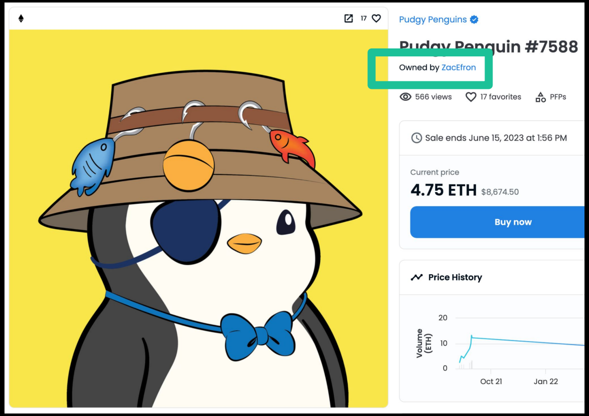
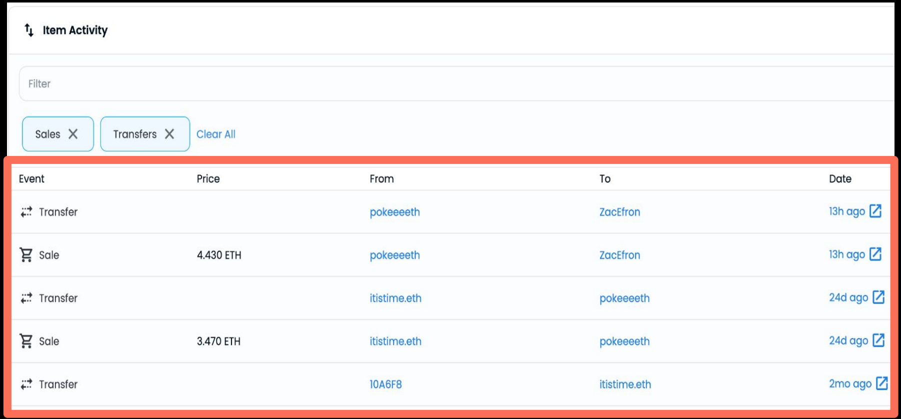
AXIOM

Scaling data-rich applications on Ethereum with Axiom

Smart contracts today are data-starved



Current contract state



Historical transaction and state

To preserve decentralization, smart contracts today **cannot access** history

Developers face painful data tradeoffs

Pay more 🤑 ⟷ **Reduce security** 🤡

Pay more 🤑

- Store more data in state
- Imposes costs on every user
- Limited scale due to gas costs

Reduce security 🤡

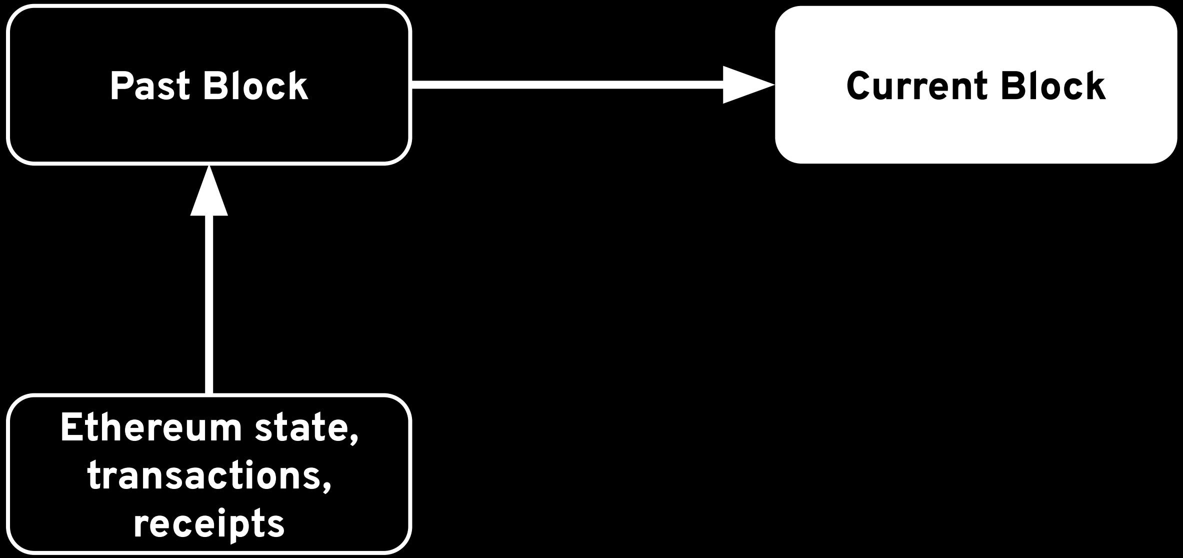
- Use trusted oracle
- Introduces additional trust assumptions on users
- Limited scale due to verification of trust assumptions

Scaling on-chain data access today **increases cost** or **reduces security**

How do we scale data access and compute for smart contracts in an application-specific way?

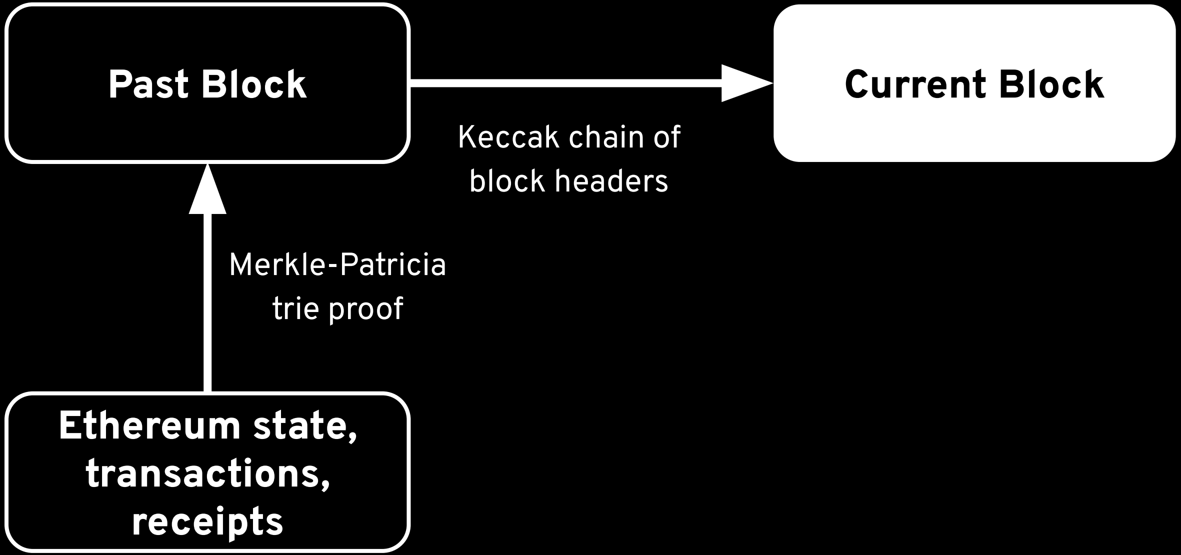
AXIOM

Blockchains offer a new way to access data



We can access on-chain history with **cryptography**, not consensus

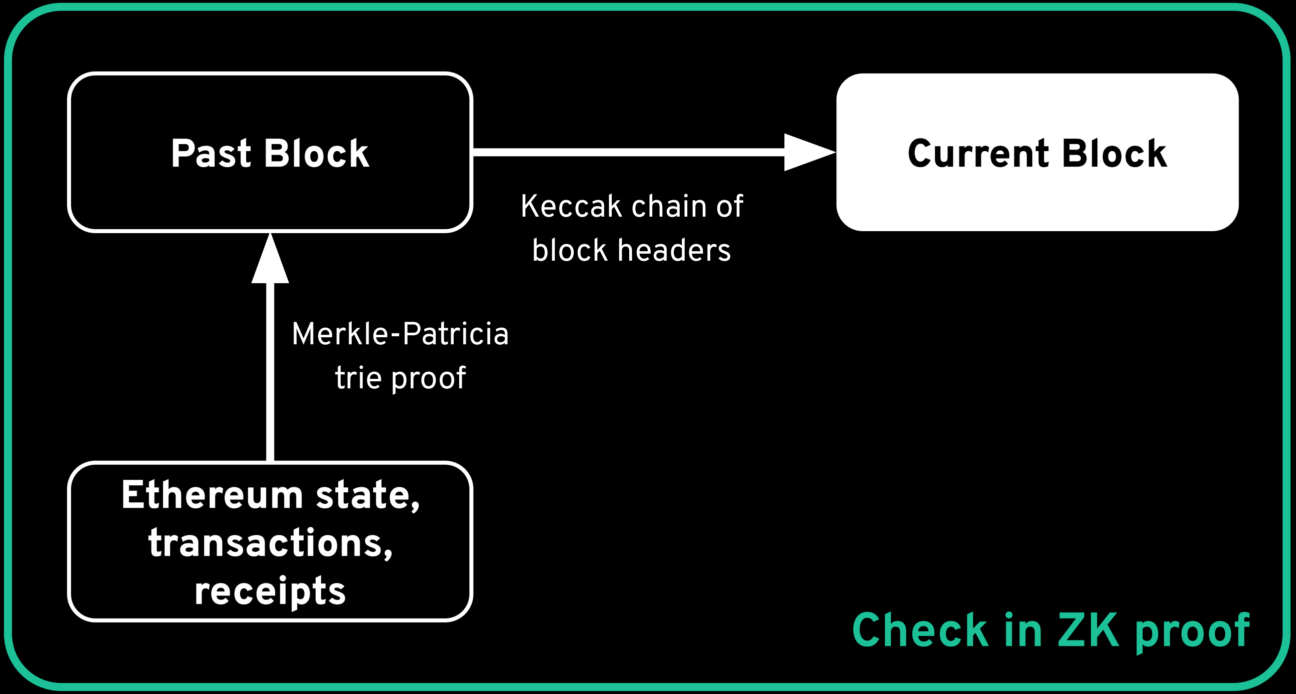
How does cryptographic data access work?



Accessing history natively in the EVM is **prohibitively expensive**

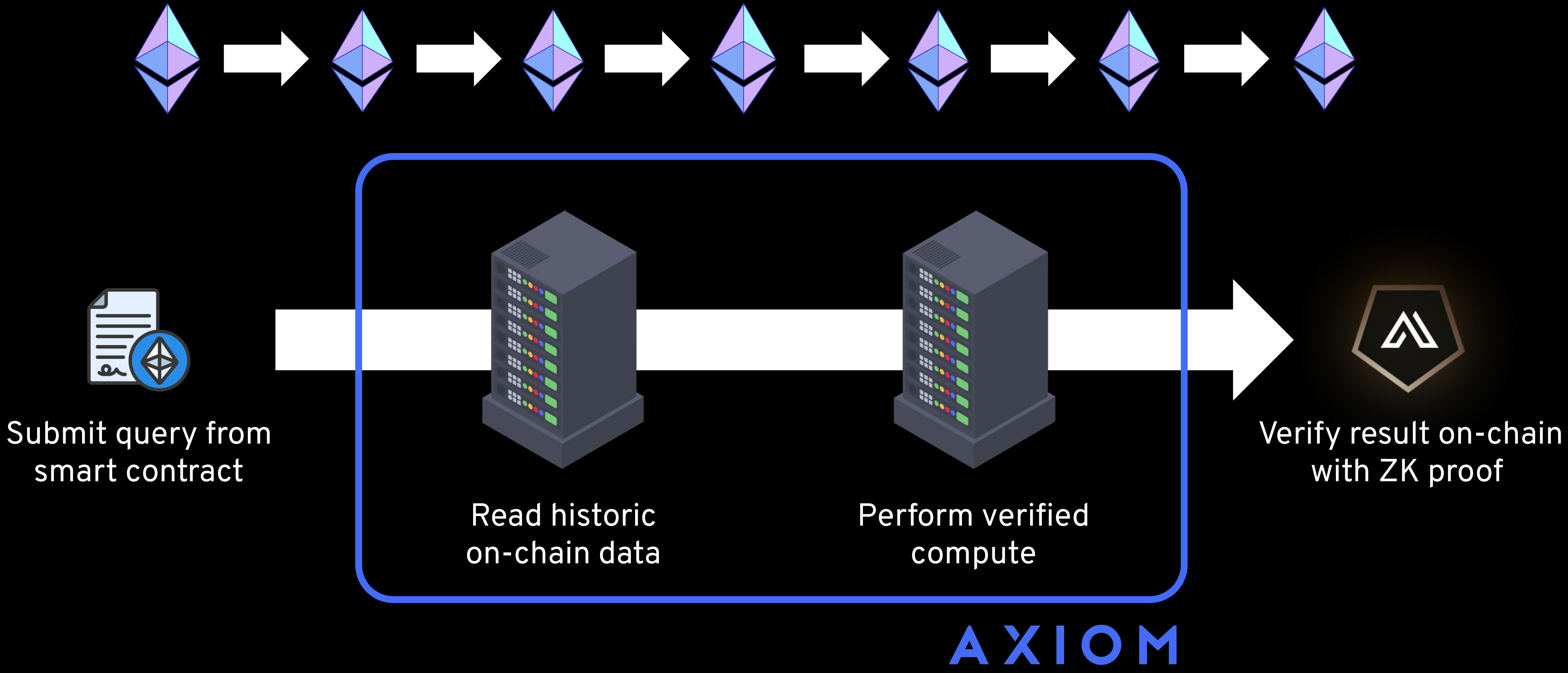
AXIOM

Axiom makes historic data access practical with ZK



Proving data reads in ZK enables **scale** and **composition**

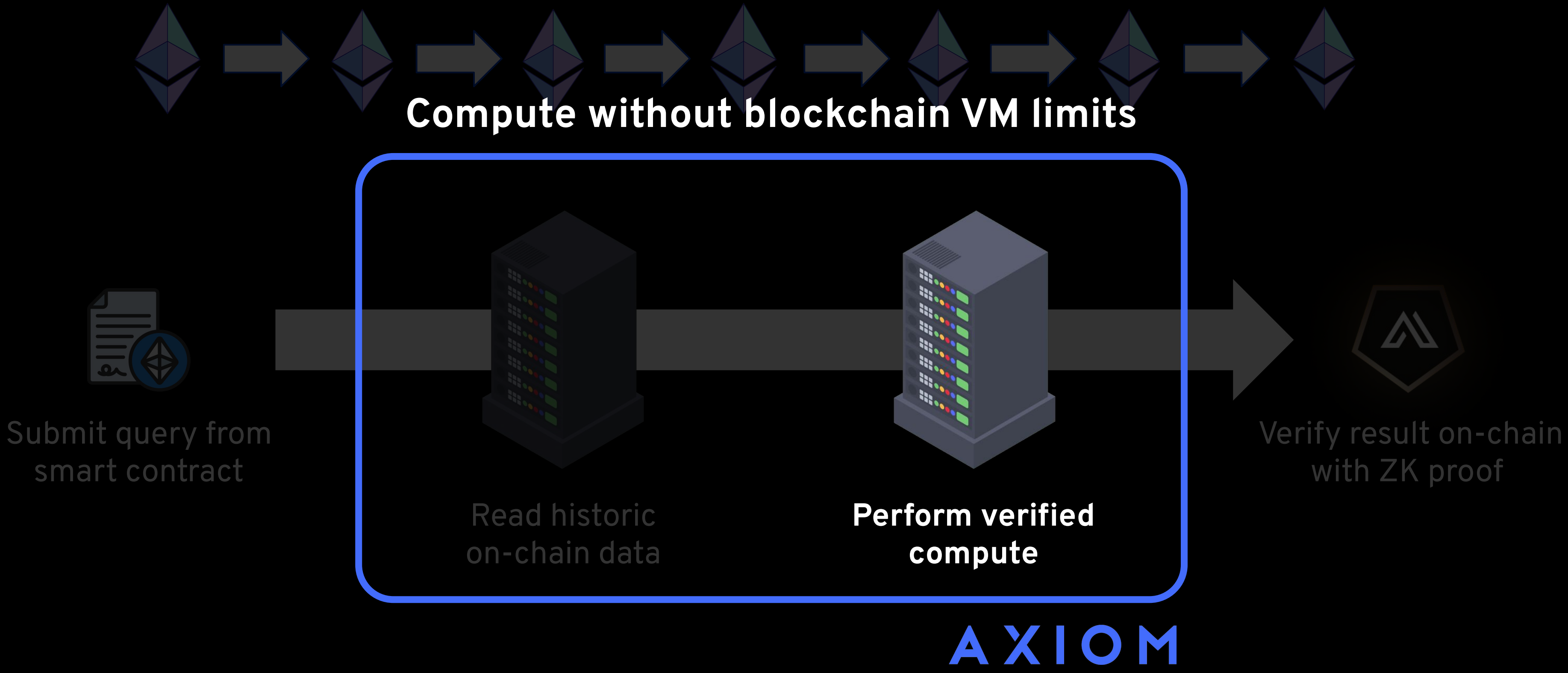
Axiom: The ZK Coprocessor for Ethereum



Every result from Axiom has security
cryptographically equivalent to Ethereum

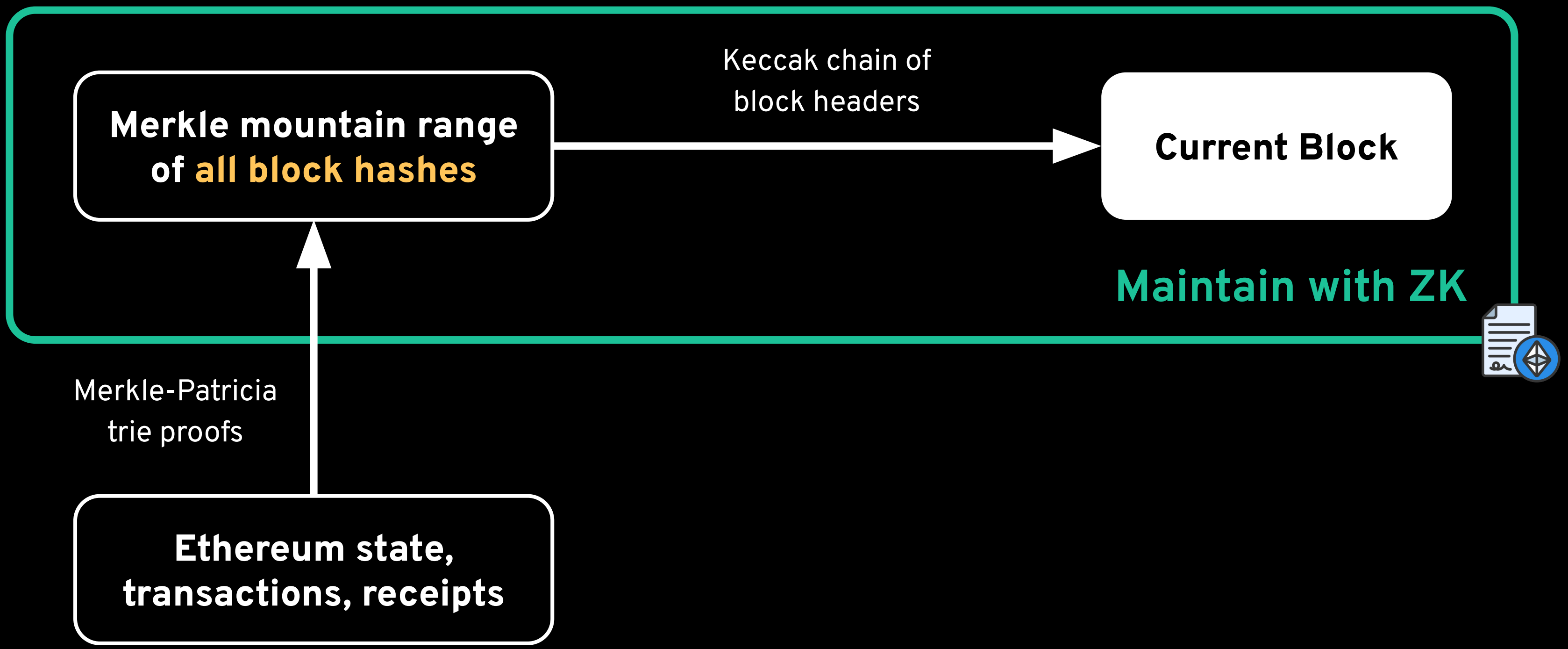
Axiom enables arbitrary on-chain async calls

Axiom enables arbitrary on-chain async calls



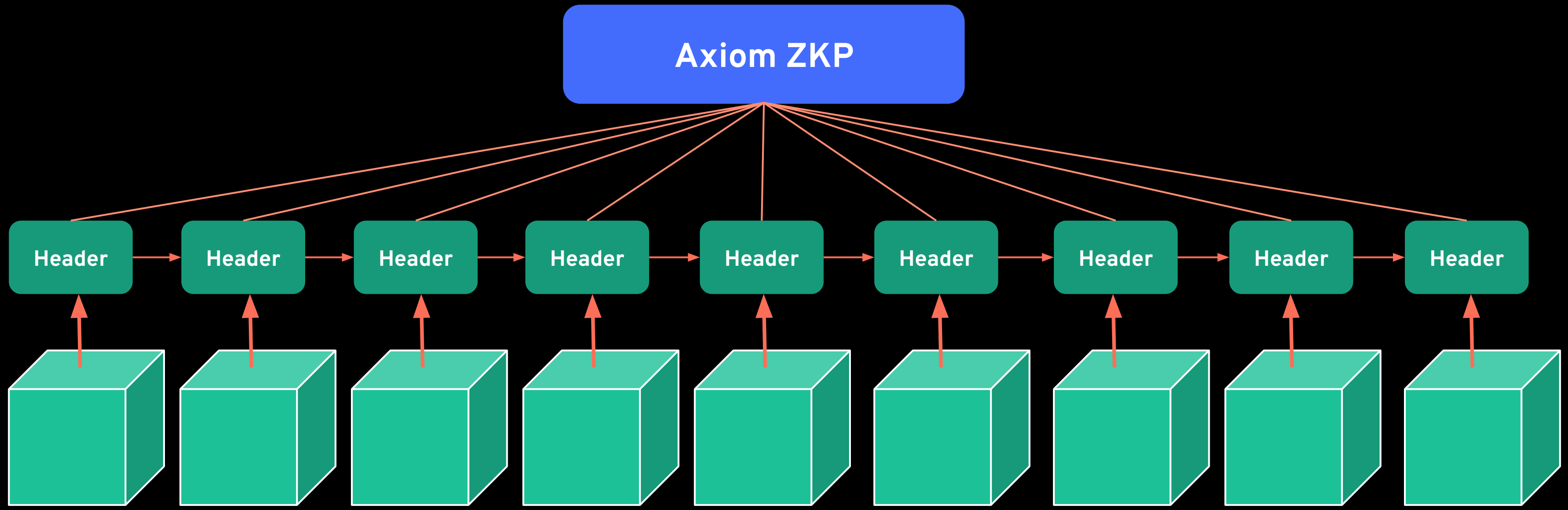
How Axiom Works

Axiom's architecture for reading on-chain history



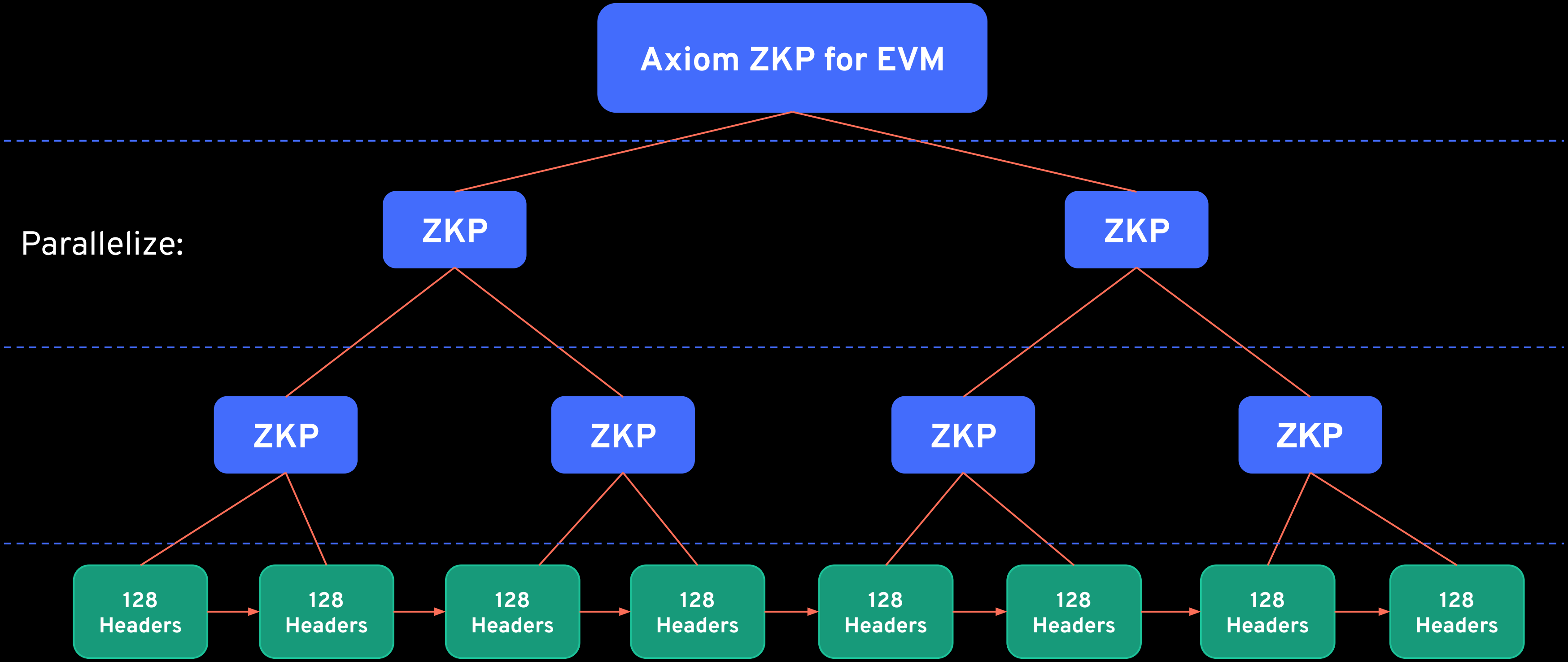
We cache block hashes back to genesis in a Merkle mountain range

Aggregating Historical Block Headers



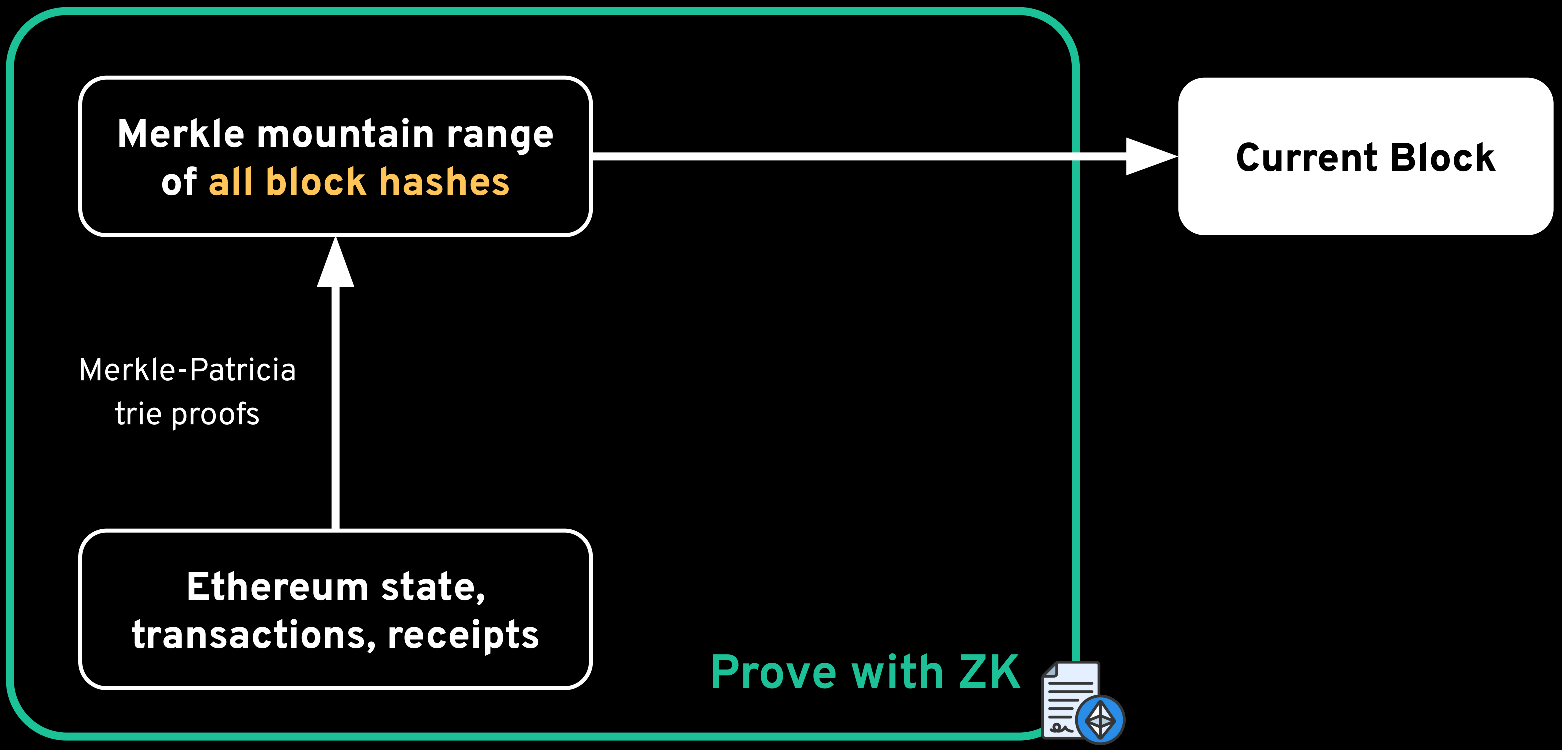
AXIOM

Aggregating Historical Block Headers



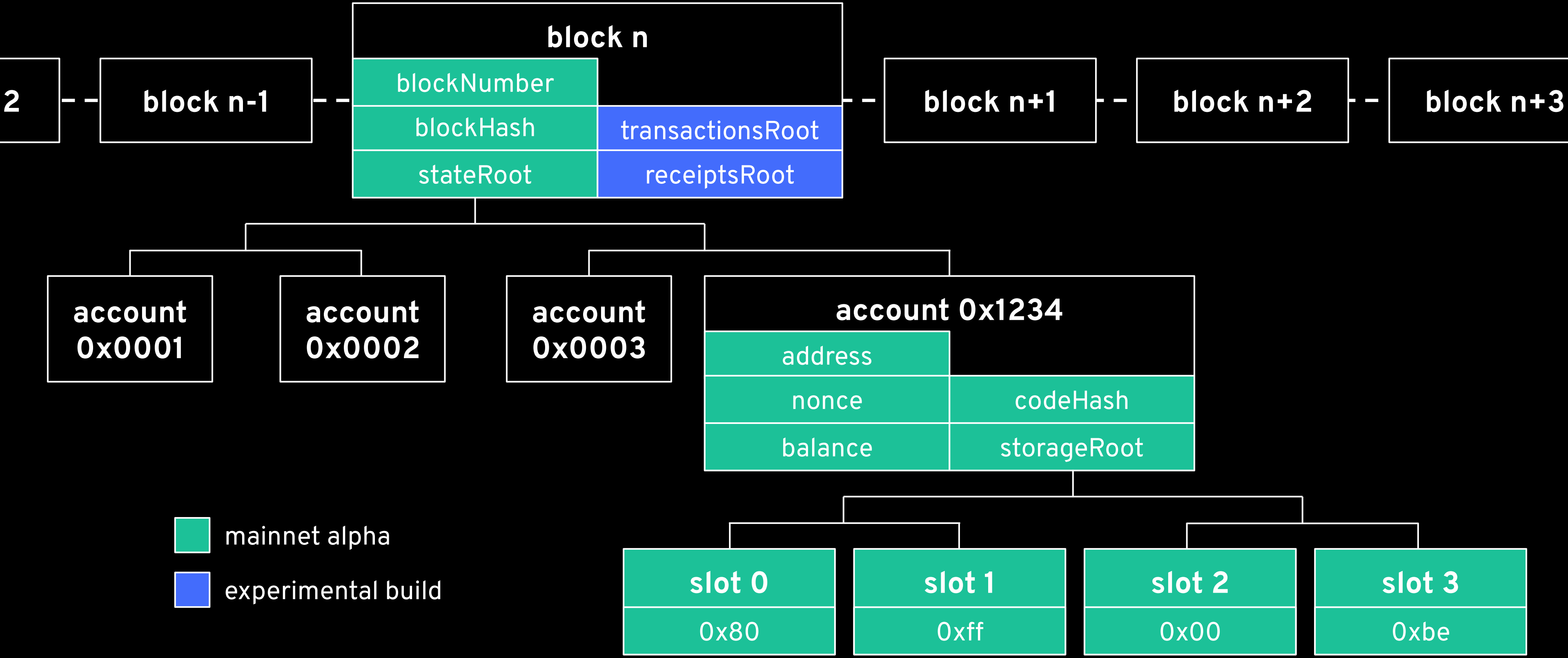
AXIOM

Axiom's architecture for reading on-chain history



Axiom can prove **any combination** of blocks, addresses, and storage slots.

What data does Axiom prove?



Axiom for Developers

Installation

- To prove data that we'll eventually use in a contract, we first use the SDK to build a Query which contains all of the different pieces of data that we want to prove.
- Install **@axiom-crypto/core** and other useful packages:

NPM

```
npm i @axiom-crypto/core ethers
```

YARN

```
yarn add @axiom-crypto/core ethers
```

PNPM

```
pnpm i @axiom-crypto/core ethers
```

What can we prove?

Block Data

- block number
- block hash
- transactions root (experimental)
- receipts root (experimental)

Account Data

- block number
- address
- nonce
- balance
- storage root
- code hash

Storage Data

- block number
- address
- slot number
- slot value

What can we prove? (Experimental)

- Install **@axiom-crypto/experimental** with your favorite package manager

Transaction Data

- nonce
- maxPriorityFeePerGas
- maxFeePerGas
- gasLimit
- to
- value
- data
- v, r, s

Receipt Data

- status
- cumulativeGas
- logsBloom
- logs
  - address
  - topics
  - data

IERC20.sol

```
interface IERC20 {
  event Approval(address indexed owner, address indexed spender, uint value);
  event Transfer(address indexed from, address indexed to, uint value);
  ...
}
```

AXIOM

An example

- I want to create a contract that checks that a user was “active” during the previous bear market and if so, let them mint an NFT:
  - Our custom “bear market”: period of low activity and prices between Aug 10, 2018 (block 6120000) to July 10, 2020 (block 10430000)
- Use an **account proof** to check the difference in an account's nonce at two different block numbers

AXIOM

Setup

- Create a config object and create a new Axiom instance from it:

example.ts

```
import { Axiom, AxiomConfig } from '@axiom-crypto/core';
import { ethers } from 'ethers';

const config: AxiomConfig = {
  providerUri,
  version: "v1",
  chainId: 5,
  mock: true,
};
const ax = new Axiom(config);
```

AXIOM

Overview of steps

Build Query (Typescript SDK)

Submit Query (ethers.js)

Wait for Proof

Parse Proof (Typescript SDK)

Validate Proof (Solidity)

Building a Query

- Create a new QueryBuilder object by calling **newQueryBuilder** on the Axiom instance, then append data to it:

example.ts

```
const queryData = [
  {
    blockNumber: 6120000,
    address: "0xd8da6bf26964af9d7eed9e03e53415d37aa96045", // vitalik.eth
  }, {
    blockNumber: 10430000,
    address: "0xd8da6bf26964af9d7eed9e03e53415d37aa96045", // replace both of these with your address
  }
];

const qb = ax.newQueryBuilder();
await qb.append(queryData[0]);
await qb.append(queryData[1]);
const { keccakQueryResponse, queryHash, query } = await qb.build();
```

Building a Query

- Create a new QueryBuilder object by calling **newQueryBuilder** on the Axiom instance, then append data to it:

example.ts

```
const queryData = [
  {
    blockNumber: 6120000,
    address: "0xd8da6bf26964af9d7eed9e03e53415d37aa96045", // vitalik.eth
  }, {
    blockNumber: 10430000,
    address: "0xd8da6bf26964af9d7eed9e03e53415d37aa96045", // replace both of these with your address
  }
];

const qb = ax.newQueryBuilder();
await qb.append(queryData[0]);
await qb.append(queryData[1]);
const { keccakQueryResponse, queryHash, query } = await qb.build();
```

TYPEDEF

```
export interface QueryRow {
  blockNumber: number;
  address?: string;
  slot?: ethers.BigNumberish;
  value?: ethers.BigNumberish;
}
```

Submitting a Query

Build Query | **Submit Query** | Wait for Proof | Parse Proof | Validate Proof

- Call the **sendQuery** function on the AxiomV1Query contract:

example.ts

```
const providerUri = <your_provider_URI (Alchemy, Tenderly, Infura, etc)>;
const provider = new ethers.JsonRpcProvider(providerUri);
const wallet = new ethers.Wallet(process.env.PRIVATE_KEY ?? "", provider);

const axiomV1Query = new ethers.Contract(
  ax.getAxiomQueryAddress() as string,
  ax.getAxiomQueryAbi(),
  wallet
);
const txResult = await axiomV1Query.sendQuery(
  keccakQueryResponse,
  wallet.address,
  query,
  { value: ethers.parseEther("0.01") } // Goerli payment amount
);
const txReceipt = await txResult.wait();
```

Submitting a Query

- Call the **sendQuery** function on the AxiomV1Query contract:

example.ts

```
const providerUri = <your_provider_URI (Alchemy, Tenderly, Infura, etc)>;
const provider = new ethers.JsonRpcProvider(providerUri);
const wallet = new ethers.Wallet(process.env.PRIVATE_KEY ?? "", provider);

const axiomV1Query = new ethers.Contract(
  ax.getAxiomQueryAddress() as string,
  ax.getAxiomQueryAbi(),
  wallet
);
const txResult = await axiomV1Query.sendQuery(
  keccakQueryResponse,
  wallet.address,
  query,
  { value: ethers.parseEther("0.01") } // Goerli payment
);
const txReceipt = await txResult.wait();
```

Important: if the keccakQueryResponse has previously been submitted to AxiomV1Query, then the new transaction will fail.

Proof generation

- Once the Query is successfully submitted via **sendQuery**, the Axiom Prover will generate a proof.
- Once the proof is generated, the Prover will write the status to the AxiomV1Query contract, which will emit the following event:

EVENT

```
event QueryFulfilled(bytes32 keccakQueryResponse, uint256 payment, address prover);
```

Preparing the proof data

- After the proof is generated, get the ResponseTree struct and build it into the format that's required for Axiom's on-chain verifier:

example.ts

```ts
const responseTree = await ax.query
  .getResponseTreeForKeccakQueryResponse(<your keccakQueryResponse>);
const responses = {
  responseTree.blockTree.getHexRoot(),
  responseTree.accountTree.getHexRoot(),
  responseTree.storageTree.getHexRoot(),
  blockResponses: [] as SolidityBlockResponse[],
  accountResponses: [] as SolidityAccountResponse[],
  storageResponses: [] as SolidityStorageResponse[],
};
for (let i = 0; i < queryData.length; i++) {
  const witness: ValidationWitnessResponse = ax.query.getValidationWitness(
  responseTree,
  queryData[i].blockNumber,
  queryData[i].address
) as ValidationWitnessResponse;
if (witness.accountResponse) {
  responses.accountResponses.push(witness.accountResponse);
}
```

Preparing the proof data

Build Query | Submit Query | Wait for Proof | **Parse Proof** | Validate Proof

- After the proof is generated, get the ResponseTree struct and build it into the format that's required for Axiom's on-chain verifier:

example.ts

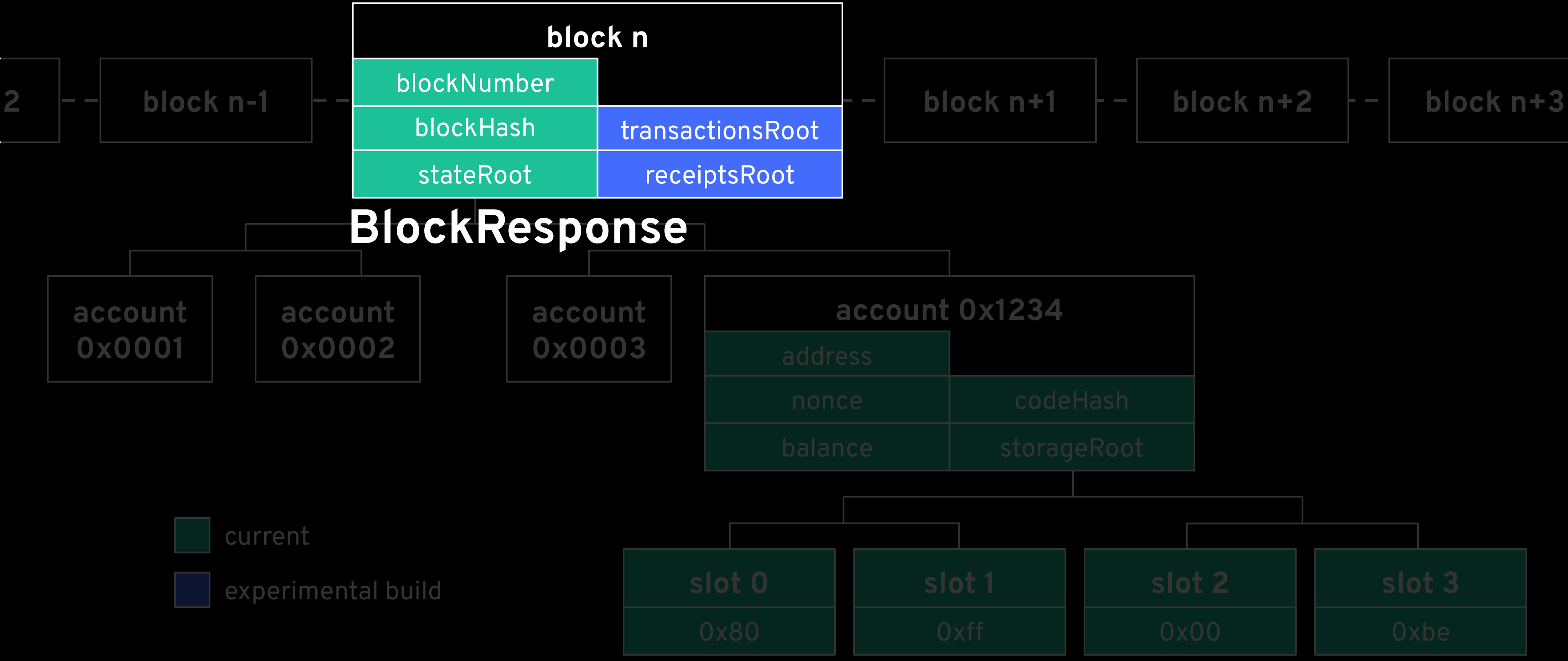
```
const responseTree = await ax.query
  .getResponseTreeForKeccakQueryResponse(<your keccakQueryResponse>);
const responses = {
  responseTree.blockTree.getHexRoot(),
  responseTree.accountTree.getHexRoot(),
  responseTree.storageTree.getHexRoot(),
  blockResponses: [] as SolidityBlockResponse[],
  accountResponses: [] as SolidityAccountResponse[],
  storageResponses: [] as SolidityStorageResponse[],
};
for (let i = 0; i < queryData.length; i++) {
  const witness: ValidationWitnessResponse = ax.query.get
  responseTree,
  queryData[i].blockNumber,
  queryData[i].address
) as ValidationWitnessResponse;
if (witness.accountResponse) {
  responses.accountResponses.push(witness.accountResponse);
}
```

We want to use the ResponseTree's **AccountResponse** since we are looking at account data (nonce)

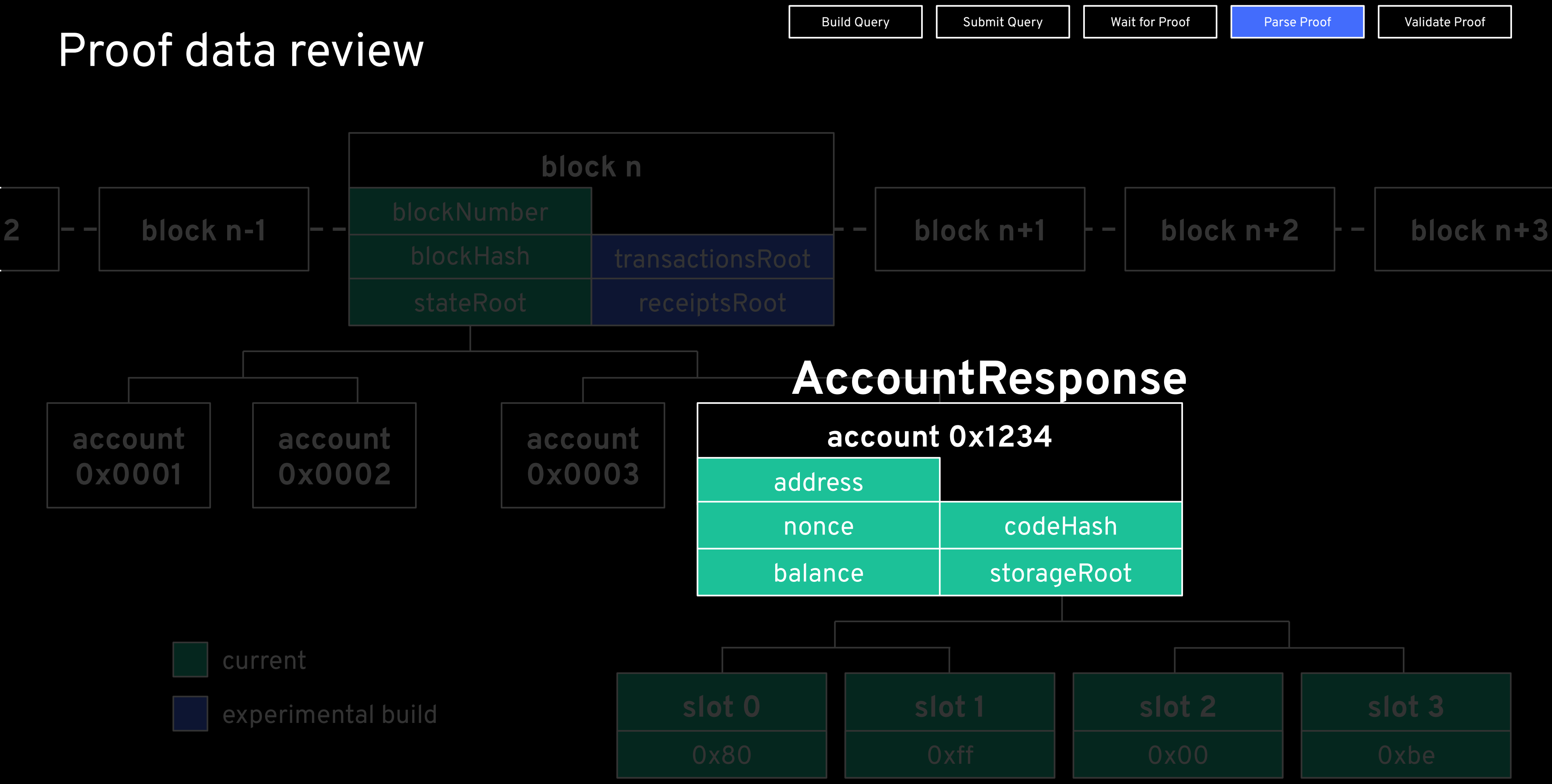
AXIOM

Proof data review

Build Query | Submit Query | Wait for Proof | **Parse Proof** | Validate Proof

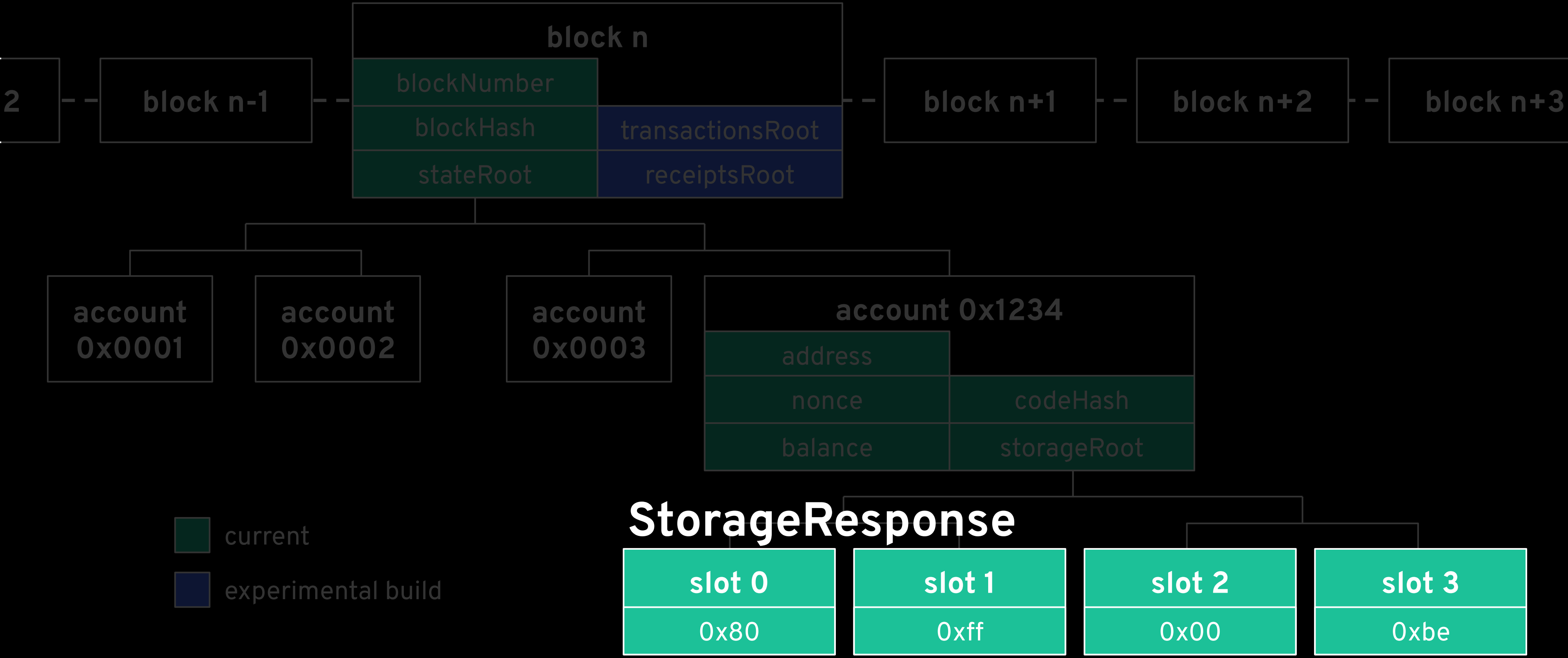


Proof data review



Proof data review

Build Query | Submit Query | Wait for Proof | Parse Proof | Validate Proof



Using the proof in your contract

- Create a struct with the Responses that we will pass proof data to

Distributor.sol

```
struct ResponseStruct {
  bytes32 keccakBlockResponse;
  bytes32 keccakAccountResponse;
  bytes32 keccakStorageResponse;
  IAxiomV1Query.BlockResponse[] blockResponses;
  IAxiomV1Query.AccountResponse[] accountResponses;
  IAxiomV1Query.StorageResponse[] storageResponses;
}

function _validateData(ResponseStruct calldata response) private view returns (bool) {
  ...
}

function mint(ResponseStruct calldata response) external {
  // Validates the incoming ResponseStruct
  _validateData(response);

  // Mints a new NFT to the sender if input validation passes
  _safeMint(msg.sender, totalSupply());
}
```

Using the proof in your contract

Build Query | Submit Query | Wait for Proof | Parse Proof | **Validate Proof**

- Validate the proof with **areResponsesValid** on AxiomV1Query:

Distributor.sol

```
function _validateData(ResponseStruct calldata response) private view returns (bool) {
  // Mainnet AxiomV1Query address
  IAxiomV1Query axiomV1Query = IAxiomV1Query(AXIOM_V1_QUERY_MAINNET_ADDR);
  // Check that the responses are valid
  bool valid = axiomV1Query.areResponsesValid(
    response.keccakBlockResponse,
    response.keccakAccountResponse,
    response.keccakStorageResponse,
    response.blockResponses,
    response.accountResponses,
    response.storageResponses
  );
  if (!valid) {
    revert ProofError();
  }

  // Decode the query metadata
  uint256 length = response.accountResponses.length;
  if (length != 2) {
    revert InvalidDataLengthError();
  }
  ...
```

AXIOM

Using the proof in your contract

Build Query | Submit Query | Wait for Proof | Parse Proof | Validate Proof

Distributor.sol

```
function _validateData(ResponseStruct calldata response) private view returns (bool) {
  ...
  // Get values from start block
  uint256 startBlockNumber = response.blockResponses[0].blockNumber;
  uint256 startNonce = response.accountResponses[0].nonce;
  address startAddr = response.accountResponses[0].addr;

  // Get values from end block
  uint256 endBlockNumber = response.blockResponses[1].blockNumber;
  uint256 endNonce = response.accountResponses[1].nonce;
  address endAddr = response.accountResponses[1].addr;

  // Check that the start and end blocks proved match the values set in the contract
  if (startBlockNumber != BEAR_START_BLOCK || endBlockNumber != BEAR_END_BLOCK) {
    revert InvalidInputError();
  }

  // Check that the account nonce at the end of the bear market is a set threshold above the
  // account nonce at the start of the bear market, since it acts as a transaction counter
  if (endNonce - startNonce < NUM_TX_THRESHOLD) {
    revert NotEnoughTransactionsError();
  }

  // Check that the proof submitted is for the same address that is submitting the transaction
  if (startAddr != msg.sender || endAddr != msg.sender) {
    revert InvalidSenderError();
  }
}
```

AXIOM

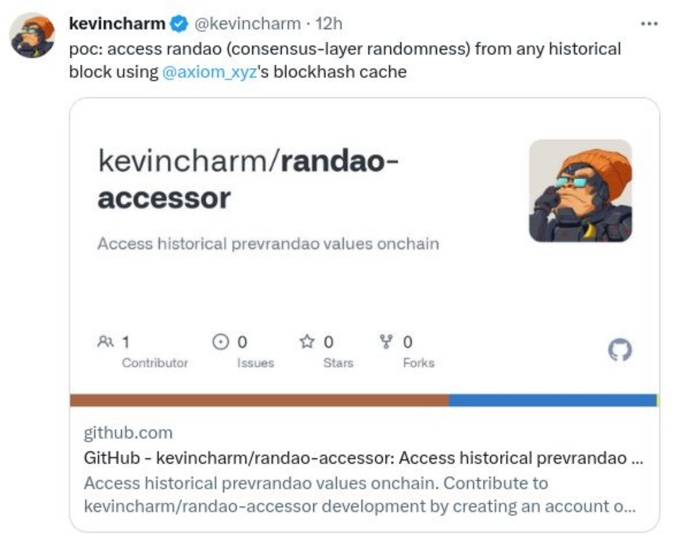
App ideas with Axiom

📜 Identity and Governance

- Autonomous airdrops
- On-chain loyalty systems (volume rebates)
- History-based gating

🔮 Trustless Oracles

- Historic Uniswap LP share pricing
- Maker health factor oracle
- Settlement for derivatives (gas price)
- NFT transacted floor price



App ideas with Axiom

```
/// @title Checks that the price of TUSD never dipped below 0.90 USD
/// @author delalunia.eth
/// @notice Ante Test to check the historical pegging of TUSD
contract AnteTUSDHistoricalPriceTest is
    AnteTest("TUSD has always remained above 0.90 USD")
{
    address public constant AXIOM_V1_QUERY = 0xd617ab7f787adF64C2b5B920c251ea10Cd35a952;
    address public constant TRUE_USD_PRICE_FEED = 0xec746eCF986E2927Abd291a2A1716c940100f8Ba;
    address public constant TRUE_USD = 0x0000000000085d4780B73119b644AE5ecd22b376;

    IAxiomV1Query internal axiom_v1 = IAxiomV1Query(AXIOM_V1_QUERY);
    AggregatorV3InterfaceExtended internal priceFeed;
    address internal aggregator;

    // storage slots
    uint256 public constant S_HOTVARS_SLOT = 43;
    uint256 public constant S_TRANSMISSIONS_SLOT = 44;
```

⛓ On-chain Accountability

- Proof of sandwich
- Proof of Sharpe
- On-chain insurance settlement
- Proof of transaction order within a block

☎️ On-chain Async Calls

- Algorithmic parameter adjustments in DeFi
- Trustless off-chain auction clearing

AXIOM

App ideas with Axiom

😊 On-chain Reputation

- Proof of Whale 🐳
  - Prove you owned at least 5 of an NFT collection before
  - Prove your account owned [>X] of a token
  - Prove you have burned at least 100 ETH in gas
- Uniswap volume oracle
  - Prove you traded at least X volume on a Uniswap pool
- Farmer badges 🧑‍🌾
  - Prove you were an OG Yam 🍠 farmer

Or a creative idea from you!

The ZK proofs behind Axiom

What operations do we need?

Parsing RLP Serialization

Merkle-Patricia Trie Inclusion

Recursion and Aggregation

All data in Ethereum is serialized with:

RLP = **R**ecursive **L**ength **P**refix

RLP is a method to serialize arbitrary nested bytearrays

- The serialization is **recursive**.
- Each RLP-serialized piece of data has a **prefix byte** and optional **length bytes** prepended to the data.
- These bytes determine the length of the next field.

AXIOM

What operations do we need?

Parsing RLP Serialization

Merkle-Patricia Trie Inclusion

Recursion and Aggregation

Key ZK primitives: **variable-length array manipulation**

- Indexing into an array
- Selecting a variable length subarray
- Concatenation of variable length arrays

Key arithmetization idea: **R**andom **L**inear **C**ombination

- After committing to arrays **a[i]**, **b[i]**, draw randomness **r**, and encode by

$$RLC(a[i], r) := (len(a), a[k]\, r^{k-1} + a[k-1]\, r^{k-2} + \ldots + a[0]$$

- If RLC(a[i], r) = RLC(b[i], r), then **a = b**.

AXIOM

What operations do we need?

Parsing RLP Serialization

Merkle-Patricia Trie Inclusion

Recursion and Aggregation

Data in Ethereum is committed to in:

MPT = **M**erkle **P**atricia **T**rie

This is a 16-ary trie where each node is RLP encoded.

Key ZK primitives:

- Keccak hash (expensive!)
- RLC for subarray checks

What operations do we need?

Parsing RLP Serialization

Merkle-Patricia Trie Inclusion

Recursion and Aggregation

Combine block header and MPT proofs with **aggregation.**
Given proofs pi_1, pi_2, ..., pi_n, we create a recursive verifier:

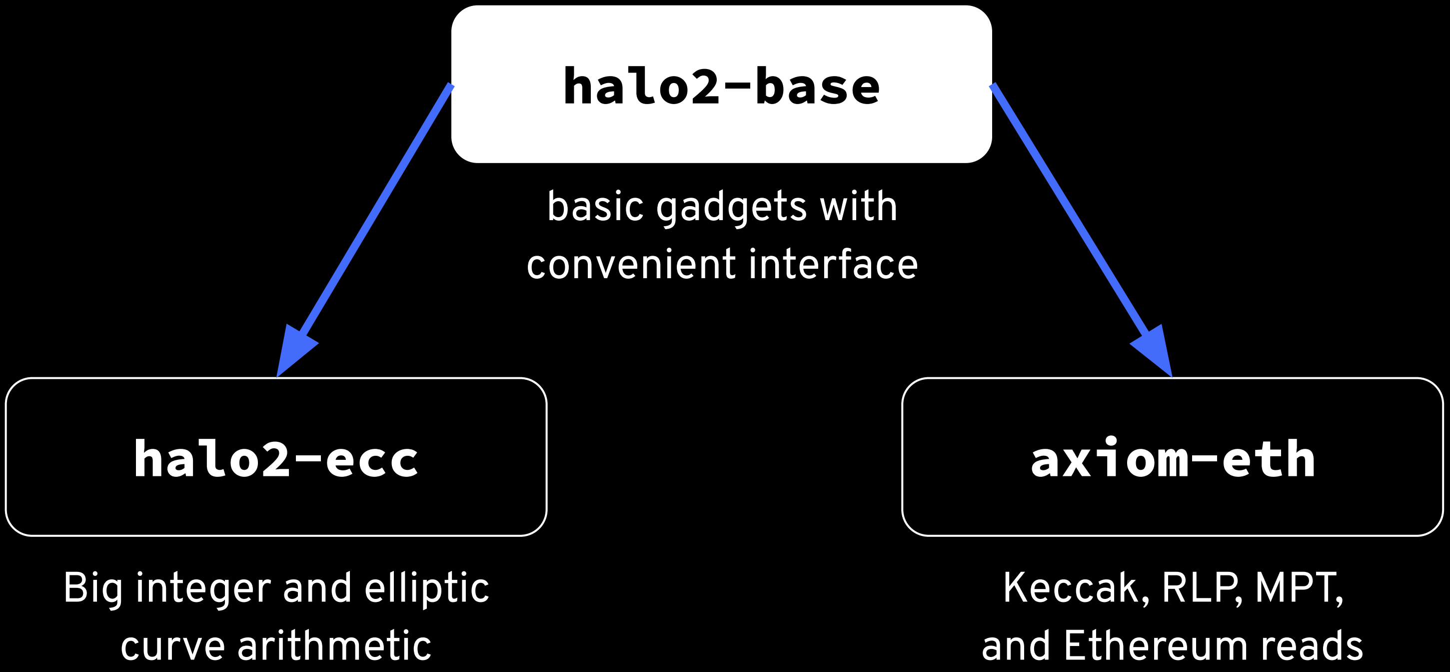
pi: all of pi_1, pi_2, ..., pi_n hold

Key ZK primitives:

- **Non-native** elliptic curve arithmetic
- Multi-scalar multiplication
- (Optionally) elliptic curve pairing verification

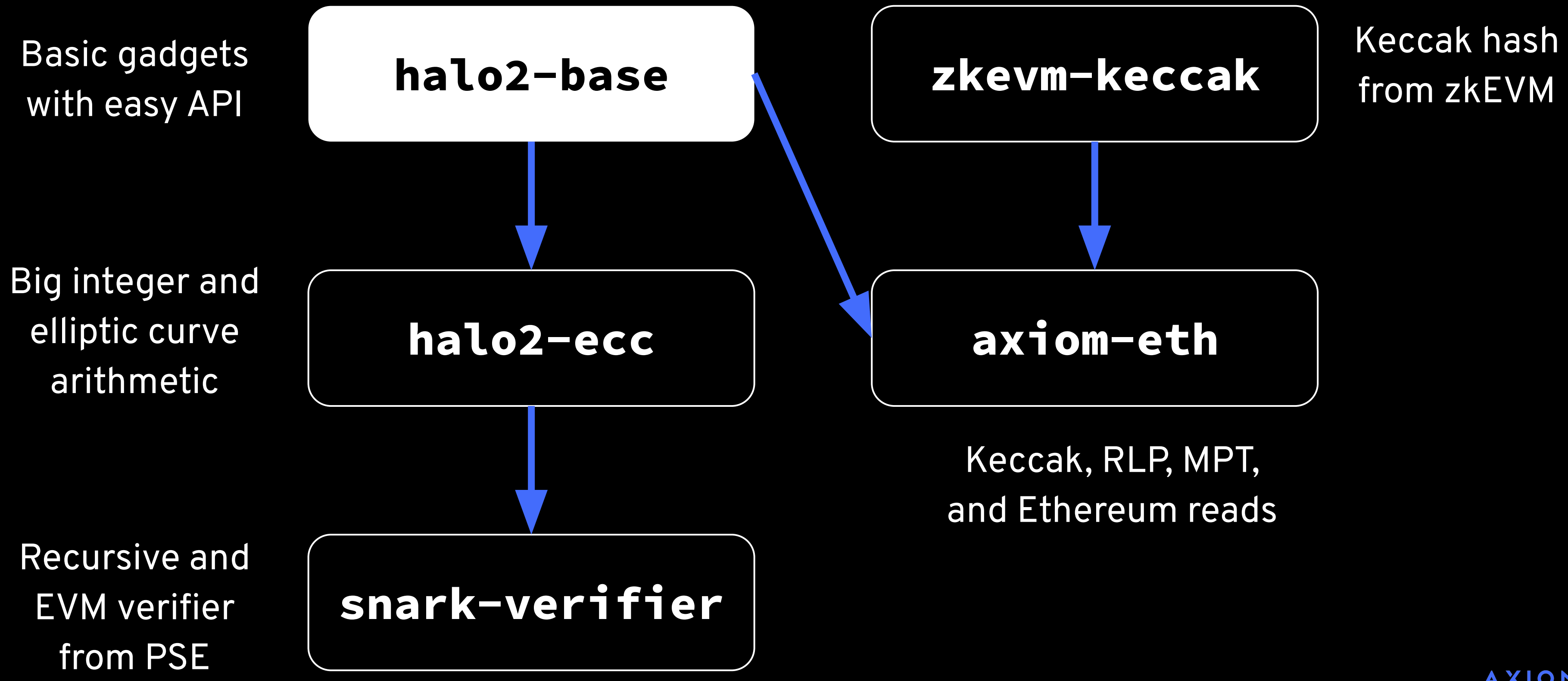
How do we build these primitives?

We use **halo2** with KZG backend (PSE fork) with a modular setup:



AXIOM

How do we build these primitives?



AXIOM

halo2-base: Vertical Gate

Cheap Verifier Setting:

- 1 advice, 1 lookup table, 1 constant, 2 selector columns
- 1 custom gate: a + b * c = d

Overlap Optimization:

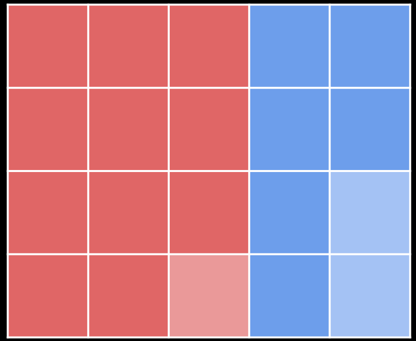
- Example: Dot product of (1, 3) with (2, 4)
- Length N dot product uses 3 N + 1 instead of 4 N cells

0	1			
1	0			
2	0			
2	1			
3	0			
4	0			
14	0			

halo2-base: Configurable Prover-Verifier Tradeoff

Give desired number of advice and fixed columns:

- Allocate basic gates evenly across columns
- Library of basic gadgets in this form:
  - Inner product
  - Range check
  - Index into array
  - Bitwise operations
  - Comparison operators



We found it difficult to outperform the basic gate using more custom gates.

halo2-base: Shared Lookup Arguments

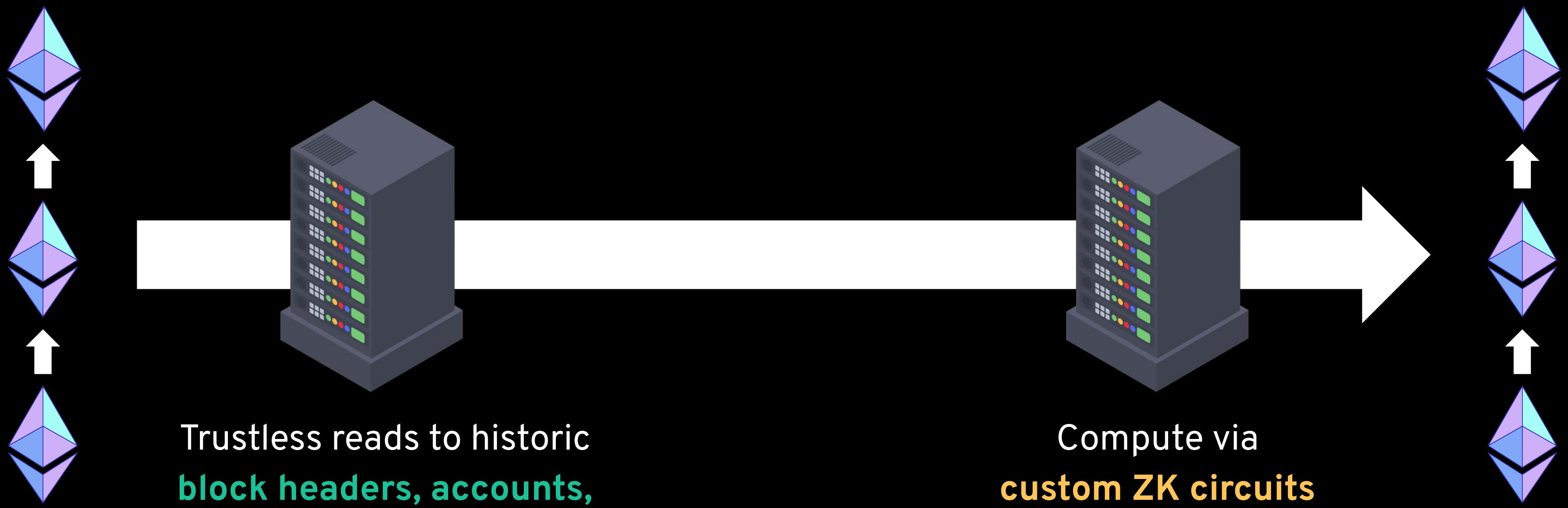
Previously: Enable lookup arguments on every advice column.

Optimization: Copy lookup values to special advice columns with lookup enabled

- User-specified number of special advice columns
- Allocate lookup values to special advice columns evenly
- Gives ~50pct proving speed improvement

Our Vision

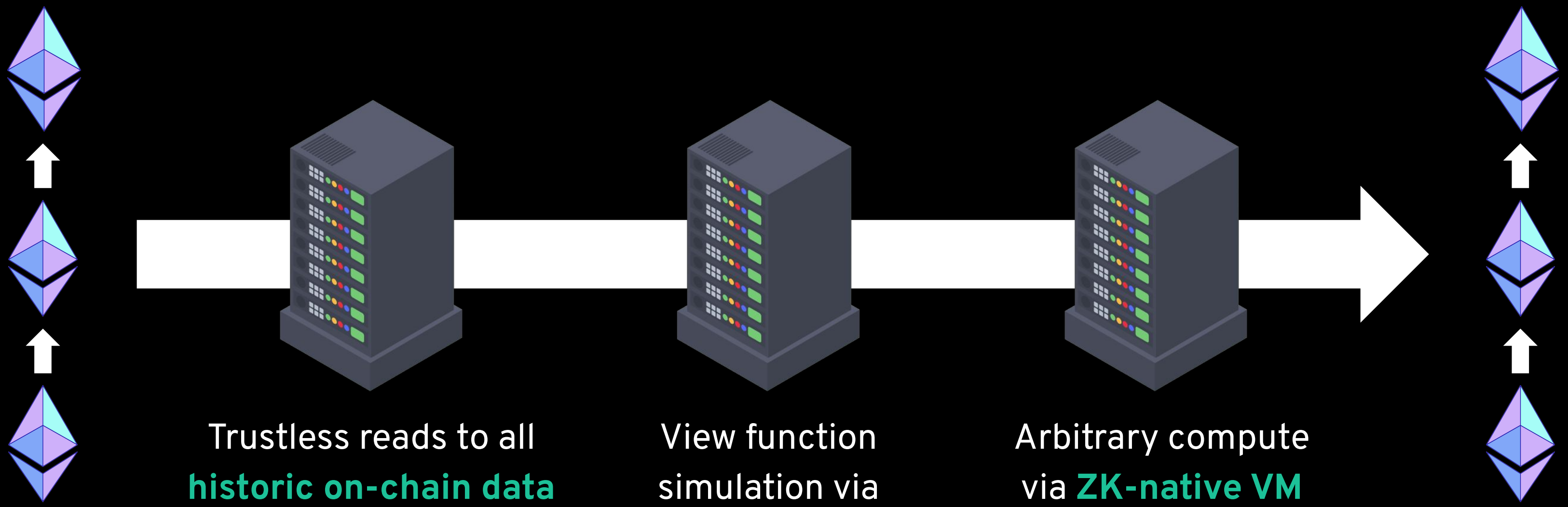
ZK coprocessing with Axiom today



Trustless reads to historic
block headers, accounts,
account storage,
transactions, and receipts

Compute via
custom ZK circuits

Our vision for ZK coprocessing with Axiom



Trustless reads to all **historic on-chain data**

View function simulation via **zkEVM proofs**

Arbitrary compute via **ZK-native VM**

ZK Archive Node and Indexer

AXIOM

The ZK Coprocessor for Ethereum, live on mainnet today!

Start building with Axiom

docs.axiom.xyz

Code examples

github.com/axiom-crypto/examples

We empower developers to build a new class of **data-rich applications** combining the rich interactions of traditional webapps with the security of Ethereum.