Fraud and Data Availability Proofs: Detecting Invalid Blocks in Light Clients

Mustafa Al-Bassam[1], Alberto Sonnino[1], Vitalik Buterin[2], and Ismail Khoffi[3]

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

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. Instead of validating blocks, 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 so that fraud can be detected, we can eliminate the honest-majority assumption for block validity, 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 while maintaining a strong assurance that on-chain data is available and valid. We present, implement, and evaluate a fraud and data availability proof system.

1 Introduction and Motivation

Due to the scalability limitations of existing blockchains, popular services have stopped accepting Bitcoin [26] payments due to transactions fees rising as high as $20 [18, 27], and a popular Ethereum [7] contract caused the pending transactions backlog to increase six-fold [38]. Users pay higher fees as they compete to get their transactions included on the chain, due to space being limited, *e.g.,* by Bitcoin's block size limit [2] or Ethereum's block gas limit [39].

While increasing on-chain capacity limits would yield higher transaction throughput, there are concerns that this creates a trade-off that would decrease decentralisation and security. This is because increasing on-chain capacity would increase the computational resources required for ordinary users to fully download and verify the blockchain, to check that all transactions are correct and valid. Consequently fewer users would afford to run fully validating nodes (full nodes) that independently verify the blockchain, requiring users to instead run light clients that assume that the chain favoured by the blockchain's consensus algorithm only contains valid transactions [24].

Light clients operate well under normal circumstances, but have weaker assurances compared to full nodes when the majority of the consensus (*e.g.,* miners or block producers) is dishonest (also known as a '51% attack'). When running a full node, a 51% attack on the Bitcoin or Ethereum network can only censor, reverse or double spend transactions, *i.e.,* by forking the chain. However if users run light clients, a 51% attack can generate blocks that contain invalid transactions that, for example, steal funds or create new money out of thin air, and light clients would not be able to detect this as they do not verify the chain. This increases the incentive for conducting a 51% attack. On the other hand, full nodes would reject those invalid blocks immediately as they verify the chain.

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 any full node that generates such proofs, so that they too can reject them. This gives light clients a level of security similar to full nodes. 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. This solves the 'data availability problem', which asks: how can light clients efficiently check that all the data for a block has been made available by the block producer?

We also implement and evaluate the security and efficiency of our overall design, and show in Section 5.4 that less than 1% of block data needs to be downloaded in order to check that the entire data of the block is available with 99% probability. Fraud proofs for invalid blocks are in the order of kilobytes; with practical parameters we show in Section 6 that for a 1MB block, fraud proofs are under 27KB.

Our work also plays a key role in efforts to scale blockchains with sharding [1, 8, 20], 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. By running light clients that download block headers for shards, nodes can receive fraud proofs for invalid shard block using the techniques described in this paper.

2 Background

Blockchains. The data structure of a blockchain consists of 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 block, and the root of the Merkle tree of all transactions in the block.

Validity Rule. 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 [26] is used, then the chain with the most accumulated work is favoured. They also have a set of transaction validity rules that dictate which transactions are valid, and thus blocks that contain invalid transactions will never be favoured by the consensus algorithm and should in fact always be rejected.

Full Nodes and Light Clients. Full nodes (also known as fully-validating nodes) are nodes which download block headers as well as the list of transactions, verifying that the transactions are valid according to the transaction validity rules. Light clients only download block headers, and assume that the list of transactions are valid according to the transaction validity rules. Light clients verify blocks against the consensus rules, but not the transaction validity rules, and thus assume that the consensus is honest in that they only included valid transactions (unlike full nodes). Light clients may also receive Merkle proofs from full nodes that a specific transaction or state is included in a block header.
Sparse Merkle Trees. A Sparse Merkle tree [11,21] is a Merkle tree that allows for commitments to key-value maps, where values can be updated, inserted or deleted trivially on average in $O(\log(k))$ time in a tree with k keys. The tree is initialised with n leaves where n is extremely large (*e.g.,* $n = 2^{256}$), but where almost all of the leaves have the same default empty value (*e.g.,* 0). The index of each leaf in the tree is its key. Sub-trees with only empty descendant leaves can be replaced by a placeholder value, and sub-trees with only one non-empty descendant leaf can be replaced by a single node. Therefore despite the extremely large number of leaves, each operation takes $O(\log(k))$ time.
Erasure Codes and Reed-Solomon Codes. Erasure codes are error-correcting codes [13, 30] 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 [37] have various applications and are among the most studied error-correcting codes. They can correct up to any combination of k of $2k$ known erasures, and operate over a finite field of order q (where q is a prime power) such that $k < n \leq q$. RS codes have been generalised to multidimensional codes [12,34] in various ways [33,35,40]. In a p multidimensional code, the message is encoded p times along p orthogonal axis, and can be represented as coding in different dimensions of a multidimensional array.

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. First, 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.

3.1 Blockchain Model

We assume a generalised blockchain architecture, where the blockchain consists of a hash-based chain of block headers $H = (h_0, h_1, ...)$. 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 unauthenticated transactions N_i from the network, a block header h_i is considered to be valid if (i) $\mathsf{root}(N_i) = \mathsf{txRoot}_i$ and (ii) given some validity function

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

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 on the state from the previous block S_{i-1}. 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:

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

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

We introduce the concept of intermediate state, which is the state of the chain after processing only some of the transactions in a given block. Thus given the intermediate state $I_i^j = \mathsf{transition}(I_i^{j-1}, t_i^j)$ after executing the first j transactions $(t_i^0, t_i^1, ..., t_i^j)$ in block i where $j \leq n$, and the base case $I_i^{-1} = S_{i-1}$, then $S_i = I_i^n$. In other words, the final intermediate state of a block is the post-state.

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

Aim. The aim of this paper 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.

3.2 Participants and Threat Model

Our protocol assumes a network that consists of full nodes and light clients.

Full nodes. These nodes download and verify the entire blockchain, generating and distributing fraud proofs if a block is invalid. 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 by producing blocks, which we call block producers.

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 the graph of honest full nodes is well connected, a standard assumption made in previous work [19, 20, 23, 26]. This results in a broadcast network, due to the synchrony assumption we will make below.

Light clients. These nodes have 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. These nodes receive fraud proofs from full nodes in the event that a block is invalid.

As is the status quo in prior work [7, 26], we assume that each light client is connected to at least one honest full node (*i.e.,* is not under an eclipse attack [17]), as this is necessary to achieve a synchronous gossiping network (discussed below). However when a light client is connected to multiple full nodes, they do not know which nodes are honest or dishonest, just that at least one of them is. Consequently, light clients may be connected to dishonest full nodes that send block headers that have consensus (state agreement) but correspond to invalid or unavailable blocks (violating state validity), and thus need fraud and data availability proofs to detect this.

For data availability proofs, we assume a minimum number of honest light clients in the network to allow for a block to be reconstructed, as each light client downloads a small chunk of every block. The specific number depends on the parameters of the system, and is analysed in Section 5.4.

Network assumptions. We assume a synchronous peer-to-peer gossiping network [5], a standard assumption in the consensus protocols of most blockchains [20, 23, 26, 28, 42] due to FLP impossibility [16]. Specifically, 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 $T' \leq T + \delta$. In order to guarantee that light clients do not accept block headers that do not have state validity, they must receive fraud proofs in time, hence a synchrony assumption is required. Block headers may be created by adversarial actors, and thus may be invalid, and we cannot rely on an honest majority of consensus-participating nodes for state validity.

4 Fraud Proofs

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 3.1, a block header h_i at height i contains at least the following elements (not including any extra data required *e.g.,* for consensus):

$\mathsf{prevHash}_i$ The hash of the previous block header.
$\mathsf{dataRoot}_i$ 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 4.1).

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, as well as including transactions in the block data, we also need to include intermediate state roots.

4.1 State Root and Execution Trace Construction

To instantiate a blockchain based on the state-based model described in Section 3.1, we make use of Sparse Merkle trees, and represent the state as a key-value map. In a UTXO-based blockchain *e.g.,* Bitcoin [26], keys would be UTXO identifiers, and values would be booleans representing if the UTXOs are unspent or not. The state keeps track of all data relevant to block processing.

We now define a variation of the function transition defined in Section 3.1, called 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 "state witness", or w for short). These Merkle proofs are effectively a deep sub-tree of the same state tree.

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

A state witness w consists of a set of (k, v) key-value pairs and their associated Sparse Merkle proofs in the state tree, $w = \{(k_0, v_0, \{k_0, v_0 \rightarrow \mathsf{stateRoot}\}), (k_1, v_1, \{k_1, v_1 \rightarrow \mathsf{stateRoot}\}), ...\}$.

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. If w is invalid and does not contain all 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 state witness for transaction t_i^j for $\mathsf{stateRoot}_i$.

Given the intermediate state root $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$ after executing the first j transactions $(t_i^0, t_i^1, ..., t_i^j)$ in block i where $j \leq n$, 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 the first j transactions $t_i^0, t_i^1, ..., t_i^j$ in block i.

4.2 Data Root and Periods

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 $\mathsf{trace}_\mathsf{i}^\mathsf{j}$ 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. Each leaf in the data tree represents a share.

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

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

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 [39]).

We thus define a function $\mathsf{parsePeriod}$ which parses a list of messages, and returns a pre-state intermediate root trace_i^x, a post-state intermediate root trace_i^{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_i^x is expected to return trace_i^{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.

$$\mathsf{parsePeriod}((m_0, m_1, ...)) \in \{(\mathsf{trace}_i^x, \mathsf{trace}_i^{x+1}, (t_i^g, t_i^{g+1}, ..., t_i^{g+h})), \mathsf{err}\}$$

Note that trace_i^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_i^{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 $\mathsf{stateRoot}_i$.

4.3 Proof of Invalid State Transition

A malicious block producer may provide a bad $\mathsf{stateRoot}_i$ in the block header that modifies the state an invalid way, *i.e.,* it does not match the new state root that should be returned according to $\mathsf{rootTransition}$. We can use the execution trace provided in $\mathsf{dataRoot}_i$ to prove that some part of the execution trace resulting in $\mathsf{stateRoot}_i$ was invalid, by pin-pointing the first intermediate state root that is invalid. We define a function $\mathsf{VerifyTransitionFraudProof}$ and its parameters which verifies fraud proofs of invalid state transitions received from full nodes. We denote d_i^j as share number j in block i.

Summary of VerifyTransitionFraudProof. A state transition fraud proof consists of (*i*) the relevant shares in the block that contain the bad state transition, (*ii*) Merkle proofs that those shares are in $\mathsf{dataRoot}_i$, and (*iii*) the state witnesses for the transactions contained in those shares. The function takes as input this fraud proof, then (*i*) verifies the Merkle proofs of the shares, (*ii*) parses the transactions from the shares, and (*iii*) checks if applying the transactions on the intermediate pre-state root results in the intermediate post-state root specified in the shares. If it does not, then the fraud proof is valid, and the block that the fraud proof is for should be permanently rejected by the client.

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

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

1. $\mathsf{blockHash_i}$ 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 $\mathsf{parsePeriod}(\mathsf{parseShares}((d_i^y, d_i^{y+1}, ..., d_i^{y+m}))) \in \{(\mathsf{trace}_i^x, \mathsf{trace}_i^{x+1}, (t_i^g, t_i^{g+1}, ..., t_i^{g+h})), \mathsf{err}\}$, the result must not be err. If trace_i^x is nil, then $y = 0$ is true, and if trace_i^{x+1} is nil, then $y + m = \mathsf{dataLength}_i$ is true.
4. Check that applying $(t_i^g, t_i^{g+1}, ..., t_i^{g+h})$ on trace_i^x results in trace_i^{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.

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 high 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.

A naive data availability proof scheme may simply apply a standard one dimenisonal Reed-Solomon encoding to extend the block data. However a malicious block producer could incorrectly generate the extended data. In that case, proving that the extended data is incorrectly generated would be equivalent to sending the entire block itself, as clients would have to re-encode all data themselves to verify the mismatch with the given extended data. It is therefore necessary to use multi-dimensional encoding, so that proofs of incorrectly generated codes are limited to a specific axis, rather than the entire data—limiting the size of the proof to $\mathcal{O}(\sqrt[d]{t})$ for d dimensions instead of $\mathcal{O}(t)$. For simplicity, we will only consider bi-dimensional Reed-Solomon encodings in this paper, but our scheme can be easily generalised to higher dimensions.

We first describe how $\mathsf{dataRoot}_i$ should be constructed under the scheme in Section 5.1, and how light clients can use this to have assurance that the full data is available in Section 5.2.

5.1 2D Reed-Solomon Encoded Merkle Tree Construction

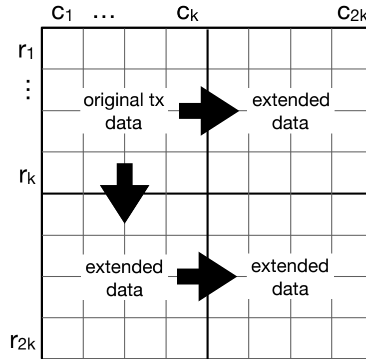

Fig. 1: Diagram showing a 2D Reed-Solomon encoding process.

Let extend be a function that takes in a list of k shares, and returns a list of $2k$ shares that represent the extended shares encoded using a standard one dimensional Reed-Solomon code.

$$\mathsf{extend}(\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_k) = (\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_{2k})$$

The first k shares that are returned are the input shares, and the latter k are the coded shares. Recall that all $2k$ shares can be recovered with knowledge of any k of the $2k$ shares. A 2D Reed-Solomon Encoded Merkle tree can then be constructed as follows from a block of data:

1. Split the original data into shares of size $\mathsf{shareSize}$ each, and arrange them into a $k \times k$ matrix O_i; apply padding if the last share is not exactly of size $\mathsf{shareSize}$, or if there are not enough shares to complete the matrix. In the next step, we extend this $k \times k$ matrix to a $2k \times 2k$ matrix M_i with Reed-Solomon encoding.
2. For each row in the original $k \times k$ matrix O_i, pass the k shares in that row to $\mathsf{extend}(\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_k)$ and append the extra shares outputted $(\mathsf{sh}_{k+1}, ..., \mathsf{sh}_{2k})$ to the row to create an extended row of length $2k$, thus extending the matrix horizontally. Repeat this process for the columns in O_i to extend the matrix vertically, so that each original column now has length $2k$. This creates an extended $2k \times 2k$ matrix with the upper-right and lower-left quadrants filled, as shown in Figure 1. Then finally apply Reed-Solomon encoding horizontally on each row of the vertically extended portion of the matrix to complete the bottom-right quadrant of the $2k \times 2k$ matrix. This results in the 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 $\mathsf{dataRoot}_i = \mathsf{root}((\mathsf{rowRoot}_i^1, \mathsf{rowRoot}_i^2, ..., \mathsf{rowRoot}_i^{2k}, \mathsf{columnRoot}_i^1, \mathsf{columnRoot}_i^2, ..., \mathsf{columnRoot}_i^{2k}))$.

We note that in step 2, we have chosen to extend the vertically extended portion of the matrix horizontally to complete the extended matrix, however it would be equally acceptable to extend the horizontally extended portion of the matrix vertically to complete the extended matrix; this will result in the same matrix because Reed-Solomon coding is linear and commutative with itself [34]. The resulting matrix has the property that all rows and columns have reconstruction capabilities.

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.

In order to allow for Merkle proofs from $\mathsf{dataRoot}_i$ to individual shares, we assume a wrapper function around VerifyMerkleProof called VerifyShareMerkleProof with the same parameters which takes into account how the underlying Merkle trees deal with an unbalanced number of leaves, as $\mathsf{dataRoot}_i$ is composed from multiple trees constructed independently from each other.

The width of the matrix can be derived as $\mathsf{matrixWidth}_i = \sqrt{\frac{1}{2}\mathsf{dataLength}_i}$. 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 5.3, 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.

5.2 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 $(2k)^2$ shares must be unavailable (see Theorem 1), as opposed to $k+1$ out of $2k$ with a 1D code. When light clients receive a new block header from the network, they should randomly sample $0 < s \leq (2k)^2$ distinct shares from the extended matrix, and only accept the block if they receive all shares. The higher the s, the greater the confidence a light client can have that the data is available (this will be analysed in Section 5.4). 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 $0 < x \leq \mathsf{matrixWidth}_i$ and $0 < y \leq \mathsf{matrixWidth}_i$, 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 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 $\mathsf{VerifyMerkleProof}(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{rowRoot}_i^a\}, \mathsf{rowRoot}_i^a, \mathsf{matrixWidth}_i, b)$ is true if the proof is from a row root, otherwise if the proof is from a column root then $\mathsf{VerifyMerkleProof}(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{columnRoot}_i^b\}, \mathsf{columnRoot}_i^b, \mathsf{matrixWidth}_i, a)$ 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 5.3).

Recovery and Selective Share Disclosure There must be a sufficient number of light clients to sample at least $(2k)^2 - (k+1)^2$ different shares in total for the block to be recoverable; recall if $(k+1)^2$ shares are unavailable, the Reed-Solomon matrix may be unrecoverable. Additionally, the block producer can selectively releases shares as light clients ask for them, and always pass the sampling challenge of the clients that ask for the first $(2k)^2 - (k+1)^2$ shares, as they will accept the blocks as available despite them being unrecoverable. The number of light clients will be discussed in Section 5.4.

Table 1 in Section 5.4 will show that the number of light clients that this may apply to is in the hundreds to low thousands if s is set to a reasonable size, which is extremely low (less than $\sim 0.2\%$ of users) compared to for example 1M+ users who have installed a popular Bitcoin Android SPV client [4]. Alternatively, block producers can be prevented from selectively releasing shares to the first clients if one assumes an enhanced network model where 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 [10]. 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. This also prevents adversarial block producers from targeting a specific light client via its known IP address, by only releasing shares to that light client. See Appendix A.2 for proofs.

5.3 Fraud Proofs of Incorrectly Generated Extended Data

If a full node has enough shares to recover any row or column, and after doing so detects that recovered data does not match its respective row or column root, then it should 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.

We define a function $\mathsf{VerifyCodecFraudProof}$ and its parameters that verifies these fraud proofs, where $\mathsf{axisRoot}_i^j \in \{\mathsf{rowRoot}_i^j, \mathsf{columnRoot}_i^j\}$. We denote axis and ax_j as row or column boolean indicators; 0 for rows and 1 for columns.

Summary of VerifyCodecFraudProof. The fraud proof consists of (*i*) the Merkle root of the incorrectly generated row or column, (*ii*) a Merkle proof that the row or column root is in the data tree, (*iii*) enough shares to be able to reconstruct that row or column, and (*iv*) Merkle proofs that each share is in the data tree. The function takes as input this fraud proof, and checks that (*i*) all of the supplied Merkle proofs are valid, (*ii*) all of the shares given by the prover are in the same row or column and (*iii*) that the recovered row or column indeed does not match the row or column root in the block. If all these conditions are true, then the fraud proof is valid, and the block that the fraud proof is for should be permanently rejected by the client.

$$
\begin{aligned}
&\mathsf{VerifyCodecFraudProof}(\mathsf{blockHash}_i, \\
&\quad \mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \to \mathsf{dataRoot}_i\}, j, && \text{(row or column root)} \\
&\quad \mathsf{axis}, && \text{(row or column indicator)} \\
&\quad ((\mathsf{sh}_0, \mathsf{pos}_0, \mathsf{ax}_0), (\mathsf{sh}_1, \mathsf{pos}_1, \mathsf{ax}_1), ..., (\mathsf{sh}_k, \mathsf{pos}_k, \mathsf{ax}_k)), && \text{(shares)} \\
&\quad (\{\mathsf{sh}_0 \to \mathsf{dataRoot}_i\}, \{\mathsf{sh}_1 \to \mathsf{dataRoot}_i\}\}, ..., \{\mathsf{sh}_k \to \mathsf{dataRoot}_i\}\}) \\
&) \in \{\mathsf{true}, \mathsf{false}\}
\end{aligned}
$$

Let $\mathsf{recover}$ be a function that takes a list of shares and their positions in the row or column $((\mathsf{sh}_0, \mathsf{pos}_0), (\mathsf{sh}_1, \mathsf{pos}_1), ..., (\mathsf{sh}_k, \mathsf{pos}_k))$, and the length of the extended row or column $2k$. The function outputs the full recovered shares $(\mathsf{sh}_0, \mathsf{sh}_1, ..., \mathsf{sh}_{2k})$ or err if the shares are unrecoverable.

$$
\mathsf{recover}(((\mathsf{sh}_0, \mathsf{pos}_0), (\mathsf{sh}_1, \mathsf{pos}_1), ..., (\mathsf{sh}_k, \mathsf{pos}_k)), 2k) \in \{(\mathsf{sh}_0, \mathsf{sh}_1, ..., \mathsf{sh}_{2k}), \mathsf{err}\}
$$

$\mathsf{VerifyCodecFraudProof}$ returns true if all of the following conditions are met:

1. $\mathsf{blockHash_i}$ corresponds to a block header h_i that the client has downloaded and stored.

2. If $\mathsf{axis} = 0$ (row root), $\mathsf{VerifyMerkleProof}(\mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, j)$ returns true.
3. If $\mathsf{axis} = 1$ (col. root), $\mathsf{VerifyMerkleProof}(\mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, \frac{1}{2}\mathsf{dataLength}_i + j)$ returns true.
4. For each $(\mathsf{sh}_x, \mathsf{pos}_x, \mathsf{ax}_x)$, $\mathsf{VerifyShareMerkleProof}(\mathsf{sh}_x, \{\mathsf{sh}_x \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, \mathsf{dataLength}, \mathsf{index})$ 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 $\mathsf{axisRoot}_i^j$. 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. $root(\mathsf{recover}(((\mathsf{sh}_0, \mathsf{pos}_0), (\mathsf{sh}_1, \mathsf{pos}_1), ..., (\mathsf{sh}_k, \mathsf{pos}_k)))) = \mathsf{axisRoot}_i^j$ is false.

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

5.4 Security Probability 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.

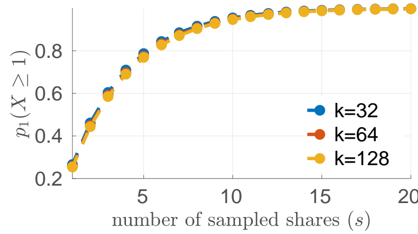

Fig. 2: $p_1(X \geq 1)$ versus the number of samples.

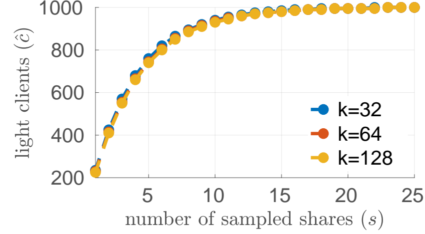

Fig. 3: Light clients $\hat{c}$ for which $p_c(Y > \hat{c}) \geq 0.99$.

Unrecoverable Block Detection Figure 2 shows the probability $p_1(X \geq 1)$ that a single light client samples at least one unavailable share in a matrix with $(k+1)^2$ unavailable shares, thus detecting that a block may be unrecoverable (see Theorem 2 in Appendix A.1). Figure 2 shows how this probability varies with the number of samples s for $k = 32, 64, 128$; each light client samples at least one unavailable share with about 60% probability after 3 samplings (*i.e.,* after querying respectively 0.073% of the block shares for $k = 32$ and 0.005% of the block shares for $k = 128$), and with more than 99% probability after 15 samplings (*i.e.,* after querying respectively 0.4% of the block shares for $k = 32$

$p_e(Z \geq \gamma)$	$s = 2$	$s = 5$	$s = 10$	$s = 20$	$s = 50$
$k = 16$	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.

and 0.02% of the block shares for $k = 128$). Furthermore, this probability is almost independent of k for large values of k (see Corollary 2 in Appendix A.1).
Multi-Client Unrecoverable Block Detection $p_c(Y > \hat{c})$ is the probability that more than $\hat{c}$ out of c light clients sample at least one unavailable share in a matrix with $(k+1)^2$ unavailable shares (see Theorem 3 in Appendix A.1). Figure 3 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, fixing $c = 1000$, and the matrix sizes are $k = 64, 128, 256$. $p_c(Y > \hat{c})$ is almost independent of k, and can be used to determine the number of light clients that will detect incomplete matrices with high probability ($p_c(Y > \hat{c}) \geq 0.99$); there is little gain in increasing s over 15.
Recovery and Selective Share Disclosure Table 1 presents the probability $p_e(Z \geq \gamma) > 0.99$ that light clients collectively samples enough shares to recover every share of the $2k \times 2k$ matrix (see Corollary 3 in Appendix A.1). We are interested in the probability that light clients—each sampling s distinct shares—collectively samples at least γ distinct shares, where γ is the minimum number of distinct shares (randomly chosen) needed to have the certainty to be able to recover the $2k \times 2k$ matrix (see Corollary 1 in Appendix A.1).

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 2,683 lines of Go code and released the code as a series of free and open-source libraries.[4] 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.

Table 2 shows the space complexity and sizes 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; this is because the number of transactions in a period remains static, but the size of the Merkle proof for each transaction increases logarithmically. On the other hand, the availability fraud proofs (as well as block headers with the axis roots) grow at least in proportion to the square root of the size of the block, as the size of a single row or column is proportional to the square root of the size of the block.

Table 3 shows the time complexity and benchmarks for various actions. To generate and verify availability fraud proofs, we use an algorithm based on Fast

[4] URLs omitted for double-blind review.

Object	Space complexity	250KB block	1MB block
State fraud proof	$O(p + p\log(d) + w\log(s) + w)$	14,090b	14,410b
Availability fraud proof	$O(d^{0.5} + d^{0.5}\log(d^{0.5}))$	12,320b	26,688b
Single sample response	$O(\mathsf{shareSize} + \log(d))$	320b	368b
Header	$O(1)$	128b	128b
Header + axis roots	$O(d^{0.5})$	2,176b	4,224b

Table 2: Worst case space complexity and illustrative sizes for various objects for 250KB and 1MB blocks. 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. For the illustrative sizes, we assume 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.

Action	Time complexity	250KB block	1MB block
[G] State fraud proof	$O(b + p\log(d) + w\log(s))$	41.22ms	182.80ms
[V] State fraud proof	$O(p + p\log(d) + w)$	0.03ms	0.03ms
[G] Availability fraud proof	$O(d\log(d^{0.5}) + d^{0.5}\log(d^{0.5}))$	4.91ms	19.18ms
[V] Availability fraud proof	$O(d^{0.5}\log(d^{0.5}))$	0.05ms	0.08ms
[G] Single sample response	$O(log(d^{0.5}))$	< 0.00001ms	< 0.00001ms
[V] Single sample response	$O(log(d^{0.5}))$	< 0.00001ms	< 0.00001ms

Table 3: Worst case time complexity and benchmarks for various actions for 250KB and 1MB blocks (mean over 10 repeats), 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 the benchmarks, 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.

Fourier Transforms (FFT) to perform the encoding and decoding, which has a $O(k\log(k))$ complexity for a message of k shares [22, 32]. 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.

7 Related Work

The Bitcoin paper [26] 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, 36], but 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.* [29] 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.

Error coding as a potential solution has been briefly discussed on IRC chatrooms with no analysis, however these early ideas [25] require semi-trusted third parties to inform clients of missing samples, and do not make use of 2D coding and proofs of incorrectly generated codes and are thus vulnerable to block producers that generate invalid codes.

Cachin and Stefano [9] introduce verifiable information dispersal, which stores files by distributing them amongst a set of servers in a storage-efficient way, where up to one third of the servers and an arbitrary number of clients may be malicious. Data availability proofs on the other hand do not make any honest majority assumptions about nodes, however require a minimum number of honest light clients.

7.1 SParse frAud pRotection (SPAR)

Since the release of this paper's pre-print, new work by Yu *et al.* [41] on data availability proofs was presented in FC'20 that builds on this work, which adopts our security definitions and framework. An alternative data availability proof scheme called SPAR is proposed where only an $O(1)$ hash commitment is required in each header with respect to the size of the block, compared to an $O(\sqrt{n})$ commitment in our scheme. The scheme uses a Merkle tree where each layer of the tree is coded with an LDPC code [6]. The scheme considers sampling with two types of adversarial block producers: a strong adversary and a weak adversary. A strong adversary can find, with NP-hardness, the specific shares that must be hidden (the stopping set) in order to make the data unavailable. A weak adversary cannot find the stopping set, and thus randomly selects shares to withhold. Under a threat model that assumes a strong adversary, clients must therefore sample more shares to achieve the same data availability guarantees.

According to the evaluation of the scheme in the paper [41], light clients are required to download 2.5-4x more samples than our 2D-RS scheme from each block to achieve the same level of data availability guarantee under a weak adversary, and 10-16x more under a strong adversary. Furthermore, the size of each sample is increased as shares must be downloaded from multiple layers of the tree, as opposed to only the bottom layer in our 2D-RS scheme. However, the overall amount of data that needs to be downloaded only increases logarithmically with the block size as the size of the Merkle proofs only increase logarithmically, while the header size is $O(1)$ instead of $O(\sqrt{n})$.

Figure 4 in Appendix C compares the overall header and sampling bandwidth cost for different block sizes for both the 2D-RS scheme and the SPAR scheme, with a target data availability guarantee of 99% and 256 byte shares (used in

the evaluation in both this paper and SPAR). We observe that due to the high sampling cost of SPAR, it outperforms 2D-RS in bandwidth costs only when the block size is greater than 50MB under the weak adversary model. After 50MB, the fact that the bandwidth cost only increases logarithmically in SPAR becomes advantageous.

On the other hand, the size of each SPAR sample is smaller when the size of the shares are smaller. To compare the best case scenario, Figure 5 in Appendix C shows the comparison between SPAR and 2D-RS assuming 32 byte shares (*i.e.,* the size of shares are equivalent to the size of the SHA-256 hash). This shows that SPAR outperforms 2D-RS for blocks greater than 6MB under the weak adversary model. However, decreasing the share size increases the fraud proof size and decoding complexity—we refer readers to the SPAR paper [41] for metrics.

8 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.

Thanks to our shephard Sreeram Kannan for providing helpful feedback.

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. Bitcoin Wallet Developers: Bitcoin wallet - apps on Google Play (2018), `https://play.google.com/store/apps/details?id=de.schildbach.wallet`
5. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE transactions on information theory **52**(6), 2508–2530 (2006)
6. Burshtein, D., Miller, G.: Asymptotic enumeration methods for analyzing ldpc codes. IEEE Transactions on Information Theory **50**(6), 1115–1131 (2004). https://doi.org/10.1109/TIT.2004.828064

7. 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
8. Buterin, V.: Ethereum sharding FAQs (2018), https://github.com/ethereum/wiki/wiki/Sharding-FAQs/c54cf1b520b0bd07468bee6950cda9a2c4ab4982
9. Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. In: 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05). pp. 191–201. IEEE (2005)
10. 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
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. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
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. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Annual International Cryptology Conference. pp. 357–388. Springer (2017)
20. 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)
21. Laurie, B., Kasper, E.: Revocation transparency (2012), https://www.links.org/files/RevocationTransparency.pdf
22. 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
23. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 17–30. CCS '16, ACM,

New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978389, http://doi.acm.org/10.1145/2976749.2978389
24. Marshall, A.: Bitcoin scaling problem, explained (2017), https://cointelegraph.com/explained/Bitcoin-scaling-problem-explained
25. Maxwell, G.: (2017), https://botbot.me/freenode/bitcoin-wizards/2017-02-01/?msg=80297226&page=2
26. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://bitcoin.org/bitcoin.pdf
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. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 643–673. Springer (2017)
29. Perard, D., Lacan, J., Bachy, Y., Detchart, J.: Erasure code-based low storage blockchain node. In: IEEE International Conference on Blockchain (2018)
30. Peterson, W.W., Wesley, W., Weldon Jr Peterson, E., Weldon, E., Weldon, E.: Error-correcting codes. MIT press (1972)
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. Shea, J.M., Wong, T.F.: Multidimensional codes. Encyclopedia of Telecommunications (2003)
35. 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)
36. Todd, P.: Fraud proofs (2016), https://diyhpl.us/wiki/transcripts/mit-bitcoin-expo-2016/fraud-proofs-petertodd/
37. Wicker, S.B.: Reed-Solomon Codes and Their Applications. IEEE Press, Piscataway, NJ, USA (1994)
38. Wong, J.I.: CryptoKitties is causing Ethereum network congestion (2017), https://qz.com/1145833/cryptokitties-is-causing-ethereum-network-congestion/
39. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger - Byzantium version, e94ebda (yellow paper) (2018), https://ethereum.github.io/yellowpaper/paper.pdf
40. Wu, J., Costello, D.: New multilevel codes over GF(q). IEEE transactions on information theory **38**(3), 933–939 (1992)
41. Yu, M., Sahraei, S., Li, S., Avestimehr, S., Kannan, S., Viswanath, P.: Coded merkle tree: Solving data availability attacks in blockchains. In: Financial Cryptography and Data Security (2020)
42. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 931–948 (2018)

A Security Theorems and Proofs

A.1 Sampling Security Analysis

We present the security theorems backing Section 5.4.

Theorem 1. *Given a $2k \times 2k$ matrix E as show in Figure 1, 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 $(E_{i,c_1}, \dots, E_{i,c_{k+1}}) \in E_{i,*}$ can be recovered from the available shares of their respective columns E_{*,c_1}, $E_{*,c_2} \dots$, $E_{*,c_{k+1}}$. 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 $(E_{*,c_1}, \dots E_{*,c_{k+1}})$.

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 $(E_{r_1,j}, \dots, E_{r_{k+1},j}) \in E_{*,j}$ can be recovered from the available shares of their respective row $E_{r_1,*}, E_{r_2,*}, \dots, E_{r_{k+1},*}$. 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.

Corollary 1. *Light clients need to collect at least $\gamma = k(3k-2)$ distinct shares to have the certainty to be able to recover the $2k \times 2k$ matrix.*

Proof. Corollary 1 follows directly from Theorem 1:

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

Theorem 2. *Given a $2k \times 2k$ matrix E as shown in Figure 1, 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:*

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

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:

$$p_1(X = 0) = \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} \tag{3}$$

The numerator of Equation (3) 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 (3):

$$p_1(X \geq 1) = 1 - p_1(X = 0) \tag{4}$$

$$= 1 - \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} \tag{5}$$

$$= 1 - \prod_{i=0}^{s-1} \left(1 - \frac{q}{4k^2 - i}\right) \tag{6}$$

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

Corollary 2. *The probability $p_1(X \geq 1)$ is independent of k for large values of k (and only depends on s).*

Proof.

$$\lim_{k \to \infty} p_1(X \geq 1) = \lim_{k \to \infty} \left(1 - \prod_{i=0}^{s-1} \left(1 - \frac{(k+1)^2}{4k^2 - i}\right)\right) \tag{7}$$

$$= 1 - (3/4)^s \tag{8}$$

Therefore $p_1(X \geq 1)$ is independent of k.

Theorem 3. *Given a $2k \times 2k$ matrix E as shown in Figure 1, 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:*

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

where $p_1(X \geq 1)$ is given by Equation (2).

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

$$p_{s,\hat{c}}(Y = \hat{c}) = \binom{c}{\hat{c}} \big(p_1(X \geq 1)\big)^{\hat{c}} \big(1 - p_1(X \geq 1)\big)^{c-\hat{c}} \tag{10}$$

where $p_1(X \geq 1)$ is given by Equation (2). Equation (10) 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 (10) easily generalises to the binomial cumulative distribution function expressed in Equation (11)—the probability of observing *at most* $\hat{c}$ successes is the sum of the probabilities of observing j successes for $j = 1, \ldots, \hat{c}$.

$$p_c(Y \leq \hat{c}) = \sum_{j=1}^{\hat{c}} \binom{c}{j} \big(p_1(X \geq 1)\big)^{j} \big(1 - p_1(X \geq 1)\big)^{c-j} \tag{11}$$

Therefore the probability of observing *more than* $\hat{c}$ successes is given by Equation (12) below, which expands as Equation (9).

$$p_c(Y > \hat{c}) = 1 - p_c(Y \leq \hat{c}) \tag{12}$$

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]*:*

$$p_e(Z \geq n - \lambda) = 1 - \sum_{i=1}^{\infty} (-1)^i \binom{\lambda + i - 1}{\lambda} \binom{n}{\lambda + i} \big(W_i\big)^c \tag{13}$$

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

Corollary 3. *Given a $2k \times 2k$ matrix E as shown in Figure 1, 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 3 can be easily proven by substituting $\lambda = n - \gamma$ and $n = (2k)^2$ into Theorem 4, where γ is computed by Corollary 1.

A.2 Properties Analysis

We define below *soundness* and *agreement* and discuss them.

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.*

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

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.*

We assume two network connection models that sample requests can be made under, which we will analyse:

- **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.

Corollary 4. *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 3 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 5.2, 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.

Corollary 5. *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,*

$$p_x(X \geq 1) = \sum_{i=1}^{d} \frac{\binom{s}{i}\binom{s(c-1)}{d-i}}{\binom{c \cdot s}{d}} \tag{14}$$

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 5 starts as the proof of Corollary 4; 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 5.2. The enhanced model removes the notion of 'first' clients described in Corollary 4 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 (14). The numerator of Equation (14) 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 (2), Equation (14) 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 3, 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, $d \geq (k+1)^2$.

Note that 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.

B Computation of *index* in Step 4 of VerifyCodecFraudProof

In Step 4 of VerifyCodecFraudProof in Section 5.3, $index$ 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 $ax_x = 1$, $index = \frac{1}{2}dataLength + pos_x * \mathsf{matrixWidth}_i + j$.

C Comparison with SPAR

The graphs below compare the overall header and sampling bandwidth costs between SPAR and our 2D-RS data availability scheme. They were generated using the same analysis code[6] used in the SPAR paper [41].

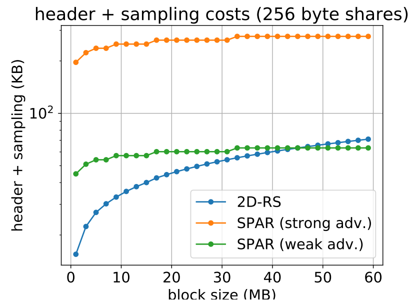

Fig. 4: Comparison of overall header and sampling bandwidth costs between 2D-RS costs, for 256-byte shares with a target data availability guarantee ($p_1(X \geq 1)$) of 99%.

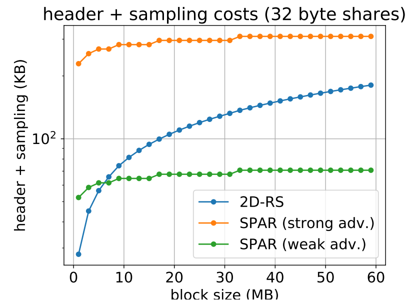

Fig. 5: Comparison of overall header and sampling bandwidth costs between 2D-RS costs, for 32-byte shares with a target data availability guarantee ($p_1(X \geq 1)$) of 99%.

[6] https://github.com/songzLi/SPAR_fraud_proof