Benchmarks

Table 4 shows the size of various objects when transmitted over the network. We observe that the size of the state fraud proof only increases logarithmically

[6] 2D Reed-Solomon Merkle tree data availability scheme: `https://github.com/musalbas/rsmt2d`
State transition fraud proofs prototype: `https://github.com/asonnino/fraudproofs-prototype`
Sparse Merkle tree library: `https://github.com/musalbas/smt`

Action	Worst case time complexity
[G] State fraud proof	$O(b + p\log(d) + w)$
[V] State fraud proof	$O(p + p\log(d) + w)$
[G] Availability fraud proof	$O(d^2 + d^{0.5}\log(d^{0.5}))$
[V] Availability fraud proof	$O(d + d^{0.5}\log(d^{0.5}))$
[G] Availability fraud proof (FFTs)	$O(d \times d^{0.5}\log(d^{0.5}))$
[V] Availability fraud proof (FFTs)	$O(d^{0.5}\log(d^{0.5}))$
[G] Single sample response	$O(log(d^{0.5}))$
[V] Single sample response	$O(log(d^{0.5}))$

Table 3: Worst case time complexity for various actions, where [G] means generate and [V] means verify. p represents the number of transactions in a period, b represents the number of transactions in the block, w represents the number of witnesses for those transactions, d is short for dataLength, and s is the number of key-value pairs in the state tree. For generating and verifying state fraud proofs, we assume that each transaction takes the same amount of time to process. For generating fraud proofs, we also include the cost of verifying the block itself.

Object (10 tx/period)	Size (250KB block)	Size (1MB block)
State fraud proof	14,090b	14,410b
Availability fraud proof	5,120b	12,288b
Single sample response	320b	368b
Block header with. axis roots	2,176b	4,224b
Block header excl. axis roots	128b	128b

Table 4: Illustrative sizes for objects for 250KB and 1MB blocks, assuming that a period consists of 10 transactions, the average transaction size is 225 bytes, and that conservatively there are 2^{30} non-default nodes in the state tree.

with the size of the block; this is because the number of transactions in a period remains static, but the size of the Merkle proof for each transactions increases slightly. Block size impacts the size of availability fraud proofs and the axis roots the most, as the size of a single row or column is proportional to the square root of the size of the block.

Table 5 shows the computation time for generating and verifying various objects; the benchmark for state fraud proof generation includes time spent verifying the block. Although verification is linear in the size of the block, in our implementation it has a high constant factor due to the need for 256 hash operations per update in the tree. This can be improved by using a SHA-256 library that uses SIMD instructions [16] and splitting up the tree into subtrees [10] so that updates can be processed in parallel. Alternatively, a more complex key-value tree construction can be used such as a Patricia tree [40].

Action	Time (250KB block)	Time (1MB block)
[G] State fraud proof	289.78 ms	981.88 ms
[V] State fraud proof	1.50 ms	1.50 ms
[G] Availability fraud proof	7.96ms	50.88ms
[V] Availability fraud proof	0.05ms	0.19ms
[G] Single sample response	< 0.00001ms	< 0.00001ms
[V] Single sample response	< 0.00001ms	< 0.00001ms

Table 5: Computation time (mean over ten repeats) for various actions, where [G] means generate and [V] means verify. We assume that a period consists of 10 transactions, the average transaction size is 225 bytes, and each transaction writes to one key in the state tree.

As expected, verifying an availability fraud proof is significantly quicker than generating one. This is because generation requires checking the entire data matrix, whereas verification only requires checking one row or column. Note that we used a library that uses standard Reed-Solomon algorithms that take $O(k^2)$ time to encode/decode—the benchmarks can be improved by using FFT-based algorithms that take $O(k \log(k))$ time.

7 Discussion

7.1 Succinct Proofs of Computation

There have been advances in succinct proofs of computation, including zk-SNARKs [5] and more recently zk-STARKs [4], which allow a prover to prove that $f(x, W) = y$ for some provided x and y, where even if the witness W is very large in size and the computation f takes a very long time to compute, the proof itself has only logarithmic or constant size and takes logarithmic or constant time to verify.

For future work, we can require block headers to come with such a proof to show that they are correctly erasure coded, removing the need for fraud proofs. Also note that the only significant advantage of the 2D Reed Solomon scheme over the 1D scheme is smaller fraud proofs, so if succinct proofs are used switching back to 1D may be optimal (constructing a legitimate erasure code takes only $O(n \log(n))$ computation time for n shares if Fast Fourier transforms are used [21, 32]).

7.2 Locally decodable codes

Another strategy for removing the need for fraud proofs from this scheme is to use the local decodability feature of multi-dimensional Reed-Solomon codes [42]. Particularly, we construct a "proof of proximity" that consists of a set of

pseudorandomly selected rows and columns (or axis-parallel lines more generally for higher-dimensional codes), using the Merkle root of the data as a source of entropy, which the verifier can verify have degree $< k$, thereby probabilistically verifying that a very high percentage of all axis-parallel lines have degree $< k$ and therefore any non-axis-parallel line has degree $< d * k$. The file would be extended from $k * ... * k$ to $(k * 2d) * ... * (k * 2d)$.

Any block (header) that comes with a valid proof of proximity is admissible, even if some small portion of the extended polynomial data is incorrect. Because of this, a single Merkle branch no longer suffices to prove a single value. Instead, the verifier can select a random non-axis-parallel line that passes through the point, and require the prover to provide at least $\frac{3}{2} * d$ points along the line. The verifier computes the correct value at the desired point, doing error correction if necessary. For added soundness, the verifier can select multiple random non-axis-parallel lines.

This scheme has the benefit that it does not rely on fraud proofs or expensive proofs of computation, but has the weaknesses that (*i*) it requires more encoded data to be stored across the network, though this is mitigated by the fact that the larger number of shares makes it safer to have a smaller number of copies of each share stored across the network, and (*ii*) Merkle proofs become roughly two orders of magnitude larger.

8 Related Work

The original Bitcoin whitepaper [25] briefly mentions the possibility of 'alerts', which are messages sent by full nodes to alert light clients that a block is invalid, prompting them to download the full block to verify the inconsistency. Little further exploration has been done on this, partly due to the data availability problem.

There have been online discussions about how one may go about designing a fraud proof system [31, 37], however no complete design that deals with all block invalidity cases and data availability has been proposed. These earlier systems have taken the approach of attempting to design a fraud proof for each possible way to create a block that violates the protocol rules (*e.g.,* double spending inputs, mining a block with a reward too high, etc), whereas this paper generalises the blockchain into a state transition system with only one fraud proof.

On the data availability side, Perard *et al.* [28] have proposed using erasure coding to allow light clients to voluntarily contribute to help storing the blockchain without having to download all of it, however they do not propose a scheme to allow light clients to verify that the data is available via random sampling and fraud proofs of incorrectly generated erasure codes.

9 Conclusion

We presented, implemented and evaluated a complete fraud and data availability proof scheme, which enables light clients to have security guarantees almost at the level of a full node, with the added assumptions that there is at least one honest full node in the network that distributes fraud proofs within a maximum network delay, and that there is a minimum number of light clients in the network to collectively recover blocks.

Acknowledgements

Mustafa Al-Bassam is supported by a scholarship from The Alan Turing Institute and Alberto Sonnino is supported by the European Commission Horizon 2020 DECODE project under grant agreement number 732546.

Thanks to George Danezis, Alexander Hicks and Sarah Meiklejohn for helpful discussions about the mathematical proofs.

References

1. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: Chainspace: A sharded smart contracts platform. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2018)
2. Antonopoulos, A.M.: Mastering Bitcoin: Unlocking Digital Crypto-Currencies. O'Reilly Media, Inc., 1st edn. (2014)
3. Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., Danezis, G.: Consensus in the age of blockchains. CoRR **abs/1711.03936** (2017), https://arxiv.org/abs/1711.03936
4. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptology ePrint Archive **2018**, 46 (2018)
5. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 781–796. USENIX Association, San Diego, CA (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson
6. Buterin, V.: Ethereum: The ultimate smart contract and decentralized application platform (white paper) (2013), http://web.archive.org/web/20131228111141/http://vbuterin.com/ethereum.html
7. Buterin, V.: Ethereum sharding FAQs (2018), https://github.com/ethereum/wiki/wiki/Sharding-FAQs/c54cf1b520b0bd07468bee6950cda9a2c4ab4982
8. Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM **24**(2), 84–90 (Feb 1981). https://doi.org/10.1145/358549.358563, http://doi.acm.org/10.1145/358549.358563
9. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D., Wattenhofer, R.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security. pp. 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

10. Cutter, A.: Trillian—storage layer (2016), https://github.com/google/trillian/blob/7c2425ba66b08cbf78d88ea365bc81a8f7daef65/storage/README.md
11. Dahlberg, R., Pulls, T., Peeters, R.: Efficient sparse merkle trees. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems. pp. 199–215. Springer International Publishing, Cham (2016)
12. Dudáček, L., Veřtát, I.: Multidimensional parity check codes with short block lengths. In: Telecommunications Forum (TELFOR), 2016 24th. pp. 1–4. IEEE (2016)
13. Elias, P.: Error-free coding. Transactions of the IRE Professional Group on Information Theory **4**(4), 29–37 (1954)
14. Euler, L.: Solutio quarundam quaestionum difficiliorum in calculo probabilium. Opuscula Analytic **2**, 331–346 (1785)
15. Ferrante, M., Saltalamacchia, M.: The coupon collector's problem. Materials matemàtics pp. 0001–35 (2014)
16. Guilford, J., Yap, K., Gopal, V.: Fast SHA-256 implementations on Intel® architecture processors (2012), https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf
17. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on Bitcoin's peer-to-peer network. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 129–144. USENIX Association, Washington, D.C. (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman
18. Karlo, T.: Ending Bitcoin support (2018), https://stripe.com/blog/ending-Bitcoin-support
19. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: A secure, scale-out, decentralized ledger via sharding. In: Proceedings of IEEE Symposium on Security & Privacy. IEEE (2018)
20. Laurie, B., Kasper, E.: Revocation transparency (2012), https://www.links.org/files/RevocationTransparency.pdf
21. Lin, S.J., Chung, W.H., Han, Y.S.: Novel polynomial basis and its application to Reed-Solomon erasure codes. In: Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science. pp. 316–325. FOCS '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/FOCS.2014.41, http://dx.doi.org/10.1109/FOCS.2014.41
22. Marshall, A.: Bitcoin scaling problem, explained (2017), https://cointelegraph.com/explained/Bitcoin-scaling-problem-explained
23. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87. pp. 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
24. Miller, A., Bentov, I., Kumaresan, R., McCorry, P.: Sprites: Payment channels that go faster than Lightning. CoRR **abs/1702.05812** (2017), http://arxiv.org/abs/1702.05812
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://bitcoin.org/bitcoin.pdf
26. Nieva, R.: Facebook still cashing in on your data as sales surge (2018), https://www.cnet.com/news/facebook-first-quarter-earnings-2018/
27. Orland, K.: Your Bitcoin is no good here—Steam stops accepting cryptocurrency (2017), https://arstechnica.com/gaming/2017/12/steam-drops-Bitcoin-payment-option-citing-fees-and-volatility/

28. Perard, D., Lacan, J., Bachy, Y., Detchart, J.: Erasure code-based low storage blockchain node. In: IEEE International Conference on Blockchain (2018)
29. Peterson, W.W., Wesley, W., Weldon Jr Peterson, E., Weldon, E., Weldon, E.: Error-correcting codes. MIT press (1972)
30. Poon, J., Dryja, T.: The Bitcoin Lightning network: Scalable off-chain instant payments (2016), https://lightning.network/lightning-network-paper.pdf
31. Ranvier, J.: Improving the ability of SPV clients to detect invalid chains (2017), https://gist.github.com/justusranvier/451616fa4697b5f25f60
32. Reed, I., Scholtz, R., Truong, T.K., Welch, L.: The fast decoding of Reed-Solomon codes using Fermat theoretic transforms and continued fractions. IEEE Transactions on Information Theory **24**(1), 100–106 (January 1978). https://doi.org/10.1109/TIT.1978.1055816
33. Saints, K., Heegard, C.: Algebraic-geometric codes and multidimensional cyclic codes: a unified theory and algorithms for decoding using Grobner bases. IEEE Transactions on Information Theory **41**(6), 1733–1751 (1995)
34. Schwartz, D.: Hash tree - Ripple wiki (2013), https://wiki.ripple.com/index.php?title=Hash_Tree&oldid=3120
35. Shea, J.M., Wong, T.F.: Multidimensional codes. Encyclopedia of Telecommunications (2003)
36. Shen, B.Z., Tzeng, K.: Multidimensional extension of Reed-Solomon codes. In: Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on. p. 54. IEEE (1998)
37. Todd, P.: Fraud proofs (2016), https://diyhpl.us/wiki/transcripts/mit-bitcoin-expo-2016/fraud-proofs-petertodd/
38. Wicker, S.B.: Reed-Solomon Codes and Their Applications. IEEE Press, Piscataway, NJ, USA (1994)
39. Wong, J.I.: CryptoKitties is causing Ethereum network congestion (2017), https://qz.com/1145833/cryptokitties-is-causing-ethereum-network-congestion/
40. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger - Byzantium version, e94ebda (yellow paper) (2018), https://ethereum.github.io/yellowpaper/paper.pdf
41. Wu, J., Costello, D.: New multilevel codes over GF(q). IEEE transactions on information theory **38**(3), 933–939 (1992)
42. Yekhanin, S.: Locally decodable codes. NOW Publishers (January 2010), https://www.microsoft.com/en-us/research/publication/locally-decodable-codes/
43. Young, J.: Vitalik Buterin and Joseph Poon on Ethereum scalability issues, Plasma and Cosmos (2018), https://journal.binarydistrict.com/vitalik-buterin-and-joseph-poon-on-ethereum-scalability-issues-plasma-and-cosmos-/

A Double-tree Design

In the paper we have considered that block headers contain a single data root $\mathsf{dataRoot}_i$ that includes both transactions and intermediate state roots, in order to allow for data availability proofs. However, we can design a more simplified structure with two trees, txRoot_i and $\mathsf{txLength}_i$ for transactions and $\mathsf{traceRoot}_i$ and $\mathsf{traceLength}_i$ for intermediate state roots. Each leaf of the txRoot_i is a transaction and each leaf of the $\mathsf{traceRoot}_i$ is an intermediate state root. This does not require arranging data into fixed-size shares, however it does not support data availability proofs.

A.1 Period Criterion

The protocol may define a rule such that an intermediate state root must be added to the trace Merkle tree after a certain criterion has been met, for example after every p transactions. We called this a 'period criterion'. However unlike the period criterion mechanism described in Section 4.3, we assume that the period criterion under the double-tree design is a fixed number of transactions, so that it is possible for a fraud proof verified to know which trace is mapped to a specific transaction without downloading all the transactions (*i.e.,* there is a deterministic mapping between transaction indexes and trace indexes).

Based on the rule, we assume a function $\mathsf{period}(txIndex) = traceIndex$ that returns the index $traceIndex$ of the intermediate trace root in the execution trace that is the pre-state for a transaction at index $txIndex$ in the block's transaction list, or -1 if the pre-state if the previous block's *stateRoot*. We denote $trace_i^j$ as the jth intermediate state root for block i in the tree committed to by $\mathsf{traceRoot}$. If $\mathsf{period}(txIndex) = traceIndex$, the pre-state root is $trace_i^{traceIndex}$ if $traceIndex \geq 0$, or *stateRoot* if $traceIndex = -1$.

A standard implementation of period may be $\mathsf{period}(txIndex) = \lfloor \frac{txIndex}{p} \rfloor - 1$ if there is a trace every p transactions.

A.2 Proof of Invalid State Transition

A miner may incorrectly compute $\mathsf{stateRoot}_i$, for example by placing a series of random bytes as $\mathsf{stateRoot}_i$, or by crafting a malicious $\mathsf{stateRoot}_i$ that modifies the state in an invalid way. We can thus use the execution trace provided by $\mathsf{traceRoot}_i$ to prove that some part of the execution trace was invalid.

We define a function $\mathsf{VerifyTransitionFraudProof}$ and its parameters which verifies fraud proofs received from full nodes. If the fraud proof is valid, then the block that the fraud proof is for is permanently rejected by the client.

$$
\begin{aligned}
&\mathsf{VerifyTransitionFraudProof}(\mathsf{blockHash}_i,\\
&\quad trace_i^x, \{trace_i^x \rightarrow \mathsf{traceRoot}_i\}, x, && \text{(pre-state root)}\\
&\quad trace_i^{x+1}, \{trace_i^{x+1} \rightarrow \mathsf{traceRoot}_i\}, && \text{(post-state root)}\\
&\quad (t_i^y, t_i^{y+1}, ..., t_i^{y+m}), y, && \text{(transactions)}\\
&\quad (\{t_i^y \rightarrow \mathsf{txRoot}_i\}, \{t_i^{y+1} \rightarrow \mathsf{txRoot}_i\}\}, ..., \{t_i^{y+m} \rightarrow \mathsf{txRoot}_i\}\}),\\
&\quad (w_i^y, w_i^{y+1}, ..., w_i^{y+m}), && \text{(witnesses)}\\
&) \in \{true, false\}
\end{aligned}
$$

The pre-state root may be omitted from the fraud proof parameters if it is simply the state root of the previous block, and the post-state root may be omitted if it is the state root of the current block, as the client already knows these roots as they are in the block headers.

$\mathsf{VerifyTransitionFraudProof}$ returns $true$ if all of the following conditions are met, otherwise $false$ is returned:

1. $blockHash_i$ corresponds to a block header h_i that the client has downloaded and stored.
2. $\mathsf{VerifyMerkleProof}(trace_x, \{trace_x \rightarrow \mathsf{traceRoot}_i\}, \mathsf{traceRoot}_i, \mathsf{traceLength}_i, x)$ returns *true* if a pre-state root is specified.
3. $\mathsf{VerifyMerkleProof}(trace_{x+1}, \{trace_{x+1} \rightarrow \mathsf{traceRoot}_i\}, \mathsf{traceRoot}_i, \mathsf{traceLength}_i, x+1)$ returns *true* if a post-state root is specified.
4. For each transaction t_i^{y+a} in the proof, $period(y+a) = x$ is true if a pre-state root is specified, otherwise $period(y+a) = -1$ and $y = 0$ is true.
5. For each transaction t_i^{y+a} in the proof, $\mathsf{VerifyMerkleProof}(t_i^{y+a}, \{t_i^{y+a} \rightarrow \mathsf{txRoot}_i\}, \mathsf{txRoot}_i, \mathsf{txLength}_i, y+a)$ returns *true*.
6. Let the intermediate state roots after applying every transaction in the proof one at a time be $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$. If a pre-state root is specfied, then the base case is $\mathsf{interRoot}_i^y = \mathsf{trace}_x$, otherwise $\mathsf{interRoot}_i^y = \mathsf{stateRoot}_{i-1}$. If a post-state is specified, $\mathsf{trace}_{x+1} = \mathsf{interRoot}_i^{y+m}$ is true, otherwise $\mathsf{stateRoot}_i = \mathsf{interRoot}_i^{y+m}$ and $y + m = txLength$ is true.

B Computation of *index* in Step 4 of VerifyCodecFraudProof

In Step 4 of VerifyCodecFraudProof in Section 5.5, *index* can be computed as follows:

- If $axis = 0$ and $ax_x = 0$, $index = j * \mathsf{matrixWidth}_i + pos_x$.
- If $axis = 1$ and $ax_x = 0$, $index = pos_x * \mathsf{matrixWidth}_i + j$.
- If $axis = 1$ and $ax_x = 1$, $index = \frac{1}{2} dataLength + j * \mathsf{matrixWidth}_i + pos_x$.
- If $axis = 0$ and $ax_x = 1$, $index = \frac{1}{2} dataLength + pos_x * \mathsf{matrixWidth}_i + j$.