L1StateOracle

2023 ZK Hackathon project ***(zk-hacking)*** *(https://zk-hacking.org/)*

Problem

Currently, there is no way for L2s to access L1 state in a *trustless, cheap and easy way*. One option is to use arbitrary messaging bridges to send over the L1 state, but in this case you need to rely on the honesty of the messenger. Another option is to set up a specific purpose bridge (think ERC20 or ERC721 token bridge) so that you don't need to trust the messenger anymore. But this is not generalizable and costly since you need to create a bridge for every single purpose. So our question was, is there a better way to send over L1 state to L2s?

Our approach

Instead of creating an entirely new system from scratch, we took advantage of two existing systems to create a solution to this problem. We were inspired by the Hashi team (**ethresearch post** (https://ethresear.ch/t/hashi-a-principled-approach-to-bridges/14725/1), **presentation** (https://docs.google.com/presentation/d/1yMdO179XFJeeryIqsJg8L4RewH8jaA_p97iCO-vI9mY/edit#slide=id.g21cefba53b5_0_148)) to combine two existing systems to create a solution.

One is **Hashi** (https://github.com/gnosis/hashi), which is a system that provides additive security for bridge systems. Essentially, it improves security by allowing L2 protocols to not rely on a single bridge system. Under the hood, it is connected to multiple bridges deployed on L2 and provide aggregate L1 block hash data to L2 protocols. As a result, L2 protocols that rely on a bridge system can avoid being hacked when a single bridge is compromised.

Another is **Axiom** (https://www.axiom.xyz/), which enables accessing any historic state on-chain via smart contracts. Storing historic states requires a lot of storage, so it's normally unaffordable on-chain, but Axiom leverages ZK proofs to make this cheap. One thing to point out is that Axiom is

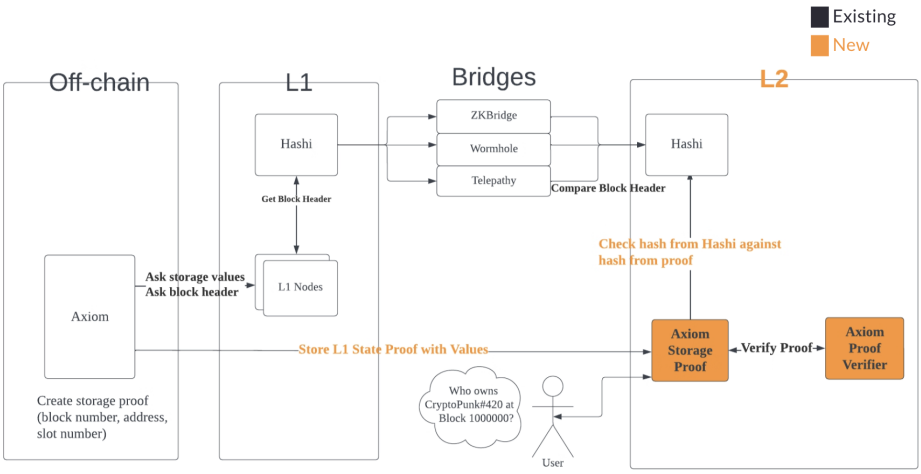
currently intended to be used only on L1, but the system is modular so we were able to think about porting a part of it on L2.

Solution

> Axiom ✍️ + Hashi 橋 => L1StateOracle (Time Travel 🚀 L1 state on L2)

Our approach is to take the proof module of Axiom and to integrate it with Hashi. Below you can check out our architecture and how it leverages Axiom and Hashi's existing architecture.

L1StateOracle Architecture



As you can see in the flow chart above, we created new `AxiomStorageProof` and `AxiomProofVerifier` contracts and deployed them on L2.

Once a user creates a storage proof using **Axiom's backend** (https://demo.axiom.xyz/custom), it can send the proof to the L2 contract, which will verify the block hash used in the proof against Hashi's `getHash` function.

When the ZK proof itself is also verified via the `AxiomProofVerifier`, we can safely store the storage proof on-chain, and *voilà*! **Any L2 protocol can confidently use the attested storage data without worrying about a single bridge being compromised.**

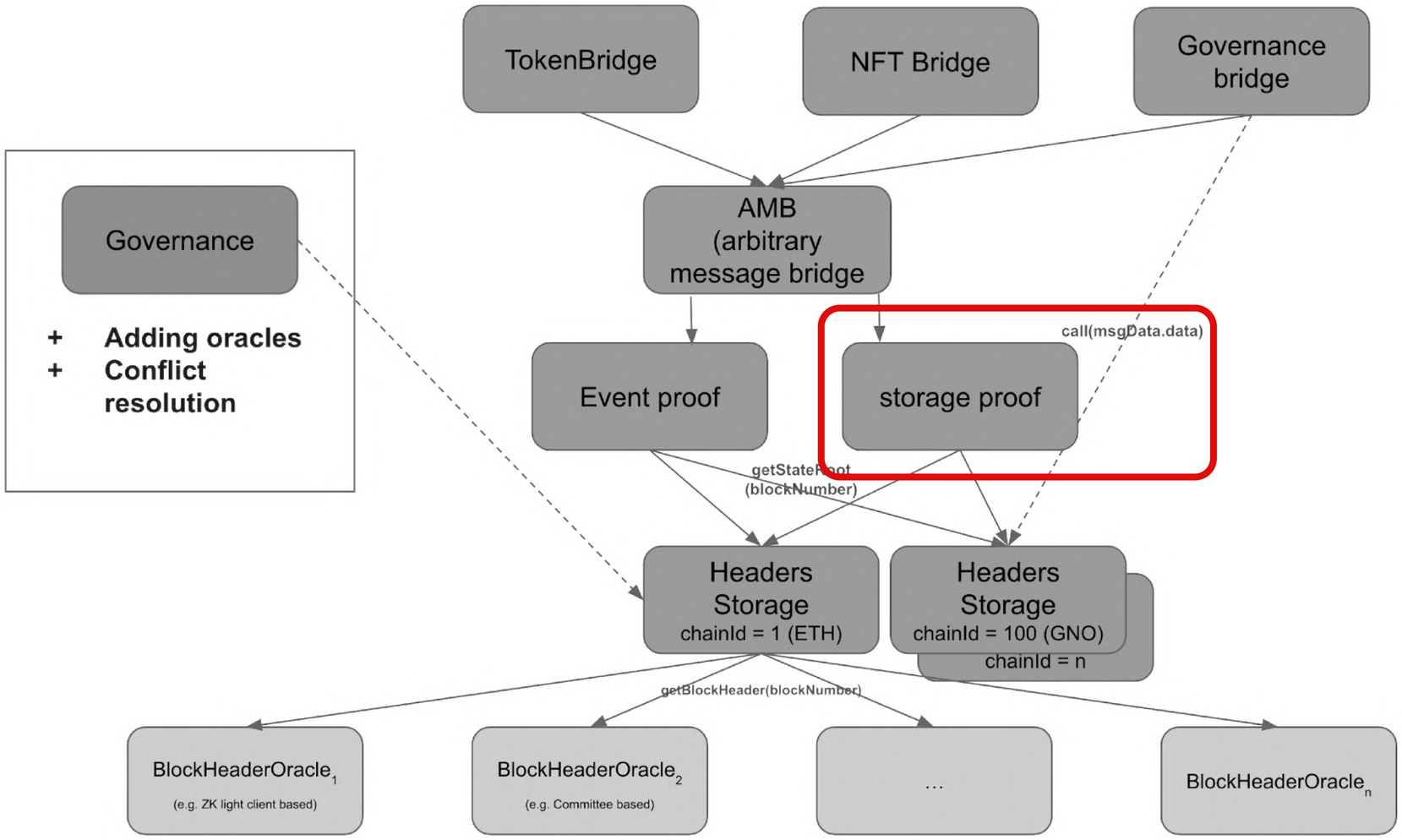
L1StateOracle

Trustlessly Time Travel 🚀 L1 state on L2

Problem

- ZK bridges enable trustless messaging between L1 and L2 chains
- But they don't give L2s access to L1 on-chain data
- e.g. I want to check the ownership of CryptoPunk#420 at block 10000000

Inspiration



https://ethresear.ch/t/hashi-a-principled-approach-to-**bridges**/14725

Technical Introduction

Hashi - An EVM hash oracle aggregator by Gnosis

- Provides additive security to bridges
  - By comparing L1 block header hashes from multiple bridges

Axiom - The ZK Coprocessor for Ethereum

- Allows trustless access to historic on-chain data from your smart contract
  - By creating ZK proofs of any L1 historical state,
  - And verifying proofs on-chain via block header hashes

Solution

Hashi 橋 => Reliable L1 block header on L2

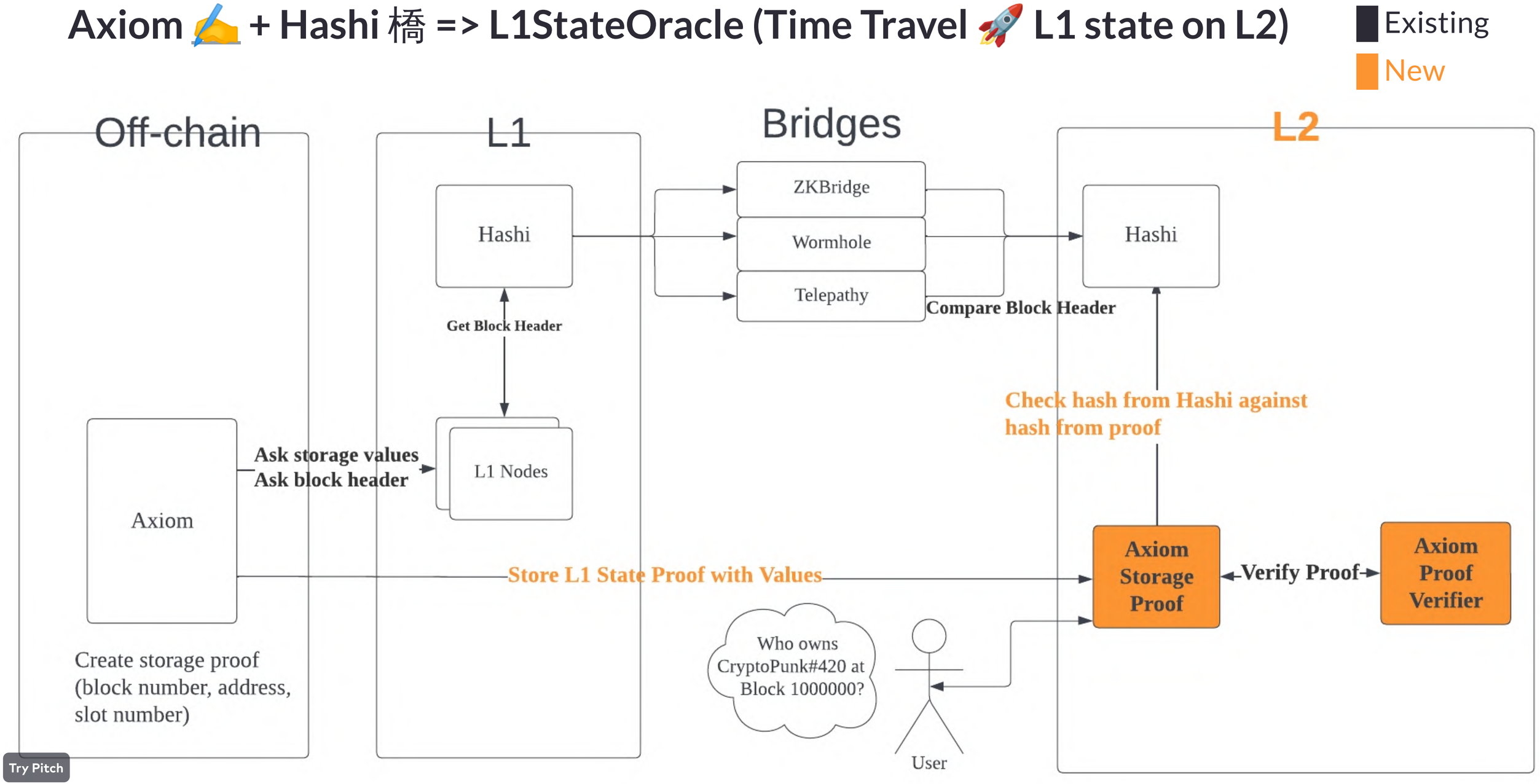
\+

Axiom ✍️ => Provable L1 state associated with the block header

=

L1StateOracle => **Trustlessly Time Travel 🚀 L1 state on L2**

Axiom ✍️ + Hashi 橋 => L1StateOracle (Time Travel 🚀 L1 state on L2)



VERIFY

FOR ANYONE
Account age

FOR PROTOCOLS
Token price (V2)

FOR PROTOCOLS
Token price (V3)

FOR ORACLES
Randomizer

Custom

Custom

Block number
10000000

Address
0xb47e3cd837dDF8e4c57F05d70Ab865de6e193

Storage slot How do I find this?
0f92f3ad435570e9f610d535ca71c2a4c5ef34aa438e925fe55dbb614b291b4b

Value at slot: 0x000000000000000000000000c352b534e8b987e036a93539fd6897f53488e56a

⊕ Add slot

Regenerate proof

```
{
  "block": "10000000",
  "address": "0xb47e3cd837dDF8e4c57F05d70Ab865de6e193BBB",
  "slots": [
    {
      "value":
"0x000000000000000000000000c352b534e8b987e036a93539fd6897f5348
8e56a",
      "slotNumber":
"0f92f3ad435570e9f610d535ca71c2a4c5ef34aa438e925fe55dbb614b291b4b
"
    }
  ],
```

Deeper Dive

Axiom Storage Proof contract

- verifies the proof with an on-chain verifier

```
(bool success,) = verifierAddress.call(proof);
if (!success) {
    revert("Proof verification failed");
}
```

- parses the data from Axiom to get up to 10 proofs of storage slots
- saves attestations based on hash of **(blockNumber, account, slot, slotValue)**

```
uint16 i = 0; // slot number to find
uint256 slot = (uint256(bytes32(proof[384 + 128 + 128 * i:384 + 160 + 128 * i])) << 128)
        | uint128(bytes16(proof[384 + 176 + 128 * i:384 + 192 + 128 * i]));
uint256 slotValue = (uint256(bytes32(proof[384 + 192 + 128 * i:384 + 224 + 128 * i])) << 128)
    | uint128(bytes16(proof[384 + 240 + 128 * i:384 + 256 + 128 * i]));
(bytes32 hashedVal) = keccak256(abi.encodePacked(blockData.blockNumber, account, slot, slotValue));
slotAttestations[hashedVal] = true;
// CryptoPunk#420 address at the slot for the given block
return address(uint160(slotValue));
```

Demo

https://github.com/gnosis/hashi/pull/11

```
const expectedAddress = await storageProof.callStatic.attestCryptoPunk420AddressWithHashi(
  blockHashWitness,
  proof,
  CHAIN_ID,
  [ambAdapter.address, ambAdapter.address]
)
expect(
  expectedAddress.toLowerCase()
).to.equal(CryptoPunk420OwnerAtBlock10Mil.toLowerCase())
```

```
  End-to-end tests
    Execution layer
      ✔ Attest slots for the claimed block head with the block hash agreed on by N adapters (2459ms)
      ✔ Reverts if the claimed block header is different from the block hash agreed on by N adapters (143ms)
    Execution layer
Expected:  0xc352b534e8b987e036a93539fd6897f53488e56a Got:  0xc352b534e8b987e036a93539fd6897f53488e56a
      ✔ Get the correct cryptopunk#420 owner address with the proof (150ms)


  3 passing (3s)
```

Yes, the owner was 0xc352b534e8b987e036a93539fd6897f53488e56a at block 10,000,000

✍️ Thank You 🚀

Hire me!

resume.sjlee.me