Authors: Garvit Goel, Jinank Jain (Electron Labs)

This is an article on how to bring IBC to Ethereum. The goal of the article is to provide an overview of the technical details of this project and gather support from the Ethereum community. Let's dive into it.

IBC stands for Inter Blockchain Communication - the cross-chain standard in the Cosmos ecosystem https://ibcprotocol.org/

Problem background

IBC works on the light client principle where the light clients of the origin and destination blockchains need to be implemented as smart contracts in order to verify cross-chain transactions.

This means that in order to connect IBC to Eth, we will need to run the tendermint light client on Ethereum as a solidity smart contract. However, this turns out to be an extremely gas expensive operation since this requires verification of hundreds of ed25519 signatures in solidity, and ed25519 pre-compiles are not available on Ethereum. One ed25519 costs 500K gas. This means that verifying full light client headers would cost at least 50 mn gas (100 validators) and go up to 500 mn for larger cosmos chains with 1000 validators.

Hence we must find an alternative to verify these signatures cheaply on Ethereum.

Solution

We have achieved this by taking inspiration from zk-rollups. Rather than verifying the ed25519 signatures directly on Ethereum, (and performing the curve operations inside a solidity smart contract), we construct a zk-proof of signature validity and verify the proof on-chain instead.

At Electron Labs, we have built a circom-based library that allows you to generate a zk-snark proof for a batch of Ed25519 signatures. Check out the complete implementation here

How to try this out?

We have deployed a server whose endpoints allow you to submit a batch of signatures and get the zk-proof in return. You can test this out right now using the API reference given in our docs - docs.electronlabs.org/reference/generate-proof

Details of our Mathematical Approach

Creating a ZK-prover for ed25519 is a hard problem. This is because ed25519's twisted Edwards curve uses a finite field that is larger than that used by the altbn128 curve (used by zk-snarks). Performing large finite field operations inside a smaller field is difficult because several basic operations such as modulo and multiplication can become very inefficient.

To solve this problem, we were able to find 2^85 as a base over which to define our curve operations for twisted Edwards curve. Since the ed25519 prime p = 2^255 - 19 is a close multiple of 2^85, we were able to come up with efficient basic operators such as multiplication and modulo (under 25519 prime) for base2^85 numbers.

Next, we used these custom operations to define curve operations such as point addition, scalar multiplication, and signature verification inside our ZK-circuit.

It is hard to do justice to the details of the mathematics behind this in this doc, please refer to our detailed docs explaining this given here.

Performance of Single Signature Proof

As a result of the above optimizations, we were able to achieve the following performance figures for a single signature.

Circuit Performance for Single ED25519 Signature

Constraints

2,564,061

Circuit compilation

72s

Witness generation

6s

Trusted setup phase 2 key generation

841s

Trusted setup phase 2 contribution

1040s

Proving key size

1.6G

Proving time (rapidsnark)

6s

Proof verification Cost

~300K gas

*All metrics were measured on a 16-core 3.0GHz, 32G RAM machine (AWS c5a.4xlarge instance).

Performance of Batch Prover

To understand the performance at a system level, we need to look at 3 parameters-

- Proof generation time per signature ~ 9.6s (averaged out)
- Number of Signatures per batch/proof = ≤ 100 (maximum value)
- Time to generate zk-proof for the batch = 16 mins for 100 signature batch

The proof generation time scales linearly (almost) with respect to the number of signatures per batch. We can increase/decrease the number of signatures per batch and the proof generation time changes accordingly.

Proof production time will be visible as latency. To reduce this, we can put a lesser number of signatures in one zk proof. However, this means more proofs will be required for the same batch size (or per light client header), which will increase the gas cost of verifying that batch.

Hence, reducing the latency will increase the gas cost. Below we have laid out the expected cost of verifying a tendermint Light Client (LC) header on Ethereum as a function of latency and the number of validators participating in that cosmos chain. We can give the users/cosmos chains the option of deciding the latency and gas fees they wanna to work with.

Number of Validators

Latency (minutes)

Number of Signatures per Proof

Tx Cost per Light Header ($)

Cost Reduction (X) achieved by using ZK

200

16

100

9.0

166.7

500

16

100

22.4

166.7

1000

16

100

44.8

166.7

10000

16

100

448.5

166.7

200

8

50

17.9

83.3

500

8

50

44.8

83.3

1000

8

50

89.7

83.3

10000

8

50

896.9

83.3

200

2

12

76.2

19.6

500

2

12

188.4

19.8

1000

2

12

376.7

19.8

10000

2

12

3740.3

20.0

*based on gas prices on 5th August 2022.

We have selected 200 validators and 50 signatures per proof as the base case for further analysis.

Cost of Relayer Infrastructure

Since the tendermint block production rate is ~ 7 sec and the proof generation time is 8 minutes, we will need multiple prover machines in parallel to keep up with the block production rate.

Number of parallel machines required = 8 mins *60 / 7 sec = 69 machines

We recommend using m5.8xlarge AWS cloud instance for proof generation.

Hence the cost of this infra = $1.536*69 = $106 per hr

Machine Cost per light client header = 106/3600/7 = $0.206

Estimating Total Transaction Cost

Consider the case for 8 mins latency and 200 validators.

Total Cost of on-chain light client verification = $17.9 + $0.206 = $18.1

Let us assume a worst-case scenario (from a tx fees point of view) when only one cross-chain transaction is present in one block. Then the entire cost of verifying the LC header is borne by that transaction. Adding some overhead cost, then verifying the cross-chain transaction is ~$ 20.

Assuming an optimistic case when there 10 transactions per block, this cost will be ~$2 which is similar to the cost of a Uniswap transaction on Ethereum.

How can we reduce the gas cost and latency (using recursive)?

In order to reduce latency down to seconds and gas costs down to a few cents per transaction, we are working on recursive proof technology. This will enable us to generate multiple proofs together in parallel and then recursively combine them into a single proof.

We are evaluating various recursive libraries available in the market such as plonky2, and the works by Mina, Aztec and Starknet teams. We invite anyone working on recursive to connect with us.

By use of recursion and the use of hardware-based acceleration, we believe we can achieve sub-5 second latency for cross-chain transactions.

In the future, we can even combine multiple light headers in a single proof, costing just $4.5 per proof, and potentially <$1 per cross-chain transaction.

System Level Design Overview

Current IBC Design (Simplified)

[

image

2608×1044 128 KB

](https://ethresear.ch/uploads/default/original/2X/9/9ba15634424aa84d4555f9b63f29bfb1b54552b7.png)

Proposed IBC Design

[

image

2684×1644 223 KB

](https://ethresear.ch/uploads/default/original/2X/0/0f6c56c2db0024643ee38e1f8c1ed0e104a88e3e.png)

Points to note regarding proposed design

1. The IBC interface stays the same. This makes adoption very easy since no new developer docs and developer re-education is required. The existing code bases will also get used as it is.
2. No Governance updates are necessary on app-chains
3. Two changes are required to IBC on Ethereum side-
4. The relayer, rather than submitting the full light client header, will now just submit the proof of validity for the same.
5. The on-chain light client modules on Ethereum side will include a zk-proof verifier instead of ed25619 signatures verifier.
6. The relayer, rather than submitting the full light client header, will now just submit the proof of validity for the same.
7. The on-chain light client modules on Ethereum side will include a zk-proof verifier instead of ed25619 signatures verifier.

What Next?

We invite the Ethereum and ZK community at large to provide their comments and help us gather support to make this proposal a reality.

Execution Plan:

Phase1: Integration of our ZK engine with IBC

Phase2: Bringing down latency to ~5s through recursive proofs and hardware acceleration.

Phase3: Deploy a demo-app chain that uses connects to Ethereum via zk-IBC.

Phase4: Run the demo app-chain setup for extensive testing, and enable the community to test out transactions

Phase5: Security Audits

Phase6: Mainnet Deployment