Blobstream proofs queries

Prerequisites

- Access to a Celestiaconsensus full node
- RPC endpoint (or full node). The node doesn't need to be a validating node in order for the proofs to be queried. A full node is enough.

Overview of the proof queries

To prove the inclusion of PayForBlobs (PFB) transactions, blobs or shares, committed to in a Celestia block, we use the Celestia consensus node's RPC to query for proofs that can be verified in a rollup settlement contract via Blobstream. In fact, when a PFB transaction is included in a block, it gets separated into a PFB transaction (without the blob), and the actual data blob that it carries. These two are split into shares, which are the low level constructs of a Celestia block, and saved to the corresponding Celestia block. Learn more about shares in theshares specs .

The two diagrams below summarize how a single share, which can contain a PFB transaction, or a part of the rollup data that was posted using a PFB, is committed to in Blobstream.

The share is highlighted in green.R0 ,R1 etc, represent the respective row and column roots, the blue and pink gradients are erasure encoded data. More details on the square layout can be foundin the data square layout anddata structures portion of the specs.

The Celestia square

The commitment scheme

So to prove inclusion of a share to a Celestia block, we use Blobstream as a source of truth. Currently, we will be using the Blobstream X implementation of Blobstream, more information on Blobstream X can be found inthe overview . In a nutshell, Blobstream X attests to the data posted to Celestia in the Blobstream X contract via verifying a zk-proof of the headers of a batch of Celestia blocks. Then, it keeps reference of that batch of blocks using the merkleized commitment of their(dataRoot, height) resulting in adata root tuple root . Check the above diagram which shows:

- 0: those are the shares, that when unified, contain the PFB or the rollup data blob.
- 1: the row and column roots are the namespace merkle tree roots over the shares. More information on the NMT in theNMT specs
- . These commit to the rows and columns containing the above shares.
- 2: the data roots: which are the binary merkle tree commitment over the row and column roots. This means that if you can prove that a share is part of a row, using a namespace merkle proof. Then prove that this row is committed to by the data root. Then you can be sure that that share was published to the corresponding block.
- 3: in order to batch multiple blocks into the same commitment, we create a commitment over the(dataRoot, height)
- tuple for a batch of blocks, which results in a data root tuple root. It's this commitment that gets stored in the Blobstream X smart contract.

So, if we're able to prove that a share is part of a row, then that row is committed to by a data root. Then, prove that that data root along with its height is committed to by the data root tuple root, which gets saved to the Blobstream X contract, we can be sure that that share was committed to in the corresponding Celestia block.

In this document, we will provide details on how to query the above proofs, and how to adapt them to be sent to a rollup contract for verification.

Hands-on demonstration

This part will provide the details of proof generation, and the way to make the results of the proofs queries ready to be consumed by the target rollup contract.

NOTE

For the go client snippets, make sure to have the following replaces in yourgo.mod :

go // go.mod github.com / cosmos / cosmos - sdk => github.com / celestiaorg / cosmos - sdk v1. 18.3 - sdk - v0. 46.14 github.com / gogo / protobuf => github.com / regen - network / protobuf v1. 3.3 - alpha.regen. 1 github.com / syndtr / goleveldb => github.com / syndtr / goleveldb v1. 0.1 - 0.20210819022825 - 2ae1ddf74ef7 github.com / tendermint / tendermint => github.com / celestiaorg / celestia - core v1. 32.0 - tm - v0. 34.29

) // go.mod github.com / cosmos / cosmos - sdk => github.com / celestiaorg / cosmos - sdk v1. 18.3 - sdk - v0. 46.14 github.com / gogo / protobuf => github.com / regen - network / protobuf v1. 3.3 - alpha.regen. 1 github.com / syndtr /

goleveldb => github.com / syndtr / goleveldb v1. 0.1 - 0.20210819022825 - 2ae1ddf74ef7 github.com / tendermint / tendermint => github.com / celestiaorg / celestia - core v1. 32.0 - tm - v0. 34.29

) Also, make sure to update the versions to match the latestgithub.com/celestiaorg/cosmos-sdk andgithub.com/celestiaorg/celestia-core versions.

1. Data root inclusion proof

To prove the data root is committed to by the Blobstream X smart contract, we will need to provide a Merkle proof of the data root tuple to a data root tuple root. This can be created using thedata_root_inclusion_proof query.

Thisendpoint allows querying a data root to data root tuple root proof. It takes a blockheight , a starting block, and an end block, then it generates the binary Merkle proof of theDataRootTuple , corresponding to thatheight , to theDataRootTupleRoot which is committed to in the Blobstream X contract.

The endpoint can be queried using the golang client:

go package

main

import (" context " " fmt " " github.com/tendermint/tendermint/rpc/client/http " " os ")

func

main () { ctx := context. Background () trpc, err := http. New ("tcp://localhost:26657" , "/websocket") if err !=

nil { fmt. Println (err) os. Exit (1) } err = trpc. Start () if err !=

nil { fmt. Println (err) os. Exit (1) } dcProof, err := trpc. DataRootInclusionProof (ctx, 15 , 10 , 20) if err !=

nil { fmt. Println (err) os. Exit (1) } fmt. Println (dcProof.Proof. String ()) } package

main

import (" context " " fmt " " github.com/tendermint/tendermint/rpc/client/http " " os ")

func

main () { ctx := context. Background () trpc, err := http. New ("tcp://localhost:26657" , "/websocket") if err !=

nil { fmt. Println (err) os. Exit (1) } err = trpc. Start () if err !=

nil { fmt. Println (err) os. Exit (1) } dcProof, err := trpc. DataRootInclusionProof (ctx, 15 , 10 , 20) if err !=

nil { fmt. Println (err) os. Exit (1) } fmt. Println (dcProof.Proof. String ()) }

Full example of proving that a Celestia block was committed to by Blobstream X contract

go package

main

import (" context " " fmt " " github.com/celestiaorg/celestia-app/pkg/square " " github.com/ethereum/go-ethereum/accounts/abi/bind " ethcmn

" github.com/ethereum/go-ethereum/common " " github.com/ethereum/go-ethereum/ethclient " blobstreamxwrapper

" github.com/succinctlabs/blobstreamx/bindings " " github.com/tendermint/tendermint/crypto/merkle " " github.com/tendermint/tendermint/rpc/client/http " " math/big " " os ")

func

main () { err :=

verify () if err !=

nil { fmt. Println (err) os. Exit (1) } }

func

verify () error { ctx := context. Background ()

// start the tendermint RPC client trpc, err := http. New ("tcp://localhost:26657" , "/websocket") if err !=

nil { return err } err = trpc. Start () if err !=

nil { return err }

// get the PayForBlob transaction that contains the published blob tx, err := trpc. Tx (ctx, [] byte ("tx_hash"), true) if err !=

nil { return err }

// get the block containing the PayForBlob transaction blockRes, err := trpc. Block (ctx, & tx.Height) if err !=

nil { return err }

// get the nonce corresponding to the block height that contains // the PayForBlob transaction // since BlobstreamX emits events when new batches are submitted, // we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// use the BlobstreamX contract binding wrapper, err := blobstreamxwrapper. NewBlobstreamX (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

LatestBlockNumber, err := ethClient. BlockNumber (context. Background ()) if err !=

nil { return err }

eventsIterator, err := wrapper. FilterDataCommitmentStored (& bind.FilterOpts{ Context: ctx, Start: LatestBlockNumber -

90000 , End: & LatestBlockNumber, }, nil , nil , nil ,) if err !=

nil { return err }

var event * blobstreamxwrapper.BlobstreamXDataCommitmentStored for eventsIterator. Next () { e := eventsIterator.Event if

int64 (e.StartBlock) <= tx.Height && tx.Height <

int64 (e.EndBlock) { event =

& blobstreamxwrapper.BlobstreamXDataCommitmentStored{ ProofNonce: e.ProofNonce, StartBlock: e.StartBlock, EndBlock: e.EndBlock, DataCommitment: e.DataCommitment, } break } } if err := eventsIterator. Error (); err !=

nil { return err } err = eventsIterator. Close () if err !=

nil { return err } if event ==

nil { return fmt. Errorf ("couldn't find range containing the transaction height") }

// get the block data root inclusion proof to the data root tuple root dcProof, err := trpc. DataRootInclusionProof (ctx, uint64 (tx.Height), event.StartBlock, event.EndBlock) if err !=

nil { return err }

// verify that the data root was committed to by the BlobstreamX contract committed, err :=

VerifyDataRootInclusion (ctx, wrapper, event.ProofNonce. Uint64 (), uint64 (tx.Height), blockRes.Block.DataHash, dcProof.Proof) if err !=

nil { return err } if committed { fmt. Println ("data root was committed to by the BlobstreamX contract") } else { fmt. Println ("data root was not committed to by the BlobstreamX contract") return

nil } return

nil }

func

VerifyDataRootInclusion (_ context.Context, blobstreamXwrapper * blobstreamxwrapper.BlobstreamX, nonce uint64 , height uint64 , dataRoot [] byte , proof merkle.Proof,) (bool , error) { tuple := blobstreamxwrapper.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: * (* [32] byte)(dataRoot), } sideNodes :=

make ([][32] byte , len (proof.Aunts)) for i, aunt :=

range proof.Aunts { sideNodes[i] =

* (* [32] byte)(aunt) } wrappedProof := blobstreamxwrapper.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), }

valid, err := blobstreamXwrapper. VerifyAttestation (& bind.CallOpts{}, big. NewInt (int64 (nonce)), tuple, wrappedProof,) if err !=

nil { return

false , err } return valid, nil } package

main

import (" context " " fmt " " github.com/celestiaorg/celestia-app/pkg/square " " github.com/ethereum/go-ethereum/accounts/abi/bind " ethcmn

" github.com/ethereum/go-ethereum/common " " github.com/ethereum/go-ethereum/ethclient " blobstreamxwrapper

" github.com/succinctlabs/blobstreamx/bindings " " github.com/tendermint/tendermint/crypto/merkle " " github.com/tendermint/tendermint/rpc/client/http " " math/big " " os ")

func

main () { err :=

verify () if err !=

nil { fmt. Println (err) os. Exit (1) } }

func

verify () error { ctx := context. Background ()

// start the tendermint RPC client trpc, err := http. New ("tcp://localhost:26657" , "/websocket") if err !=

nil { return err } err = trpc. Start () if err !=

nil { return err }

// get the PayForBlob transaction that contains the published blob tx, err := trpc. Tx (ctx, [] byte ("tx_hash"), true) if err !=

nil { return err }

// get the block containing the PayForBlob transaction blockRes, err := trpc. Block (ctx, & tx.Height) if err !=

nil { return err }

// get the nonce corresponding to the block height that contains // the PayForBlob transaction // since BlobstreamX emits events when new batches are submitted, // we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// use the BlobstreamX contract binding wrapper, err := blobstreamxwrapper. NewBlobstreamX (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

LatestBlockNumber, err := ethClient. BlockNumber (context. Background ()) if err !=

nil { return err }

eventsIterator, err := wrapper. FilterDataCommitmentStored (& bind.FilterOpts{ Context: ctx, Start: LatestBlockNumber -

90000 , End: & LatestBlockNumber, }, nil , nil , nil ,) if err !=

nil { return err }

var event * blobstreamxwrapper.BlobstreamXDataCommitmentStored for eventsIterator. Next () { e := eventsIterator.Event if int64 (e.StartBlock) <= tx.Height && tx.Height <

int64 (e.EndBlock) { event =

& blobstreamxwrapper.BlobstreamXDataCommitmentStored{ ProofNonce: e.ProofNonce, StartBlock: e.StartBlock, EndBlock: e.EndBlock, DataCommitment: e.DataCommitment, } break } } if err := eventsIterator. Error (); err !=

nil { return err } err = eventsIterator. Close () if err !=

nil { return err } if event ==

nil { return fmt. Errorf ("couldn't find range containing the transaction height") }

// get the block data root inclusion proof to the data root tuple root dcProof, err := trpc. DataRootInclusionProof (ctx, uint64 (tx.Height), event.StartBlock, event.EndBlock) if err !=

nil { return err }

// verify that the data root was committed to by the BlobstreamX contract committed, err :=

VerifyDataRootInclusion (ctx, wrapper, event.ProofNonce. Uint64 (), uint64 (tx.Height), blockRes.Block.DataHash, dcProof.Proof) if err !=

nil { return err } if committed { fmt. Println ("data root was committed to by the BlobstreamX contract") } else { fmt. Println ("data root was not committed to by the BlobstreamX contract") return

nil } return

nil }

func

VerifyDataRootInclusion (_ context.Context, blobstreamXwrapper * blobstreamxwrapper.BlobstreamX, nonce uint64 , height uint64 , dataRoot [] byte , proof merkle.Proof,) (bool , error) { tuple := blobstreamxwrapper.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: * (* [32] byte)(dataRoot), }

sideNodes :=

make ([][32] byte , len (proof.Aunts)) for i, aunt :=

range proof.Aunts { sideNodes[i] =

* (* [32] byte)(aunt) } wrappedProof := blobstreamxwrapper.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), }

valid, err := blobstreamXwrapper. VerifyAttestation (& bind.CallOpts{}, big. NewInt (int64 (nonce)), tuple, wrappedProof,) if err !=

nil { return

false , err } return valid, nil }

2. Transaction inclusion proof

To prove that a rollup transaction is part of the data root, we will need to provide two proofs: (1) a namespace Merkle proof of the transaction to a row root. This could be done via proving the shares that contain the transaction to the row root using a namespace Merkle proof. (2) And, a binary Merkle proof of the row root to the data root.

These proofs can be generated using theProveShares query.

Thisendpoint allows querying a shares proof to row roots, then a row roots to data root proofs. It takes a blockheight , a starting share index and an end share index which define a share range. Then, two proofs are generated:

- An NMT proof of the shares to the row roots
- A binary Merkle proof of the row root to the data root

NOTE

If the share range spans multiple rows, then the proof can contain multiple NMT and binary proofs. The endpoint can be queried using the golang client:

go sharesProof, err := trpc. ProveShares (ctx, 15 , 0 , 1) if err !=

nil { ... } sharesProof, err := trpc. ProveShares (ctx, 15 , 0 , 1) if err !=

nil { ... }

Converting the proofs to be usable in theDAVerifier

library

Smart contracts that use theDAVerifier library takes the following proof format:

solidity /// @notice Contains the necessary parameters to prove that some shares, which were posted to /// the Celestia network, were committed to by the BlobstreamX smart contract. struct

SharesProof { // The shares that were committed to. bytes [] data; // The shares proof to the row roots. If the shares span multiple rows, we will have multiple nmt proofs. NamespaceMerkleMultiproof[] shareProofs; // The namespace of the shares. Namespace namespace; // The rows where the shares belong. If the shares span multiple rows, we will have multiple rows. NamespaceNode[] rowRoots; // The proofs of the rowRoots to the data root. BinaryMerkleProof[] rowProofs; // The proof of the data root tuple to the data root tuple root that was posted to the BlobstreamX contract. AttestationProof attestationProof; }

/// @notice Contains the necessary parameters needed to verify that a data root tuple /// was committed to, by the BlobstreamX smart contract, at some specif nonce. struct

AttestationProof { // the attestation nonce that commits to the data root tuple. uint256 tupleRootNonce; // the data root tuple that was committed to. DataRootTuple tuple; // the binary Merkle proof of the tuple to the commitment. BinaryMerkleProof proof; } /// @notice Contains the necessary parameters to prove that some shares, which were posted to /// the Celestia network, were committed to by the BlobstreamX smart contract. struct

SharesProof { // The shares that were committed to. bytes [] data; // The shares proof to the row roots. If the shares span multiple rows, we will have multiple nmt proofs. NamespaceMerkleMultiproof[] shareProofs; // The namespace of the shares. Namespace namespace; // The rows where the shares belong. If the shares span multiple rows, we will have multiple rows. NamespaceNode[] rowRoots; // The proofs of the rowRoots to the data root. BinaryMerkleProof[] rowProofs; // The proof of the data root tuple to the data root tuple root that was posted to the BlobstreamX contract. AttestationProof attestationProof; }

/// @notice Contains the necessary parameters needed to verify that a data root tuple /// was committed to, by the BlobstreamX smart contract, at some specif nonce. struct

AttestationProof { // the attestation nonce that commits to the data root tuple. uint256 tupleRootNonce; // the data root tuple that was committed to. DataRootTuple tuple; // the binary Merkle proof of the tuple to the commitment. BinaryMerkleProof proof; } To construct theSharesProof , we will need the proof that we queried above, and it goes as follows:

data

This is the raw shares that were submitted to Celestia in thebytes format. If we take the example blob that was submitted in theRollupInclusionProofs.t.sol , we can convert it to bytes using theabi.encode(...) as done forthis variable . This can be gotten from the above result of thetransaction inclusion proof query in the fielddata .

shareProofs

This is the shares proof to the row roots. These can contain multiple proofs if the shares containing the blob span across multiple rows. To construct them, we will use the result of thetransaction inclusion proof section.

While theNamespaceMerkleMultiproof being:

solidity /// @notice Namespace Merkle Tree Multiproof structure. Proves multiple leaves. struct

NamespaceMerkleMultiproof { // The beginning key of the leaves to verify. uint256 beginKey; // The ending key of the leaves to verify. uint256 endKey; // List of side nodes to verify and calculate tree. NamespaceNode[] sideNodes; } /// @notice Namespace Merkle Tree Multiproof structure. Proves multiple leaves. struct

NamespaceMerkleMultiproof { // The beginning key of the leaves to verify. uint256 beginKey; // The ending key of the leaves to verify. uint256 endKey; // List of side nodes to verify and calculate tree. NamespaceNode[] sideNodes; } So, we can construct theNamespaceMerkleMultiproof with the following mapping:

- beginKey

- in the Solidity struct==
- start
- in the query response
- endKey
- in the Solidity struct==
- end
- in the query response
- sideNodes
- in the Solidity struct==
- nodes
- in the query response
- TheNamespaceNode
- , which is the type of thesideNodes
- , is defined as follows:

solidity /// @notice Namespace Merkle Tree node. struct

NamespaceNode { // Minimum namespace. Namespace min; // Maximum namespace. Namespace max; // Node value. bytes32 digest; } /// @notice Namespace Merkle Tree node. struct

NamespaceNode { // Minimum namespace. Namespace min; // Maximum namespace. Namespace max; // Node value. bytes32 digest; } So, we construct aNamespaceNode via taking the values from thenodes field in the query response, we convert them from base64 tohex , then we use the following mapping:

- min
- == the first 29 bytes in the decoded value
- max
- == the second 29 bytes in the decoded value
- digest
- == the remaining 32 bytes in the decoded value

Themin andmax areNamespace type which is:

solidity /// @notice A representation of the Celestia-app namespace ID and its version. /// See: https://celestiaorg.github.io/celestia-app/specs/namespace.html struct

Namespace { // The namespace version. bytes1 version; // The namespace ID. bytes28 id; } /// @notice A representation of the Celestia-app namespace ID and its version. /// See: https://celestiaorg.github.io/celestia-app/specs/namespace.html struct

Namespace { // The namespace version. bytes1 version; // The namespace ID. bytes28 id; } So, to construct them, we separate the 29 bytes in the decoded value to:

- first byte:version
- remaining 28 bytes:id

An example of doing this can be found in the[RollupInclusionProofs.t.sol](RollupInclusionProofs.t.sol) test.

A golang helper that can be used to make this conversion is as follows:

go func

toNamespaceMerkleMultiProofs (proofs [] * tmproto.NMTProof) []client.NamespaceMerkleMultiproof { shareProofs :=

make ([]client.NamespaceMerkleMultiproof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([]client.NamespaceNode, len (proof.Nodes)) for j, node :=

range proof.Nodes { sideNodes[j] =

* toNamespaceNode (node) } shareProofs[i] = client.NamespaceMerkleMultiproof{ BeginKey: big. NewInt (int64 (proof.Start)), EndKey: big. NewInt (int64 (proof.End)), SideNodes: sideNodes, } } return shareProofs }

func

minNamespace (innerNode [] byte) * client.Namespace { version := innerNode[0] var id [28] byte for i, b :=

range innerNode[1 : 28] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

maxNamespace (innerNode [] byte) * client.Namespace { version := innerNode[29] var id [28] byte for i, b :=

range innerNode[30 : 57] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

toNamespaceNode (node [] byte) * client.NamespaceNode { minNs :=

minNamespace (node) maxNs :=

maxNamespace (node) var digest [32] byte for i, b :=

range node[58 :] { digest[i] = b } return

& client.NamespaceNode{ Min: * minNs, Max: * maxNs, Digest: digest, } } func

toNamespaceMerkleMultiProofs (proofs [] * tmproto.NMTProof) []client.NamespaceMerkleMultiproof { shareProofs :=

make ([]client.NamespaceMerkleMultiproof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([]client.NamespaceNode, len (proof.Nodes)) for j, node :=

range proof.Nodes { sideNodes[j] =

* toNamespaceNode (node) } shareProofs[i] = client.NamespaceMerkleMultiproof{ BeginKey: big. NewInt (int64 (proof.Start)), EndKey: big. NewInt (int64 (proof.End)), SideNodes: sideNodes, } } return shareProofs }

func

minNamespace (innerNode [] byte) * client.Namespace { version := innerNode[0] var id [28] byte for i, b :=

range innerNode[1 : 28] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

maxNamespace (innerNode [] byte) * client.Namespace { version := innerNode[29] var id [28] byte for i, b :=

range innerNode[30 : 57] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

toNamespaceNode (node [] byte) * client.NamespaceNode { minNs :=

minNamespace (node) maxNs :=

maxNamespace (node) var digest [32] byte for i, b :=

range node[58 :] { digest[i] = b } return

& client.NamespaceNode{ Min: * minNs, Max: * maxNs, Digest: digest, } } withproofs beingsharesProof.ShareProofs .

namespace

Which is the namespace used by the rollup when submitting data to Celestia. As described above, it can be constructed as follows:

solidity /// @notice A representation of the Celestia-app namespace ID and its version. /// See: https://celestiaorg.github.io/celestia-app/specs/namespace.html struct

Namespace { // The namespace version. bytes1 version; // The namespace ID. bytes28 id; } /// @notice A representation of the Celestia-app namespace ID and its version. /// See: https://celestiaorg.github.io/celestia-app/specs/namespace.html struct

Namespace { // The namespace version. bytes1 version; // The namespace ID. bytes28 id; } Via taking thenamespace value from theprove_shares query response, decoding it from base64 to hex, then:

- first byte:version
- remaining 28 bytes:id

An example can be found in theRollupInclusionProofs.t.sol test.

A method to convert to namespace, provided that the namespace size is 29, is as follows:

go func

namespace (namespaceID [] byte) * client.Namespace { version := namespaceID[0] var id [28] byte for i, b :=

range namespaceID[1 :] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } } func

namespace (namespaceID [] byte) * client.Namespace { version := namespaceID[0] var id [28] byte for i, b :=

range namespaceID[1 :] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } } withnamespace beingsharesProof.NamespaceID .

rowRoots

Which are the roots of the rows where the shares containing the Rollup data are localised.

In golang, the proof can be converted as follows:

go func

toRowRoots (roots []bytes.HexBytes) []client.NamespaceNode { rowRoots :=

make ([]client.NamespaceNode, len (roots)) for i, root :=

range roots { rowRoots[i] =

* toNamespaceNode (root. Bytes ()) } return rowRoots } func

toRowRoots (roots []bytes.HexBytes) []client.NamespaceNode { rowRoots :=

make ([]client.NamespaceNode, len (roots)) for i, root :=

range roots { rowRoots[i] =

* toNamespaceNode (root. Bytes ()) } return rowRoots } withroots beingsharesProof.RowProof.RowRoots .

rowProofs

These are the proofs of the rows to the data root. They are of typeBinaryMerkleProof :

solidity /// @notice Merkle Tree Proof structure. struct

BinaryMerkleProof { // List of side nodes to verify and calculate tree. bytes32 [] sideNodes; // The key of the leaf to verify. uint256 key; // The number of leaves in the tree uint256 numLeaves; } /// @notice Merkle Tree Proof structure. struct

BinaryMerkleProof { // List of side nodes to verify and calculate tree. bytes32 [] sideNodes; // The key of the leaf to verify. uint256 key; // The number of leaves in the tree uint256 numLeaves; } To construct them, we take the response of theprove_shares query, and do the following mapping:

- key
- in the Solidity struct==
- index
- in the query response
- numLeaves
- in the Solidity struct==

- total
- in the query response
- sideNodes
- in the Solidity struct==
- aunts
- in the query response

The type of thesideNodes is abytes32 .

An example can be found in theRollupInclusionProofs.t.sol test.

A golang helper to convert the row proofs is as follows:

go func

toRowProofs (proofs [] * merkle.Proof) []client.BinaryMerkleProof { rowProofs :=

make ([]client.BinaryMerkleProof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([][32] byte , len (proof.Aunts)) for j, sideNode :=

range proof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[j] = bzSideNode } rowProofs[i] = client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), } } } func

toRowProofs (proofs [] * merkle.Proof) []client.BinaryMerkleProof { rowProofs :=

make ([]client.BinaryMerkleProof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([][32] byte , len (proof.Aunts)) for j, sideNode :=

range proof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[j] = bzSideNode } rowProofs[i] = client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), } } } withproofs beingsharesProof.RowProof.Proofs .

attestationProof

This is the proof of the data root to the data root tuple root, which is committed to in the Blobstream X contract:

solidity /// @notice Contains the necessary parameters needed to verify that a data root tuple /// was committed to, by the BlobstreamX smart contract, at some specif nonce. struct

AttestationProof { // the attestation nonce that commits to the data root tuple. uint256 tupleRootNonce; // the data root tuple that was committed to. DataRootTuple tuple; // the binary Merkle proof of the tuple to the commitment. BinaryMerkleProof proof; } /// @notice Contains the necessary parameters needed to verify that a data root tuple /// was committed to, by the BlobstreamX smart contract, at some specif nonce. struct

AttestationProof { // the attestation nonce that commits to the data root tuple. uint256 tupleRootNonce; // the data root tuple that was committed to. DataRootTuple tuple; // the binary Merkle proof of the tuple to the commitment. BinaryMerkleProof proof; } * tupleRootNonce * : the nonce at which Blobstream X committed to the batch containing the block containing the data. * tuple * : theDataRootTuple * of the block:

solidity /// @notice A tuple of data root with metadata. Each data root is associated /// with a Celestia block height. /// @dev availableDataRoot in /// https://github.com/celestiaorg/celestia-specs/blob/master/src/specs/data_structures.md#header struct

DataRootTuple { // Celestia block height the data root was included in. // Genesis block is height = 0. // First queryable block is height = 1. uint256 height; // Data root. bytes32 dataRoot; } /// @notice A tuple of data root with metadata. Each data root is associated /// with a Celestia block height. /// @dev availableDataRoot in /// https://github.com/celestiaorg/celestia-specs/blob/master/src/specs/data_structures.md#header struct

DataRootTuple { // Celestia block height the data root was included in. // Genesis block is height = 0. // First queryable block is height = 1. uint256 height; // Data root. bytes32 dataRoot; } which comprises adataRoot , i.e. the block containing the Rollup data data root, and theheight which is theheight of that block.

- proof
- : theBinaryMerkleProof
- of the data root tuple to the data root tuple root. Constructing it is similar to constructing the row roots to data root proof in therowProofs
- section.

An example can be found in theRollupInclusionProofs.t.sol test.

A golang helper to create an attestation proof:

go func

toAttestationProof (nonce uint64 , height uint64 , blockDataRoot [32] byte , dataRootInclusionProof merkle.Proof,) client.AttestationProof { sideNodes :=

make ([][32] byte , len (dataRootInclusionProof.Aunts)) for i, sideNode :=

range dataRootInclusionProof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[i] = bzSideNode }

return client.AttestationProof{ TupleRootNonce: big. NewInt (int64 (nonce)), Tuple: client.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: blockDataRoot, }, Proof: client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (dataRootInclusionProof.Index), NumLeaves: big. NewInt (dataRootInclusionProof.Total), }, } } func

toAttestationProof (nonce uint64 , height uint64 , blockDataRoot [32] byte , dataRootInclusionProof merkle.Proof,) client.AttestationProof { sideNodes :=

make ([][32] byte , len (dataRootInclusionProof.Aunts)) for i, sideNode :=

range dataRootInclusionProof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[i] = bzSideNode }

return client.AttestationProof{ TupleRootNonce: big. NewInt (int64 (nonce)), Tuple: client.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: blockDataRoot, }, Proof: client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (dataRootInclusionProof.Index), NumLeaves: big. NewInt (dataRootInclusionProof.Total), }, } } with thenonce being the attestation nonce, which can be retrieved usingBlobstreamX contract events. Check below for an example. Andheight being the Celestia Block height that contains the rollup data, along with theblockDataRoot being the data root of the block height. Finally,dataRootInclusionProof is the Celestia block data root inclusion proof to the data root tuple root that was queried in the begining of this page.

If thedataRoot or thetupleRootNonce is unknown during the verification:

- dataRoot
- : can be queried using the/block?height=15
- query (15
- in this example endpoint), and taking thedata_hash
- field from the response.
- tupleRootNonce
- : can be retried via querying theBlobstreamXDataCommitmentStored
- events from the BlobstreamX contract and looking for the nonce attesting to the corresponding data. An example:

go // get the nonce corresponding to the block height that contains the PayForBlob transaction // since BlobstreamX emits events when new batches are submitted, we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// use the BlobstreamX contract binding wrapper, err := blobstreamxwrapper. NewBlobstreamX (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

LatestBlockNumber, err := ethClient. BlockNumber (ctx) if err !=

nil { return err }

eventsIterator, err := wrapper. FilterDataCommitmentStored (& bind.FilterOpts{ Context: ctx, Start: LatestBlockNumber -

90000 , // 90000 can be replaced with the range of EVM blocks to look for the events in End: & LatestBlockNumber, }, nil , nil , nil ,) if err !=

nil { return err }

var event * blobstreamxwrapper.BlobstreamXDataCommitmentStored for eventsIterator. Next () { e := eventsIterator.Event if

int64 (e.StartBlock) <= tx.Height && tx.Height <

int64 (e.EndBlock) { event =

& blobstreamxwrapper.BlobstreamXDataCommitmentStored{ ProofNonce: e.ProofNonce, StartBlock: e.StartBlock, EndBlock: e.EndBlock, DataCommitment: e.DataCommitment, } break } } if err := eventsIterator. Error (); err !=

nil { return err } err = eventsIterator. Close () if err !=

nil { return err } if event ==

nil { return fmt. Errorf ("couldn't find range containing the block height") } // get the nonce corresponding to the block height that contains the PayForBlob transaction // since BlobstreamX emits events when new batches are submitted, we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// use the BlobstreamX contract binding wrapper, err := blobstreamxwrapper. NewBlobstreamX (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

LatestBlockNumber, err := ethClient. BlockNumber (ctx) if err !=

nil { return err }

eventsIterator, err := wrapper. FilterDataCommitmentStored (& bind.FilterOpts{ Context: ctx, Start: LatestBlockNumber -

90000 , // 90000 can be replaced with the range of EVM blocks to look for the events in End: & LatestBlockNumber, }, nil , nil , nil ,) if err !=

nil { return err }

var event * blobstreamxwrapper.BlobstreamXDataCommitmentStored for eventsIterator. Next () { e := eventsIterator.Event if

int64 (e.StartBlock) <= tx.Height && tx.Height <

int64 (e.EndBlock) { event =

& blobstreamxwrapper.BlobstreamXDataCommitmentStored{ ProofNonce: e.ProofNonce, StartBlock: e.StartBlock, EndBlock: e.EndBlock, DataCommitment: e.DataCommitment, } break } } if err := eventsIterator. Error (); err !=

nil { return err } err = eventsIterator. Close () if err !=

nil { return err } if event ==

nil { return fmt. Errorf ("couldn't find range containing the block height") }

Listening for new data commitments

For listening for newBlobstreamXDataCommitmentStored events, sequencers can use theWatchDataCommitmentStored as follows:

go ethClient, err := ethclient. Dial ("evm_rpc") if err !=

nil { return err } defer ethClient. Close () blobstreamWrapper, err := blobstreamxwrapper. NewBlobstreamXFilterer (ethcmn. HexToAddress ("contract_address"), ethClient) if err !=

nil { return err }

eventsChan :=

make (chan

* blobstreamxwrapper.BlobstreamXDataCommitmentStored, 100) subscription, err := blobstreamWrapper. WatchDataCommitmentStored (& bind.WatchOpts{ Context: ctx, }, eventsChan, nil , nil , nil ,) if err !=

nil { return err } defer subscription. Unsubscribe ()

for { select { case

<- ctx. Done (): return ctx. Err () case err :=

<- subscription. Err (): return err case event :=

<- eventsChan: // process the event fmt. Println (event) } } ethClient, err := ethclient. Dial ("evm_rpc") if err !=

nil { return err } defer ethClient. Close () blobstreamWrapper, err := blobstreamxwrapper. NewBlobstreamXFilterer (ethcmn. HexToAddress ("contract_address"), ethClient) if err !=

nil { return err }

eventsChan :=

make (chan

* blobstreamxwrapper.BlobstreamXDataCommitmentStored, 100) subscription, err := blobstreamWrapper. WatchDataCommitmentStored (& bind.WatchOpts{ Context: ctx, }, eventsChan, nil , nil , nil ,) if err !=

nil { return err } defer subscription. Unsubscribe ()

for { select { case

<- ctx. Done (): return ctx. Err () case err :=

<- subscription. Err (): return err case event :=

<- eventsChan: // process the event fmt. Println (event) } } Then, new proofs can be created as documented above using the new data commitments contained in the received events.

Example rollup that uses the DAVerifier

An example rollup that uses the DAVerifier can be as simple as:

solidity pragma

solidity

^0.8.22 ;

import { DAVerifier } from

"@blobstream/lib/verifier/DAVerifier.sol" ; import { IDAOracle } from

"@blobstream/IDAOracle.sol" ;

contract SimpleRollup { IDAOracle bridge; ... function

submitFraudProof (SharesProof

memory _sharesProof, bytes32 _root) public { // (1) verify that the data is committed to by BlobstreamX contract (bool committedTo, DAVerifier.ErrorCodes err) = DAVerifier. verifySharesToDataRootTupleRoot (bridge, _sharesProof, _root); if (! committedTo) { revert ("the data was not committed to by Blobstream"); } // (2) verify that the data is part of the rollup block // (3) parse the data // (4) verify invalid state transition // (5) effects } } pragma

solidity

^0.8.22 ;

import { DAVerifier } from

"@blobstream/lib/verifier/DAVerifier.sol" ; import { IDAOracle } from

"@blobstream/IDAOracle.sol" ;

contract SimpleRollup { IDAOracle bridge; ... function

submitFraudProof (SharesProof

memory _sharesProof, bytes32 _root) public { // (1) verify that the data is committed to by BlobstreamX contract (bool committedTo, DAVerifier.ErrorCodes err) = DAVerifier. verifySharesToDataRootTupleRoot (bridge, _sharesProof, _root); if (! committedTo) { revert ("the data was not committed to by Blobstream"); } // (2) verify that the data is part of the rollup

block // (3) parse the data // (4) verify invalid state transition // (5) effects } } Then, you can submit the fraud proof using golang as follows:

go package

main

import (" context " " fmt " " github.com/celestiaorg/celestia-app/pkg/square " " github.com/celestiaorg/celestia-app/x/qgb/client " " github.com/ethereum/go-ethereum/accounts/abi/bind " ethcmn

" github.com/ethereum/go-ethereum/common " " github.com/ethereum/go-ethereum/ethclient " blobstreamxwrapper

" github.com/succinctlabs/blobstreamx/bindings " " github.com/tendermint/tendermint/crypto/merkle " " github.com/tendermint/tendermint/libs/bytes " tmproto

" github.com/tendermint/tendermint/proto/tendermint/types " " github.com/tendermint/tendermint/rpc/client/http " " github.com/tendermint/tendermint/types " " math/big " " os ")

func

main () { err :=

verify () if err !=

nil { fmt. Println (err) os. Exit (1) } }

func

verify () error { ctx := context. Background ()

// ... // check the first section for this part of the implementation

// get the nonce corresponding to the block height that contains the PayForBlob transaction // since Blobstream X emits events when new batches are submitted, we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// ... // check the first section for this part of the implementation

// now we will create the shares proof to be verified by the SimpleRollup // contract that uses the DAVerifier library

// get the proof of the shares containing the blob to the data root sharesProof, err := trpc. ProveShares (ctx, 16 , uint64 (blobShareRange.Start), uint64 (blobShareRange.End)) if err !=

nil { return err }

// use the SimpleRollup contract binding to submit to it a fraud proof simpleRollupWrapper, err := client. NewWrappers (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

// submit the fraud proof containing the share data that had the invalid state transition for example // along with its proof err =

submitFraudProof (ctx, simpleRollupWrapper, sharesProof, event.ProofNonce. Uint64 (), uint64 (tx.Height), dcProof.Proof, blockRes.Block.DataHash,)

return

nil }

func

submitFraudProof (ctx context.Context, simpleRollup * client.Wrappers, sharesProof types.ShareProof, nonce uint64 , height uint64 , dataRootInclusionProof merkle.Proof, dataRoot [] byte ,) error { var blockDataRoot [32] byte for i, b :=

range dataRoot[58 :] { blockDataRoot[i] = b } tx, err := simpleRollup. SubmitFraudProof (& bind.TransactOpts{ Context: ctx, }, client.SharesProof{ Data: sharesProof.Data, ShareProofs: toNamespaceMerkleMultiProofs (sharesProof.ShareProofs), Namespace: * namespace (sharesProof.NamespaceID), RowRoots: toRowRoots (sharesProof.RowProof.RowRoots), RowProofs: toRowProofs (sharesProof.RowProof.Proofs), AttestationProof: toAttestationProof (nonce, height, blockDataRoot, dataRootInclusionProof), }, blockDataRoot,) if err !=

nil { return err } // wait for transaction }

func

toAttestationProof (nonce uint64 , height uint64 , blockDataRoot [32] byte , dataRootInclusionProof merkle.Proof,) client.AttestationProof { sideNodes :=

make ([][32] byte , len (dataRootInclusionProof.Aunts)) for i, sideNode :=

range dataRootInclusionProof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[i] = bzSideNode }

return client.AttestationProof{ TupleRootNonce: big. NewInt (int64 (nonce)), Tuple: client.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: blockDataRoot, }, Proof: client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (dataRootInclusionProof.Index), NumLeaves: big. NewInt (dataRootInclusionProof.Total), }, } }

func

toRowRoots (roots []bytes.HexBytes) []client.NamespaceNode { rowRoots :=

make ([]client.NamespaceNode, len (roots)) for i, root :=

range roots { rowRoots[i] =

* toNamespaceNode (root. Bytes ()) } return rowRoots }

func

toRowProofs (proofs [] * merkle.Proof) []client.BinaryMerkleProof { rowProofs :=

make ([]client.BinaryMerkleProof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([][32] byte , len (proof.Aunts)) for j, sideNode :=

range proof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[j] = bzSideNode } rowProofs[i] = client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), } } }

func

toNamespaceMerkleMultiProofs (proofs [] * tmproto.NMTProof) []client.NamespaceMerkleMultiproof { shareProofs :=

make ([]client.NamespaceMerkleMultiproof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([]client.NamespaceNode, len (proof.Nodes)) for j, node :=

range proof.Nodes { sideNodes[j] =

* toNamespaceNode (node) } shareProofs[i] = client.NamespaceMerkleMultiproof{ BeginKey: big. NewInt (int64 (proof.Start)), EndKey: big. NewInt (int64 (proof.End)), SideNodes: sideNodes, } } return shareProofs }

func

minNamespace (innerNode [] byte) * client.Namespace { version := innerNode[0] var id [28] byte for i, b :=

range innerNode[1 : 28] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

maxNamespace (innerNode [] byte) * client.Namespace { version := innerNode[29] var id [28] byte for i, b :=

range innerNode[30 : 57] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

toNamespaceNode (node [] byte) * client.NamespaceNode { minNs :=

minNamespace (node) maxNs :=

maxNamespace (node) var digest [32] byte for i, b :=

range node[58 :] { digest[i] = b } return

& client.NamespaceNode{ Min: * minNs, Max: * maxNs, Digest: digest, } }

func

namespace (namespaceID [] byte) * client.Namespace { version := namespaceID[0] var id [28] byte for i, b :=

range namespaceID[1 :] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } } package

main

import (" context " " fmt " " github.com/celestiaorg/celestia-app/pkg/square " " github.com/celestiaorg/celestia-app/x/qgb/client " " github.com/ethereum/go-ethereum/accounts/abi/bind " ethcmn

" github.com/ethereum/go-ethereum/common " " github.com/ethereum/go-ethereum/ethclient " blobstreamxwrapper

" github.com/succinctlabs/blobstreamx/bindings " " github.com/tendermint/tendermint/crypto/merkle " " github.com/tendermint/tendermint/libs/bytes " tmproto

" github.com/tendermint/tendermint/proto/tendermint/types " " github.com/tendermint/tendermint/rpc/client/http " " github.com/tendermint/tendermint/types " " math/big " " os ")

func

main () { err :=

verify () if err !=

nil { fmt. Println (err) os. Exit (1) } }

func

verify () error { ctx := context. Background ()

// ... // check the first section for this part of the implementation

// get the nonce corresponding to the block height that contains the PayForBlob transaction // since Blobstream X emits events when new batches are submitted, we will query the events // and look for the range committing to the blob // first, connect to an EVM RPC endpoint ethClient, err := ethclient. Dial ("evm_rpc_endpoint") if err !=

nil { return err } defer ethClient. Close ()

// ... // check the first section for this part of the implementation

// now we will create the shares proof to be verified by the SimpleRollup // contract that uses the DAVerifier library

// get the proof of the shares containing the blob to the data root sharesProof, err := trpc. ProveShares (ctx, 16 , uint64 (blobShareRange.Start), uint64 (blobShareRange.End)) if err !=

nil { return err }

// use the SimpleRollup contract binding to submit to it a fraud proof simpleRollupWrapper, err := client. NewWrappers (ethcmn. HexToAddress ("contract_Address"), ethClient) if err !=

nil { return err }

// submit the fraud proof containing the share data that had the invalid state transition for example // along with its proof err =

submitFraudProof (ctx, simpleRollupWrapper, sharesProof, event.ProofNonce. Uint64 (), uint64 (tx.Height), dcProof.Proof, blockRes.Block.DataHash,)

return

nil }

func

submitFraudProof (ctx context.Context, simpleRollup * client.Wrappers, sharesProof types.ShareProof, nonce uint64 , height uint64 , dataRootInclusionProof merkle.Proof, dataRoot [] byte ,) error { var blockDataRoot [32] byte for i, b :=

range dataRoot[58 :] { blockDataRoot[i] = b } tx, err := simpleRollup. SubmitFraudProof (& bind.TransactOpts{ Context: ctx, }, client.SharesProof{ Data: sharesProof.Data, ShareProofs: toNamespaceMerkleMultiProofs (sharesProof.ShareProofs), Namespace: * namespace (sharesProof.NamespaceID), RowRoots: toRowRoots (sharesProof.RowProof.RowRoots), RowProofs: toRowProofs (sharesProof.RowProof.Proofs), AttestationProof: toAttestationProof (nonce, height, blockDataRoot, dataRootInclusionProof), }, blockDataRoot,) if err !=

nil { return err } // wait for transaction }

func

toAttestationProof (nonce uint64 , height uint64 , blockDataRoot [32] byte , dataRootInclusionProof merkle.Proof,) client.AttestationProof { sideNodes :=

make ([][32] byte , len (dataRootInclusionProof.Aunts)) for i, sideNode :=

range dataRootInclusionProof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[i] = bzSideNode }

return client.AttestationProof{ TupleRootNonce: big. NewInt (int64 (nonce)), Tuple: client.DataRootTuple{ Height: big. NewInt (int64 (height)), DataRoot: blockDataRoot, }, Proof: client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (dataRootInclusionProof.Index), NumLeaves: big. NewInt (dataRootInclusionProof.Total), }, } }

func

toRowRoots (roots []bytes.HexBytes) []client.NamespaceNode { rowRoots :=

make ([]client.NamespaceNode, len (roots)) for i, root :=

range roots { rowRoots[i] =

* toNamespaceNode (root. Bytes ()) } return rowRoots }

func

toRowProofs (proofs [] * merkle.Proof) []client.BinaryMerkleProof { rowProofs :=

make ([]client.BinaryMerkleProof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([][32] byte , len (proof.Aunts)) for j, sideNode :=

range proof.Aunts { var bzSideNode [32] byte for k, b :=

range sideNode { bzSideNode[k] = b } sideNodes[j] = bzSideNode } rowProofs[i] = client.BinaryMerkleProof{ SideNodes: sideNodes, Key: big. NewInt (proof.Index), NumLeaves: big. NewInt (proof.Total), } } }

func

toNamespaceMerkleMultiProofs (proofs [] * tmproto.NMTProof) []client.NamespaceMerkleMultiproof { shareProofs :=

make ([]client.NamespaceMerkleMultiproof, len (proofs)) for i, proof :=

range proofs { sideNodes :=

make ([]client.NamespaceNode, len (proof.Nodes)) for j, node :=

range proof.Nodes { sideNodes[j] =

* toNamespaceNode (node) } shareProofs[i] = client.NamespaceMerkleMultiproof{ BeginKey: big. NewInt (int64 (proof.Start)), EndKey: big. NewInt (int64 (proof.End)), SideNodes: sideNodes, } } return shareProofs }

func

minNamespace (innerNode [] byte) * client.Namespace { version := innerNode[0] var id [28] byte for i, b :=

range innerNode[1 : 28] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

maxNamespace (innerNode [] byte) * client.Namespace { version := innerNode[29] var id [28] byte for i, b :=

range innerNode[30 : 57] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } }

func

toNamespaceNode (node [] byte) * client.NamespaceNode { minNs :=

minNamespace (node) maxNs :=

maxNamespace (node) var digest [32] byte for i, b :=

range node[58 :] { digest[i] = b } return

& client.NamespaceNode{ Min: * minNs, Max: * maxNs, Digest: digest, } }

func

namespace (namespaceID [] byte) * client.Namespace { version := namespaceID[0] var id [28] byte for i, b :=

range namespaceID[1 :] { id[i] = b } return

& client.Namespace{ Version: [1] byte {version}, Id: id, } } For the step (2), check the[rollup inclusion proofs documentation] for more information.

Conclusion

After creating all the proofs, and verifying them:

1. Verify inclusion proof of the transaction to Celestia data root
2. Prove that the data root tuple is committed to by the Blobstream X smart contract

We can be sure that the data was published to Celestia, and then rollups can proceed with their normal fraud proving mechanism.

NOTE

The above proof constructions are implemented in Solidity, and may require different approaches in other programming languages. [][Edit this page on GitHub] Last updated: Previous page Integrate with Blobstream client Next page Requesting data commitment ranges []