Abstract:

A zkRollup with extremely low calldata cost, enabling many types of ERC20s/NFTs transfers to many recipients in one transaction. The combination of client-side ZKP and limited online assumption achieves near zero gas cost in return for 3-40 sec latency at the client-side.

The previous research works on introducing online communication to zkRollup succeeded in cutting calldata costs when a transactor and an aggregator agreed to reduce the cost by being online. This work keeps the lowest cost even if communication fails. This upgrade was achieved by separating the proposal world state root to distribute proofs from the approval world state root resulting in the cancellation of transactions in the same circuit and in the same block. This feature means not only efficiency but also the removal of the DDoS attack vector, which stems from intentional communication failures. The client-side ZKP helps compress transactions without imposing additional tasks on aggregators.

Most of the components are already implemented, and the benchmark implies that 0.5 M token transfers per second for one network is a realistic number with the recent ZKP technologies.

Background:

Solutions with online communication to cut calldata costs in zkRollup are well found previously, for example,Adamantium, zkRollup with no tx history and Springrollup. In zkRollup with no tx history data, a Merkle proof of asset which can be used for a fund exit to Layer1 is not generated from calldata of tx history/state-diff by the user side but given by the aggregator(/sequencer/operator) directly, so that there was no calldata use except transactors' address lists. When a user exits, that old Merkle proof gets proven to be still effective by the non-movement proof of state, which is processed by the non-inclusion proof of the transactors' address list. This separates safety and liveness, and the risk of Data Availability Attacks is limited to liveness so that the protocol can guarantee the safety of funds while cutting almost all of the calldata costs. And, Springrollup proved that recoding the last block number that a transactor committed is a much more elegant way to perform non-movement proof.

Regarding these solutions with online communications, there are things left to be modified.

1. zkRollup with no tx history data requires calldata use to store state-diff when the online communication fails.
2. Protective withdrawal and force withdrawal are simple and elegant, but using Layer1 is still expensive if it is the case for being offline and presents difficulties in terms of usability.
3. A sender needs twice the amount of communications with limited nodes in some cases.

And in all works in this category,

1. Batching the txs to many recipients adds linear calculation costs to an aggregator, while these cut the calldata costs
2. Sending many types of ERC20s/NFTs at the same time adds linear calculation costs to an aggregator

This post explains one-time communication with cancelation mechanism fixes (1) ~ (3), and client-side zkp fixes (4)~(5).

Terms

Let us say Sender=Alice=She, Recipient=Bob=He, Aggregator=He.

The word "aggregator" includes "operator", "sequencer", and "validator"; then we use "he" for it since a sender has more communications than a recipient has.

"tx" means transaction.

Overview:

A sender gets proof of the result of a transaction from an aggregator just after she sends it to an aggregator, and the proof is available when she exits her funds to Layer1 using the non-movement proof of assets with her last block number. This communication conveying the proof is restricted by the zkp circuit, which a Layer1 smart contract verifies. The circuit requires the signature on the proof from the sender.

Intmax has these features.

1. When the aggregator fails to get a signature from a sender to put it into the circuit, the state and the root of the sender's assets get restored trustlessly. So the failure of the communication never causes additional calldata consumption.

2. One transaction accommodates unlimited numbers of ERC20s/NFTs to many recipients with the same call data cost/ aggregator side's zkp cost. Instead, the latency of client-side zkp calculation takes longer.

3. After a commitment of a transaction, the token transfer completes with the recipient showing the proof from the contents of the block-header by the sender. (Knowing the contents of state-diff Merkle tree give the recipient the ability to generate the next tx's zkp proof, then the transfer completes)

Any user has their asset storage and its Merkle root. A sender purges the part of the tree of assets, and a recipient gets a part of the partial tree, which is designated for him. He merges it when he transfers these assets. The root transition with such inputs can get verified with the client-side zkp, but the inputs themselves should be verified with an inclusion proof to the block-header in calldata.

[

ERnewpost (9)

960×540 56.2 KB

](https://ethresear.ch/uploads/default/original/2X/2/2698d2b1b38672ba648031b586debd9f126f33cc.jpeg)

Aggregators know all the roots of all users' assets, so they have the Merkle tree consisting of the roots.

In a nutshell, this whole protocol consists of the constraints of the aggregators' circuit about these points:

1. proving the transition of roots of senders and recipients

2. proving all the relevant data to secure the fund exits delivered to the sender and recipients

3. proving the cancel tx and restoring the root when (2) fails

The picture of the whole state tree follows as below, naming the data [1]~[14].

[

ERnewpost (7)

960×540 58 KB

](https://ethresear.ch/uploads/default/original/2X/a/a258c3f2fa6c5e48967dacce2419d0ca97a55b7a.jpeg)

Protocol steps

The explanation of each step follows.

1. A sender purges the state-diff trees [11], which is the set of assets to transfer, from her asset storage tree[10]. Generates a tx with the root of state-diffs(= tx-hash), the root of AssetStorage after the purge, the root of AssetStorage before the purge, and the zkp proving these sets are corresponding. She sends it to the aggregator. She can set a different recipient to each state-diff and make tx-hash[9] with all of these state-diffs.

2. The aggregator makes a block with txs and makes a Merkle tree from all tx-hash from all transactions in the block [7]. He updates the Merkle tree consisting of all the roots of all users [6] by verifying the zkp of each tx data. The root of this tree at this step [2] is the proposal world state in the block. He sends each (Merkle proof of tx-hash/ Merkle proof of asset root) back to each sender.

3. The sender signs these two Merkle proofs and sends these back to the aggregator.

4. The aggregator labels addresses with a 1-bit flag with their signatures. If effective signature, an address gets labeled with a success flag; otherwise, it gets labeled with a failure flag. This flagged address list makes a deterministic Merkle tree with {key: address, value: lastBlockNumber}. If a success flag is on an address, lastBlockNumber of the address gets updated to the current block number; otherwise no update. This process means the recipient can use this lastBlockNumber as a flag of cancellation. Any node can reconstruct the tree by the flagged address list in the calldata on-chain. The root of this tree [13] is also in calldata.

5. The aggregator restores the roots of those who failed to return the signatures to the previous roots. He updates the tree consisting of the roots [8] again, making the root of this tree an approval root [3]

6. The aggregator inputs [1]~[5], [11]~[15] to the zkp circuit to generate a zkp proof for the block. He puts these in calldata too.

7. The recipient gets all data of state-diff and its proof up to the block header from the sender since the sender got the proof of tx-hash up to the block-header from the aggregator. This completes the transfer, and the recipient can make a

zkp proof of the transaction to use this asset if it was not canceled. Withholding this data just harms the sender, so there is no incentive to do so for her.

8. When the recipient uses the given asset, he makes the zkp proof to prove the blockNumber of the state-diff equals the block number in the LastestAccountTree at the block for the recipient. At the same time, the zkp circuit confirms the old AssetRoot and new AssetRoot for the result of the merge and purge, just the same as the first step.

When the block generation stops due to some reasons, like a failure to propagate Asset Root Tree data among aggregators, users can exit their fund with the last root and the last block number. This propagation failure is the only case of a Data Availability attack in this protocol, but funds are safe.

.

[

ERnewpost

960×540 67.5 KB

](https://ethresear.ch/uploads/default/original/2X/2/2783c0b152ea3c2ced0b2cdf63a4c5c02e751064.png)

Detail:

Address/ID

We can chop a public key (32 bytes) to a 64 bits address. Collision is ignorable since there is no way to update the root and the state of the used address without knowing the contents beneath the root. We introduce a random number to insert into the root for the first time as registration. Even if a collision happens at the 1 / 2^64 probability, the latter user needs to generate an address one more time.

We do not use a sequential ID number to avoid the zkp calculation cost of Merkle proof of the ID=>address mapping.

Signature

A public key is just a Poseidon hash of a private key for reducing a lot of calculations of zkp. The signature procedure is as the one below.

zkp(hash(privkey)=pubkey, hash(privkey,message)=signature)

Tx contents

Merge part：

{oldAssetRoot, newAssetRoot, given-tx-hash, zkp(oldAssetRoot, newAssetRoot, given-tx-contents)}

Purge part：

{oldAssetRoot, newAssetRoot, tx-hash, zkp(oldAssetRoot, newAssetRoot, tx-contents)}

To be sure, there is no need to do a merge zkp calculation if it is not the time for the recipient to send given assets to the other. The transfer was already completed when he knew the contents of state-diff, the proof, and inclusion proof proving that it’s not canceled.

Circuits and constraints

~Client-side part~

Merge:

・inclusion proof of state-diff (given-tx-contents) up to a block-header

・inclusion proof of the fact the state-diff is not merged before.

・rightness of transition to newAssetState from oldAssetState

Purge:

・rightness of transition to newAssetState from oldAssetState with purged state-diff

~Aggregator Side~

For each transaction

・Inclusion proof's target in the merge part exists in the AllBlockHashRoot [12]

(AllBlockHashRoot is the root of the tree consisting of all block headers before)

・Corresponding oldAssetState to the current value for the key=sender's address in Asset Root Tree

・Recursive proving to verify the rightness of zkp proof of purge/merge sent by the sender.

The first one can also be done on the client-side to reduce the zkp calculation for the aggregator.

For each block(=100txs~1000txs)

・Rightness of AllBlockHashRoot

・Rightness of ProposalWorldStateRoot and ApprovalWorldStateRoot' root

・LatestAccountTree's root stems from FlaggedAddressList correctly

・Corresponding the address list's flags to the existence of a signature for each transaction

・deposit tree's root stems from the deposit data on Layer1 smart contract (explained later)

・tx-hash tree's root is generated from tx-hashes

・The block-header is correctly generated from the block number, the other roots, and etc (described in the second picture)

Asset Storage

{key: tx-hash, value: {key:contract address: value:{key:index, value: balance} } }

All NFTs are ERC1155. For ERC721, the value will be 0 or 1, and the minter-side logic will restrict this.

Where contract address=0, a user can set a random number as the value. The asset storage will be unpredictable.

Block

~Onchain calldata~

Flagged Address List: 8byte array[100 items]

Asset Root Tree's proposal root: 32byte

Asset Root Tree's approval root: 32byte

Tx-hash tree's root: 32 byte

LastestAccountTree's root: 32byte

Root of the Merkle tree consisting of all block headers before[12]: 32byte

~Off-chain block data which aggregators propagate~

AssetRootTree's state delta, [address, root] array[100 items]

The block interval is 10 seconds, and the recursive zkp aggregates these blocks to commit the last one to layer1 every 600 seconds. The preconsensus mechanism (A pre-consensus mechanism to secure instant finality and long interval in zkRollup) will shorten this period without the cost of zkp verification on Layer1.

Withdraw

A withdrawer sends funds to a burn address and proves the returned state-diff with its Merkle proof on Layer1 smart contract. There is no difference between a withdraw tx and a usual tx in Layer2.

Layer1 smart contract can offer an individual withdrawal and an aggregated withdrawal.

Deposit

This is the mechanism transforming deposits on Layer1 smart contract to state-diff in the tx-hash tree. The merge process of the deposit is just the same as a usual receiving process using state-diffs. All the deposits will be beneath the same tx-hash.

One thing which should be noted is that the recipients(=depositors) can reconstruct the contents beneath the tx-hash, but they can not know the Merkle path up to the block-header if we put the tx-hash as the bottom leaf of the tx-hash tree. Then we put this tx-hash(deposit root) just next to the root of the tx-hash tree as described in the second picture. With this change, the recipient can merge the deposit just like a usual state-diff without communicating with an aggregator.

When the block generation stops for any reason, the last depositors will get refunded.

Exit

The users can exit their funds if the last root of their asset storage is still effective. All proof they have to submit to a Layer1 smart contract are three:

1. LastBlockNumber of the recent LatestAccountTree in the last block.
2. LastBlockNumber corresponds to the block number of that last root.
3. The proof of 2 above is included in the block-header, which has the LastBlockNumber.

At the same time, unmerged assets will exit Layer2 to Layer1 with the same circuit. The users can prove these with zkp verified on Layer1.

On the Layer1 Smart contract, there is a mapping {key: ExittedAddress, flag: True}, which gets updated for each exit transaction.

Expected Q & A

About the detailed differences from the other works, more the edge cases, reasons behind the spec.

hackmd.io

Expected Q&A - HackMD

ZKP Benchmark and Scaling

The benchmark indicates that the bottleneck is not ZKP calculation but calldata consumption which this work has already solved as above.

Benchmarks of zkp calculation are as follows.

CPU: Apple M1 CPU

[

1900×576 54.4 KB

](https://ethresear.ch/uploads/default/original/2X/b/bf797321640e1943550f971f5e3eee6f17dc1baa.png)

This indicates that aggregation of 32 * 100 token transfer takes 120 sec calculation time by one Apple M1 CPU without optimization and parallelization. Parallelization and hardware acceleration can drastically shorten this period.

First and foremost, the zkp calculation of the block generation can be parallelized up to around 20~150 processes. Unlike scaling smart contracts, there is no obstacle to separating senders into many process groups since these processes do not influence each other. Every block can accommodate each proof for each group and combine them into one state root. This makes no more delay in comparison with the single process. We can increase processes as far as the recursive zkp calculation finishes in a period like n times of the block generation period. Also, if one generation process delays or fails, another processing node must generate an empty proof to compensate for it. In conclusion, there is a limit to parallelization because of the limit of recursive zkp and the failure scenario.

Then, we can describe scaling as C * H * B times this benchmark where

C is the client-side optimization

H is the hardware or circuit-level optimization

B is the block generation parallelization

B is 20~150.

H is expected to be up to 1000.

Multithreading makes 2~5 times faster than being single (Mathematics | Free Full-Text | ZPiE: Zero-Knowledge Proofs in Embedded Systems | HTML), GPU makes FFT, which is similar to FRI behind Plonky2, hundreds of times faster (https://hal.archives-ouvertes.fr/hal-01518830/document). FPGA/ASIC is supposed to have an even shorter time to generate the proof, according to its nature of the lowest-level optimization. The vector processor will be in the middle of FPGA and ASIC. These hundred-times-level optimizations obviously can help scale 32 tx in 90 sec to 1000 tx in 90 sec even though the calculation is n* poly-log(n) order.

With the n*poly-log(n) assumption, this implies that 32 * 100 token transfers in 120 sec with Apple M1 in this benchmark can shift to 1000 * 500 token transfers in 1 seconds since we can divide client-side zkp (100 transfers in 0.4 sec to 500 transfers in 8 sec) from aggregator-side scaling (32 tx in 90 sec to 1000 tx in 90 sec).

We need to think about the bottleneck of these accelerations which is out of zkp calculations.

The upper limit of 1000 batches is not because of ZKP computation time but the limit of the ID list in calldata.

For 1000 batches per second, 1 recursive proof = 60 blocks = 480K bytes. It is possible to avoid this by dividing the calldata-post transaction or by waiting for proto-danksharding to extend it to 3000 or more.

Without proto-danksharding, for a basic batch of 500, one token transmission consumes a minimum of 0.016 byte, which is about 1.12 * 10^-8 ether when calculated backward from calldata and the usual gas market, and the other on-chain costs of zkRollup are negligible when divided by the number of transactions per 10 minutes.

For client-side ZKP, the upper limit of 100 tx compression is designed for mobile-side zkp, and if x seconds are consumed on a Laptop, up to 500 is acceptable. Therefore, it is clear that the upper limit of 100 can increase to about 500 by simply dividing the circuit or block type.

In total, at least 0.08M token transfers per second for one network if we put realistic numbers C=5, H=30, and B=20. If we introduce the assumption that hardware acceleration (H=30) makes B=150, the system achieves 0.6 M token transfers per second.

Project and testnet

The repository will be public whenever the Ethereum community requires it.

The other things about the project:

https://hackmd.io/XolAetXrSDuUgpKBZWaaeg

Reference

Adamantium

zkRollup with no tx history

Springrollup

zkRollup original by Vitalik Buterin

Plasma Prime