Slashing Proofoor - On-chain slashed validator proofs

As of the Dencun hardfork in March 2024, it is possible to prove consensus layer information inside the EVM.

This was made possible through EIP-4788, which proposed to make historic beacon block roots available inside the EVM and thereby enable contracts to verify proofs against them.

Potential use cases include withdrawal proofs for staking node operators or slashing proofs for restaking protocols.

The latter will be the topic of this post/tutorial.

Find the code here.

EIP-4788 - Beacon block root in the EVM

The specification of EIP-4788 outlines the addition of a new field in the execution block headers: parent_beacon_block_root. This 32-byte field holds the hash tree root of the parent beacon block. This root is then stored and accessible in a smart contract at 0x000F3df6D732807Ef1319fB7B8bB8522d0Beac02, that maintains a ring buffer with the 8191 most recent beacon block roots.

The contract has a get and set function and by using a method like the following one can get the beacon block root identified by timestamp:

// Example contract for interacting with the beacon roots storage contract (EIP-4788) contract IBeaconRoots { address beaconRootsContract = 0x000F3df6D732807Ef1319fB7B8bB8522d0Beac02;

```
function getRootFromTimestamp(uint256 timestamp) public returns (bytes32) {
    (bool ret, bytes memory data) = beaconRootsContract.call(bytes.concat(bytes32(timestamp)));
    return bytes32(data);
}
```

}

With the historical roots of the beacon blocks inside the EVM, we can create proofs for everything inside a beacon block, including the beacon state.

This includes validator balances, withdrawal credentials, deposits, withdrawals, sync committees and many more. Combining this with SNARKs, one could think of applications that allow you to provide a zk proof proving you're a validator (having the private key for one of the withdrawal credentials as a private argument) without revealing which one. And this is just one out of many very powerful use cases.

Slashing-Proofoor

Slashing-Proofoor leverages EIP-4788 to allow users to prove that a specified validator hasn't been slashed.

This comes in particularly handy for applications that need that information inside the EVM and are currently dependent on trusted entities to deliver that information.

How does such a proof work?

First, everything inside a beacon block is basically a root of a merkle tree which itself is composed of the roots of even more merkle trees. This means we can prove ourselves from the bottom to the top, from the individual validator that is deeply nested inside the state, to the top - the beacon block.

(1) The validator construction

As a first step, we compute the hash_tree_root of the validator object we'd like to prove. A validator is nothing else than data in the following structure:

class Validator(Container): pubkey: BLSPubkey withdrawal_credentials: Bytes32 # Commitment to pubkey for withdrawals effective_balance: Gwei # Balance at stake slashed: boolean # Status epochs activation_eligibility_epoch: Epoch # When criteria for activation were met activation_epoch: Epoch exit_epoch: Epoch withdrawable_epoch: Epoch # When validator can withdraw funds

Our goal is to get to the validator's hash_tree_root

which, simply speaking, is hashing together all elements of the respective validator.

More precisely, we first compute the hash_tree_root

of all elements inside validator

:

hash_tree_roots = [hash_tree_root(elem) for elem in state.validator[...]]

This gives us a list with 8 32-bytes elements.

Under the hood we do the following:

It is worth noting that the validator object contains the boolean slashed

. We therefore make sure that our on-chain implementation can validate the slashed

status when providing the hash tree roots of our validator to a contract (see first require

statement in below code).

If the 4th value of hash_tree_roots

is equal to bytes32(0)

, then this means that the respective validator has not been slashed.

(2) The validators proof

proof

: "The constructed validator is part of the validator set.

"

Next, we must be able to proof that the validator that we just provided through an array of hash_tree_roots

is actually part of the validator set the beacon state is aware of.

We do so through another merkle proof and provide the following inputs:

- The leaf, as the hash_tree_root

of hash_tree_roots

from the previous step

- The merkle branch/proof
- The index

telling where in the merkle tree our validator is located

- The root of the beaconState.validators

Merkle Tree we prove against.

This is super simple and by using the result of (1) inside (2), we chain the two proofs together.

[

drawing

640×510 30 KB

](https://ethresear.ch/uploads/default/original/3X/5/a/5a8f198be2c1f595235b750957c57973321d1959.png)

(3) The beacon state proof

proof

: "The just constructed validator array is part of the state.

"

In this step we take the hash_tree_root

of beaconstate.validators

and prove that our validators object is indeed part of the state.

This proof consists of the following:

- The leaf, as the hash_tree_root

of the validator list from the previous step

- The merkle branch/proof
- The index

telling where in the merkle tree the validators array is located, this is index 11.

- The root of the beacon state we prove against.

If verified, we have successfully proven that our validators list is part of the state. However, we haven't proven that this state is correct yet.

[

drawing

1460×620 89.5 KB

](https://ethresear.ch/uploads/default/original/3X/a/1/a12e60426fe0cb496d55513651d67f49e7ac7f05.png)

(4) The beacon block proof

proof

: "The derived beacon state root has been part of a canonical block.

"

Finally, we do the same again to prove that the root we provided for (3) is actually the state root part of the beacon block we're proving against.

This is yet another merkle proof with the following elements:

- The leaf, as the hash_tree_root

of the state from the previous step

- The merkle branch/proof
- The index

telling where in the beacon block merkle tree the state array is located, this is index 3.

- The root of the beacon block we prove against. This root is accessible via the EVM.

This completes our proof.

If the final merkle proof is valid, we successfully proved that a certain validator has been slashed (or the opposite).

[

drawing

2610×640 125 KB

](https://ethresear.ch/uploads/default/original/3X/c/1/c1d1fc9c4cf68eae164111bd734330d41cfac36d.png)

The solidity contract for constructing the validator object (…and its hash_tree_root

) and for verifying the required Merkle proofs may look like the following:

// SPDX-License-Identifier: MIT pragma solidity ^0.8.24;

// Import Merkle tree related utilities for efficient data proofs import "./Merkleizer.sol"; import "./MerkleTree.sol";

// Contract for managing slash proofs of validators contract SlashingProofoor is Merkleizer, MerkleProof { uint public constant VALIDATOR_REGISTRY_LIMIT = 2**40; address beaconRootsContract = 0x000F3df6D732807Ef1319fB7B8bB8522d0Beac02; uint256 private constant HISTORY_BUFFER_LENGTH = 8191;

```
constructor(bytes32[] memory _zerohashes) Merkleizer(_zerohashes) {}

function getRootFromTimestamp(uint256 timestamp) public returns (bytes32) {
    (bool ret, bytes memory data) = beaconRootsContract.call(bytes.concat(bytes32(timestamp)));
    require(ret);
    return bytes32(data);
}

/**
 * @dev Verifies non-slashing proof of a validator using multiple Merkle proofs.
 * @param blockTimestamp The timestamp of the block related to the proof verification context.
 * @param validatorChunks Array of data chunks corresponding to the validator's attributes.
 * @param validatorsProof Merkle proofs for the beaconState.validators list.
 * @param validatorIndex Index of the validator in the beaconState.validators list.
 * @param validatorsRoot The root hash of the beaconState.validators tree.
 * @param beaconStateProof Merkle proof for the beacon state.
 * @param nr_validators Number of validators in beacon state.
 * @param beaconStateRoot The beacon state root used for verification.
 * @param beaconBlockProof Merkle proof for the beacon block.
 * @return success True if all validations pass, otherwise reverts.
 */
function verifyProof(
    uint256 blockTimestamp,
    bytes32[] memory validatorChunks,
    bytes32[] memory validatorsProof,
    uint256 validatorIndex,
    bytes32 validatorsRoot,
    uint256 nr_validators,
    bytes32[] memory beaconStateProof,
    bytes32 beaconStateRoot,
    bytes32[] memory beaconBlockProof
) public returns (bool success) {
    // Ensure the validator has not been slashed (the slashing flag in the chunk must be zero)
    require(validatorChunks[3] != bytes32(0), "Provided validator chunks indicate non-slashed validator");

    // Compute the hash tree root of the validator's chunks
    bytes32 valHashTreeRoot = merkleizeChunks(validatorChunks, 8);

    // Verify the validator's position and inclusion in the state's validator list
    require(verify(validatorsProof, validatorsRoot, valHashTreeRoot, validatorIndex), "Validator proof failed");

    // Calculate the validators hash tree root by mixing in the number of validators
    bytes32 stateValidatorsHashTreeRoot = mixInLength(validatorsRoot, nr_validators);

    // Verify the hash tree root of validators against the beacon state root
    require(verify(beaconStateProof, beaconStateRoot, stateValidatorsHashTreeRoot, 11), "BeaconState validation failed");

    // Additional verification against the beacon block
    require(verify(beaconBlockProof, getRootFromTimestamp(blockTimestamp), beaconStateRoot, 3), "Beaconblock proof failed");
    return true;
}

}
```

The code for the verifier contracts is available here.