I am very happy to share the implementation of Zkopru, a privacy layer of Ethereum. From last November@barryWhiteHat and I started to build this project, and here is the result.

Special thanks to V, W, K, J, JP, L, and A who helped, reviewed, and supported this project.

TL;DR

What's this?

- It's a layer 2

solution for private transactions

.

- It uses optimistic rollup

to manage blocks.

- It uses zk SNARK

to build private transactions.

Performance

- 8800 gas

/ private tx on Ethereum.

- Max TPS is 105 when the gas limit is 11.95M & block time is 13.2s.

Awesome features

- It supports ETH, ERC20, and even ERC721

.

- It also supports private atomic swap

. It can be used with the private order matching system.

- Subtree rollup cuts the cost about 20 times

of the merkle tree update challenge.

- Instant withdrawal

before block finalization.

- Using mass deposit & mass migration

we can construct an inter-layer-2 network.

And, most of all, it is rolling up blocks now! Visit and try it on our testnet!

https://zkopru.network

Introduction

Zkopru is a layer-2 scaling solution for private transactions using zk-SNARK and optimistic rollup. It supports private transfer and private atomic swap within the layer-2 network between ETH, ERC20, ERC721 at a low cost. Also, with the pay-in-advance feature, users can withdraw assets from the layer-2 before the finalization.

Transaction

A zk transaction accepts several UTXOs as its inflow and creates new UTXOs for its outflow. Therefore, the most important thing is to validate the inflow and outflow.

[

zkopru transaction

1447×1377 215 KB

](https://ethresear.ch/uploads/default/original/2X/7/7d555c11d3ac74c4a0db6106b16f805428556127.png)

Inflow validation

Zkopru achieves privacy using the commitment-nullifier scheme. It means that a zk transaction spends a UTXO while not revealing which note has been used. Instead, we reveal the nullifier derived from the UTXO but not possible to link with its original UTXO.

To spend the UTXOs, it should satisfy the following conditions:

UTXO membership proof

The tx builder submits Merkle proofs of every UTXO to prove its existence. For an effective SNARK computation, the UTXO tree uses Poseidon for its hash function.

Ownership proof

Only the owner should be possible to spend the UTXO. For this condition, every note has a public key field, a Babyjubjub point. Using the paired private key, the owner can create an EdDSA signature to prove its ownership.

Commitment proof

The circuit should have detailed information about the input UTXOs to calculate the total sum of the inflow. Therefore, the owner should provide the details, and its Poseidon hash should equal the leaf hash of the Merkle proof and the ownership proof.

Nullifier proof

The given nullifiers should be correctly derived from the input UTXOs.

Outflow validation

A zk transaction can create three types of outputs: UTXO, Withdrawal, and Migration. If the zk transaction creates UTXOs, Zkopru appends them to the UTXO tree. When it creates withdrawal outputs, Zkopru appends them to the withdrawal tree. Lastly, mass migration, which is a part of the layer-2 block, comprises the migration outputs of the block’s every zk transaction.

Therefore, the outflow should satisfy the followings:

1. When the output is a UTXO type, the public hash value of the output equal the calculated in the SNARK circuit.
2. If the type of output is withdrawal or migration, it should reveal the details because it should move the correct amount of assets to the outside of the network.

Zero-sum proof

Finally, the zk transaction should guarantee that the inflow equals the outflow, including the fee.

Block structure

[

image

1342×1189 122 KB

](https://ethresear.ch/uploads/default/original/2X/7/7ca5c3fb18baa85ea1c4b1f5d9042ba9207103e8.png)

Header

The first 372 bytes of the data should be the block header. The block header contains the following data:

Property

type

Proposer’s address

address

Parent block hash

bytes32

Metadata

uint256

Fee

uint256

Latest UTXO tree’s root

uint256

Latest UTXO tree’s leaf index

uint256

Nullifier tree’s root

bytes32

Latest withdrawal tree’s root

uint256

Latest withdrawal tree’s leaf index

uint256

Transactions’ root

bytes32

Deposits’ root

bytes32

Migrations’ root

bytes32

Body

A block body consists of transactions, mass deposits, and mass migrations. Moreover, the block header should contain correct information from the body. If the header does not have the correct value, the proposer will get slashed through the challenge system.

Transactions

[

image

958×843 71.6 KB

](https://ethresear.ch/uploads/default/original/2X/d/d592b30cb05b28ecf9628c96d0ff97aeab30ba3e.png)

Mass deposits

[

image

960×720 28.6 KB

](https://ethresear.ch/uploads/default/original/2X/f/fbab0542d90ce861f07565468aa53ddd5075f4c0.png)

Mass migrations

[

image

960×720 50 KB

](https://ethresear.ch/uploads/default/original/2X/7/787be20995fd6d83af417450d2907434d15e1ed5.png)

Account

A new public key structure is going to be proposed. Note that this specification will be updated.

A zkopru account manages both of the layer-1 and layer-2 key pairs. First of all, the account has an Ethereum account from a randomly generated private key. This is used for interactions on the layer-1. Second, Zkopru wallet creates a Babyjubjub private key and public key set from the Ethereum account's private key. This Babyjubjub keyset is used for the EdDSA and encrypted memo field on the layer-2.

Key	Size	How to get	Where to use
Master seed	32 byte	Randomly generated	Recover keys
Ethereum private key	32 byte	Derived from master seed with BIP39	Layer1 ECDSA
Ethereum public address	20 byte	Derived from the private key	Layer 1 interactions
Babyjubjub private key	254 bit	Derived from Ethereum private key	Create EdDSA to spend UTXO
Babyjubjub public key	(254 bit, 254 bit)	Derived from Babyjubjub private key	Proving ownership of a UTXO

UTXO

A new UTXO spec is also going to be proposed soon.

Property	Description
Ether	The amount of ETH
Public Key	A Babyjubjub point of the note owner
Salt	Random salt. Zkopru generates UTXO hash using this salt.

Token Address (Optional)

The address of the token contract when it includes ERC20 or ERC721. The default value is 0.

ERC20 Amount (Optional)

The amount of the ERC20 token when it includes ERC20. The default value is 0.

NFT Id (Optional)

The id of the ERC721 token when it includes ERC721. The default value is 0.

And then Zkopru computes the leaf hash with Poseidon hash:

var intermediate_hash = poseidon(ether, pub_key.x, pub_key.y, salt) var result_hash = poseidon(intemediate_hash, token_address, erc20, nft)

How does the recipient know?

A zk transaction can include 81 bytes of encrypted memo field for the recipient. Because of the zero-knowledge characteristics, even the recipient cannot know the reception without an interactive process. Therefore, to keep the non-interactive way, we can put some encrypted data in the memo field for the recipient.

Encryption

It generates the shared key using the Diffie-Hellman Key exchange protocol. The details steps to produce the shared key for the sender is:

1. Create an ephemeral key and its homomorphic hidden value.

ephemeral = e\ public_ephemeral = g^e

1. Multiply its ephemeral key to the recipient's public key

recipient_pubkey = g^a\ shared_key = (g^a)^e

1. Prepare the compressed data to encrypt

data = { salt // 16 byte tokenId, // 1 byte value, // 32 byte }

1. Encrypt the data using chacha20 algorithm and create the memo data with the ephemeral public key.

ephemeral = random.new() public_ephemeral = generator.multiply(ephemeral) shared_key = recipient_jubjub.multiply(ephemeral) ciphertext = chacha20.encrypt(data, shared_key) memo = public_ephemeral + ciphertext

Decryption

Using the Diffie-Hellman key exchange protocol, the recipient also creates the shared key using the public ephemeral key and the private key.

1. Parse memo and get shared key

public_ephemeral, ciphertext = parse(memo) shared_key = public_ephemeral.multiply(private_key)

1. Decrypt the ciphertext using the shared key

decrypted = ciphertext.decrypt(shared_key)

1. Using the decrypted result, the recipient tries to make various possible UTXOs. It is because the encrypted data has only 49 bytes to minimize the calldata size. Therefore, the recipient should try various combinations to check whether the transaction includes the recovered UTXO hashes. If it fails to find the recovered UTXOs in the tx, it considers that the tx has no output for the recipient.

[

image

1024×959 96.2 KB

](https://ethresear.ch/uploads/default/original/2X/5/5a2a3ae10211b52ae7344de4d9dc34336a87488a.png)

Compressed data

To minimize the calldata, Zkopru compresses original data into 49 bytes data. First of all, it gets rid of the public key from the encryption candidate because the recipient will infer that using the own public key. And it uses the Token ID, which maps the supported token addresses and the indexes from 0 to 255. And then because the value

can be ether

, erc20Amount

, or nftId

, the recipient creates three types of UTXOs for those 3 cases. Finally, if any of the inferred UTXO exists in the transaction's output list, the recipient successfully receives the UTXO.

Limitation

Zkopru does not enforce the circuit to check the encryption protocol. Therefore, if the sender does not use an appropriate shared key or data, the recipient will not receive the note.

Atomic swap

Zkopru supports the atomic swap in a straightforward way. If A and B want to swap their assets, they create notes for each other and expose the wished note on the transaction data. Then the coordinator should pair the opposite transaction or get slashed.

For example, Alice wants to swap her 50 ETH with Bob's 1000 DAI.

1. Alice spends her 60 ETH note and creates 10 ETH note for herself, and 50 ETH note for Bob.
2. Alice also calculates the hash value of her future 1000 DAI note and expose that hash value to the swap

field of her transaction.

1. Bob also spends his 3000 DAI note and create 2000 DAI for himself and 1000 DAI note for Alice.
2. Bob also calculates the hash value of his future 50 ETH note and expose that hash value to the swap

field of his transaction.

1. Once the coordinator matches the paired set of transactions in the transaction pool, it includes the pair into a new block.
2. If a block inclues only one of them, the coordinator will get slashed.

[

image

958×608 31.8 KB

](https://ethresear.ch/uploads/default/original/2X/0/09dcea1b401e618fa321a78362c7105c48149620.png)

Zkopru is using this simple version of the atomic swap. However if you want to check an MPC based zk atomic swap model, you can check details here.

Merkle tree structure

Note that Zkopru will hava a single UTXO tree and withdrawal tree with 64 depth from the next verison instead of multiple trees with 32 depth.

Zkopru's grove consists of UTXO trees, nullifier tree, and withdrawal trees.

UTXO trees are append-only usage Merkle trees containing UTXOs. Users can spend the UTXOs as the inflow of the transaction by submitting the inclusion Merkle proof. And the output results of transactions are appended back into the latest UTXO tree.

Also, if the zk-transaction creates withdrawal outputs, Zkopru appends them into the latest withdrawal tree. Once the tree's root is marked as finalized, the owner can withdraw assets by proving the ownership.

Afterward, by the commitment-nullifier scheme, the nullifier of the spent UTXOs are marked as used in the nullifier tree, a unique sparse Merkle tree. If a transaction tries to use an already nullified leaf, it becomes invalidated, and the block proposer gets slashed by the challenge system.

[

grove

1031×563 66.1 KB

](https://ethresear.ch/uploads/default/original/2X/c/ca4e3fd77d2d108e0b1e79b8545489123e254a62.png)

Specification of Merkle trees

UTXO Tree

Nullifier Tree

Withdrawal Tree

Type

Sparse Merkle Tree

Sparse Merkle Tree

Sparse Merkle Tree

Depth

31

256

31

Hash

Poseidon

Keccak256

Keccak256

How to update

append-only with subtree rollup with 5 depth tree

SMT rollup

append-only with subtree rollup with 5 depth tree

Cost (gas/leaf)

180k

351k

5.2k

UTXO tree & withdrawal tree will have 64 depth in the Burrito version.https://github.com/zkopru-network/zkopru/issues/35

How to manage the UTXO Trees

A single UTXO tree is a sparse Merkle tree for the membership proof. It uses Poseidon hash, one of the cheapest hash function inside the SNARK, to generate a zk SNARK proof to hide the spending hash and its path.

To append new leaves to the UTXO tree, the coordinator performs the following steps. 1. Prepare an array. 2. Coordinator picks MassDeposits to include, and append every deposit in the MassDeposits into the array. 3. L2 transactions generate new UTXOs. Append newly generated UTXOs to the array. 4. Split the prepared array with the chunk size 32. 5. Construct subtrees and perform the sub-tree rollup.

Suppose the UTXO tree is fully filled with (2^31) items, the system archive the filled tree and start a new tree. The archived trees are also allowed to be used to reference the inclusion proofs of the transactions.

Zkopru optimistically updates the trees' root and verifies when only there exists a challenge. For the challenge, it generates an on-chain fraud-proof using the subtree rollup methodology. Subtree rollup appends a fixed size of subtrees instead of appending items one by one. When if the subtree's depth is 5, it will append 32 items at once. If it contains only 18 items, the remaining 14 items will have remained forever as zeroes. This subtree rollup dramatically reduces the gas cost about 20 times compared to the rollup. To check the source code, please go to contracts/controllers/challenges/RollUpChallenge.sol. And to see how the subtree works please go to packages/contracts/contracts/libraries/Tree.sol

[

subtree-rollup

959×416 27.1 KB

](https://ethresear.ch/uploads/default/original/2X/7/7fd022be927d62031a6aea5f3b6f514ed6b741d6.png)

Nullifier Tree

Every transfer, withdrawal, and migration transaction spends UTXOs with the inclusion proofs and marks the derived nullifiers used on the nullifier tree. Thus, the nullifier tree is a very big sparse Merkle tree that records every spent UTXOs in the 254-depth of Sparse Merkle Tree. Therefore, Zkopru uses keccak256, the cheapest hash function, for the nullifier tree's hash function.

To update the nullifier tree, the coordinator performs the following steps.

1. Pick transactions (transfer, withdrawal, migration) and collect all nullifiers from the transactions.
2. Check there does not exist any already used nullifiers.
3. Mark every nullifier as used. During the update process, if any of the nullifiers does not change the nullifier tree root, discard the transaction because it tries a double-spending.

Just like the UTXO tree, Zkopru optimistically updates the nullifier tree's root. If there is any problem, we can prove a nullifier used more than one by generating a fraud-proof on-chain. To see how it works, please see RollUpChallenge.sol and SMT.sol.

Withdrawal Tree

The only difference with the withdrawal tree and UTXO tree is that the withdrawal tree uses keccak256 for the hash function. The reason why it uses keccak256 is that Zkopru needs the withdrawal tree's Merkle proof on the smart contract while it needs UTXO tree's Merkle proof inside the SNARK circuit. The leaves in the withdrawal trees become withdrawable on the layer-1 smart contract after its tree's root becomes finalized.

To update the withdrawal tree, the coordinator performs the following steps.

1. Collect every withdrawal leaf of the picked transactions.
2. Split the collected withdrawal array with the chunk size 32.
3. Construct subtrees and perform the sub-tree rollup.

Mass deposit

What happens when a user deposits assets to Zkropu.

1. Zkopru contract transfers the given amount of assets from the user's account to itself.
2. It verifies that the note

has a valid hash of the given information.

1. It merges the note to the last item of the MassDeposit[]

list

What is MassDeposit?

MassDeposit is a single mergedLeaves

bytes32 value to be used for rollup proof. Please check what is mergedLeaves

for rollup proof here. If the coordinator proposes a block including MassDeposits, the block appends all notes in the MassDeposit to its UTXO Merkle tree.

How does the coordinator handle MassDeposits?

Coordinators can include only the "committed" MassDeposits that do not change anymore. To include a MassDeposit, the coordinator monitors the Deposit

event from the Zkopru contract.

When does a MassDeposit become "committed"?

It is important to push the deposits to the layer-2 as quickly as possible. Therefore, it freezes the latest MassDeposit when the coordinators propose every new block.

Can the coordinator include more than one MassDeposit?

Yes, it is possible to include several MassDeposits at once in the range of maximum challengeable cost.

Mass migration

The basic idea of mass migration is pretty simple. While deposit

transactions on the layer-1 contract create a MassDeposit

object, “Migration” type outputs of transactions can create a MassMigration

that constructs a MassDeposit

for its destination network.

A transaction can have UTXO, Migration, or Withdrawal type of outputs.

In Zkopru, for the migration, there are source network and destination network. Once a mass migration on the source network is finalized(code here), the [migrateTo

function](https://github.com/zkopru-network/zkopru/blob/034ad7b41eca2a9fc0d344a5b5a8a4525e904c96/packages/contracts/contracts/controllers/Migratable.sol#L15) on the source network can be executed. The function moves the assets, including Ether, ERC20, and ERC721, while creating a MassDeposit object for the destination network.

Therefore, the destination network should implement [acceptMigration

function](https://github.com/zkopru-network/zkopru/blob/034ad7b41eca2a9fc0d344a5b5a8a4525e904c96/packages/contracts/contracts/controllers/Migratable.sol#L52). More info here

The migration standard between rollups is going to be standardized through EIP.

Instant withdrawal

In Zkopru, withdrawers can request an instant withdrawal by setting the instant withdrawal fee for each withdrawal note. Then, anyone can pay in advance for the unfinalized withdrawals and get the fee for them.

To request the instant withdrawal, the owner generates an ECDSA signature for her note and broadcast it. Anyone who has enough assets to pay can pay in advance for the withdrawal using the signature. Once Zkopru includes the transaction successfully, the smart contract transfers the ownership of the withdrawal note to the payer. Finally, the prepayer withdraws them after the finalization.

We can have a decentralized open market for the instant withdrawal fee. To follow up this, please subscribe this github issue, https://github.com/zkopru-network/zkopru/issues/33

Conclusion

With this specification, we’ve successfully built the testnet using Circom, Solidity, Typescript and etc.

GitHub

zkopru-network/zkopru

Ethereum L2 scaling solution for private transactions using zk-SNARK and optimistic rollup. - zkopru-network/zkopru

First of all, we could achieve an affordable gas cost per zk transaction. The average is about 8800 gas, and the theoretical maximum TPS marks 105 when the block gas limit is 11,950,000, and the block time is 13.2 seconds. In Zkopru, the transaction data consumes about 534 bytes. Since the proof data is 256 bytes, we can reduce the transaction cost about twice if we apply the proof aggregation in the future. Otherwise, the storage cost for each the block proposing and finalization, marks about 168k gas and 55k gas. This cost is about 6.7 % of the block generation cost when we include 350 transactions.

Furthermore, we could implement lots of features using Optimistic Rollup’s flexibility. First, Zkopru supports various types of transactions with multiple SNARK verifying keys. You can even make a transaction with 1 input and 4 outputs, or 4 inputs and 1 output. It was very simple to make it support multiple types of transactions by the optimistic rollup’s flexibility. Second, Zkopru implements a pin-point type of challenge case. It means that if the n-th transaction of a block has a problem, the challenge inspects only that specific transaction.

Also, it is very important that Zkopru needs you to run the node on your machine. Therefore, the SNARK efficiency and the light node were the important factors to consider for software implementation. Accordingly, the project is built with Typescript and

NodeJS for future usage in the react-native based mobile application. It is expected that the light node will consume only about 50~100 MB storage for tree management.

To summarize the work, we're expecting that Zkopru can be used for a privacy transaction layer of Ethereum. It is fast, cheap, and also migratable to the upgraded versions. Contributions are welcoming. You can see an organized version through the document page(https://docs.zkopru.network) of Zkopru.

Thank you so much for reading this post.

Related works

1. Ethereum 9 3/4: Optimistic rollup for zk-Mimblewimble
2. BarryWhitehat's zk-rollup
3. John Adler's Minimal Viable Merged Consensus
4. Plasma-group's Optimistic Rollup
5. Batch Deposits for [op/zk] rollup / mixers / MACI
6. Mass migration to prevent user lockin in rollup
7. Z Cash