One of the backwards-compatibility challenges in the current Ethereum design is that history access requires in-EVM verification of Merkle proofs, which assume that the blockchain will forever use the same formatting and the same cryptography. Sharding increases the importance of this, as fraud proofs and validity proofs for rollups will require pointers to shard data.

This post proposes a more future-proof way of doing this: instead of requiring in-EVM verification of proofs to history and shards, we can add precompiles that perform the abstract task of verifying a proof of a certain type. If, in the future, formats are changed, the precompile logic will change automatically. The precompiles can even have conditional logic that verifies one kind of proof for pre-transition slots and another kind of proof for post-transition slots.

Historical block data

def verifyHistoricalBlockRoot(slot: uint256, value: bytes32, proof: bytes)

The precompile will attempt to interpret the proof

in one of two ways:

1. If the proof

is empty, it will check directly if the value

is the saved historical block root in the correct position. If slot

is too old, it will fail.

1. If the proof

is a Merkle branch, it will verify it as a Merkle branch against the correct entry in historical_roots

def verifyHistoricalStateRoot(slot: uint256, value: bytes32, proof: bytes)

Verifies the state root, using the same logic as for the block root.

def verifyHistoricalStateValue(slot: uint256, key: bytes32, value: bytes32, proof: bytes)

Verifies a value in a historical state. The proof

consists of three elements:

- The state root
- A proof that shows the correctness of the state root
- A Patricia or Verkle or other proof that the value

actually is in the position key

in the state tree (this assumes that the[proposed scheme for mapping all account contents to a 32-byte key](#)is permanently enshrined)

def verifyHistoricalTransaction(slot: uint256, txindex: uint256, tx: bytes, proof: bytes)

Verifies that tx

actually is in the txindex

of the block at the given slot

. The proof contains:

- The block root
- A proof that shows the correctness of the block root
- A proof that the given tx

actually is the transaction in the given position

def verifyHistoricalReceipt(slot: uint256, txindex: uint256, receipt: bytes, proof: bytes)

Verifies that receipt

actually is the receipt for the transaction at the txindex

of the block at the given slot

. The proof contains:

- The block root
- A proof that shows the correctness of the block root
- A proof that the given receipt

actually is the receipt in the given position

Shard data

def verifyShardBlockBody(slot: uint256, shard: uint256, startChunk: uint256, chunks: List[bytes32], proof: bytes)

Verifies that chunks = body[startChunk: startChunk + len(chunks)]

where body

is the body at the given shard

in the given slot

. The proof would consist of:

- The Kate proof to prove the subset of a block
- If the slot

is too old (older than 128 epochs?), a Merkle proof to the state root at slot + 96

and then a Merkle proof from that slot to the position in the shard commitments array showing a finalized commitment

While we are using BLS-12-381 Kate commitments, the precompile would also verify that data

is a list of 32-byte chunks where each chunk is less than the curve subgroup order. If no shard block is saved in a given position, the precompile acts as though a commitment to zero-length data was saved in that position. If the value at a given position is uncomfirmed, the precompile always fails.

def verifyShardPolynomialEvaluation(slot: uint256, shard: uint256, x: uint256, y: uint256, proof: bytes)

If we treat the shard block at the given (slot, shard)

as a polynomial P

, with bytes i*32* ... *i*32+31

being the evaluation at w**i

, this verifies that P(x) = y

. The proof

is the same as for the data subset proof, except the Kate proof is proving an evaluation at a point (possibly outside the domain) instead of proving data at a subset of locations.

If we move away from BLS-12-381 in the future (eg. to 32-byte binary field proofs), the precompile would take as input a SNARK verifying that data

is made up entirely of values less than the curve order, and verifying the evaluation of data

over the current field.

This precompile is useful for the cross-polynomial-commitment-scheme proof of equivalence protocol, which can be used to allow ZK rollups to directly operate over shard data.