



VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
STUDY PROGRAM: INFORMATICS

Multi-prover Ethereum Layer 2 Rollups
Daugiaproverčio Ethereum sluoksnis 2 Rollups

Research Work (part III)

By: Mykola Kolombet

VU email: mykola.kolombet@mif.stud.vu.lt

Supervisor: Saulius Grigaitis

Reviewer: Linas Laibinis

Vilnius
2024

Acknowledgments to individuals and/or organizations

The author is thankful for providing useful links by Optimistic and zkSync teams regarding their work on Layer 2 solutions. Special thanks to Alchemy support on discord that were responsive during any time of the day and week.

Summary

The purpose of this paper is to investigate and understand existing solutions for Layer 2 Ethereum approaches as well as their possibility of collaboration.

This paper describes how the Ethereum network operates based on Layer 1 and Layer 2, the main differences from the implementation perspective, transaction flow, existing problems, and Layer 2 solutions.

Most of the practical implementations of rollups were taken from the Optimism project for optimistic rollups and zkSynk, Tornado Cash for the Zero Knowledge rollups.

Several different ways are presented to combine two Layer 2 solutions with possible benefits and drawbacks. Possible combinations are Optimistic and ZK rollups on the same level. ZK rollups as a proof mechanism for Optimistic fraud-proof and a combination of ZK rollups.

As a result, there is a program and implementation ideas for Optimistic and ZK solution, that allows Optimistic rollups to migrate to ZK rollups allowing a test period and avoiding modifications for existing data.

Santrauka

Šio straipsnio tikslas yra ištirti ir suprasti esamus 2 sluoksnio Ethereum metodų sprendimus ir jų bendradarbiavimo galimybę.

Šiame dokumente aprašoma, kaip Ethereum tinklas veikia remiantis 1 ir 2 sluoksniu, pagrindiniai skirtumai iš įgyvendinimo perspektyvos, operacijų srautas, esamos problemos ir 2 lygmens sprendimai.

Dauguma praktinių apibendrinimų buvo paimti iš projekto „Optimism“, skirto optimistiniams apibendrinimams, ir „zkSynk“, „Tornado Cash“, skirto „Zero Knowledge“ apibendrinimams.

Pateikiami keli skirtingi būdai, kaip sujungti du Layer 2 sprendimus su galimais pranašumais ir trūkumais. Galimi deriniai yra „Optimistinis“ ir „ZK“ paketai tame pačiame lygyje. ZK apvyniojimai kaip įrodomasis mechanizmas, užtikrinantis atsparumą sukčiavimui, ir ZK rinkinių derinys.

Dėl to yra „Optimistic“ ir „ZK“ sprendimo programa ir įgyvendinimo idėjos, leidžiančios „Optimistic“ rinkinius perkelti į ZK paketus, leidžiančius atlikti bandymo laikotarpį ir išvengti esamų duomenų pakeitimų.

Table of contents

Acknowledgments to individuals and/or organizations.....	
Summary.....	
Santrauka.....	
Table of contents.....	
Introduction.....	
1. Optimistic rollups.....	1
1.1. Transactions.....	1
1.1.1. Block.....	2
1.1.2. Channel.....	3
1.1.3. Batcher.....	3
1.1.4. Batcher transaction.....	4
1.1.5. Batcher Transaction Format.....	4
1.1.6. Sequencer Batch.....	4
1.1.7. Channel Frame.....	4
1.1.8. Frame Format.....	5
1.1.9. Channel Format.....	5
1.1.10. Batch Format.....	6
1.2. Sequencing & Batch Submission Overview.....	7
1.3. Transaction proving.....	9
1.4. Fraud proof.....	10
1.5. Validators.....	10
1.6. Data storage.....	11
1.7. Markle Tree.....	12
1.7. Data required for optimistic rollups operations.....	13
1.8. Data availability.....	14
2. Zero-Knowledge rollups.....	15
2.1. Data availability.....	15
2.2. Transaction finality.....	15
2.3 Transactions.....	16
2.4. State commitments.....	16
2.5. Validity proofs.....	17
2.6. Proof generation.....	17
2.7. Proof verification.....	18
3. Sharding and EIP-4844.....	20
4. Possible solutions.....	21
4.1. Optimistic and Optimistic.....	21

4.2. Optimism and ZK-rollups.....	22
4.3. Zero-Knowledge and Zero-Knowledge.....	22
5. Existing Solutions.....	23
5.1. RiskZero.....	23
5.2. OP Stack Zero Knowledge Proof.....	23
6. Combination decision.....	24
7. Program implementation.....	25
7.1. Tornado Cash.....	25
7.2. Optimistic rollups.....	26
7.3. Data access.....	26
7.4. Alchemy.....	26
7.5. Optimism Etherscan.....	28
7.6. Choice of data provider.....	28
7.7. ZK solutions.....	28
7.8. Smart Contract.....	29
7.9. Remix.....	30
7.10. Metamask.....	30
7.11. The program.....	31
7.11.1 Sequence diagram.....	31
7.11.2 Functional view.....	35
7.11.3 Proff generation.....	36
7.11.4 Smart contract.....	37
8. Experiment Conducted.....	37
8.1. Experiment hardware.....	38
8.2 Experiments results.....	38
9. Results and Conclusions.....	40
Definitions of the terms.....
Abbreviations.....
Resources.....
Appendix.....

Introduction

Research of the multip-proving system using ZK-rollups and optimistic rollups can provide substantial improvements in the scalability of decentralized systems while preserving security. While both approaches are currently under construction, it is possible to have bugs and vulnerabilities that are not yet discovered. However, it is highly unlikely that both of the systems will have the same problem, so we can expect that transactions are valid with a higher probability if they are valid for both ZK-rollups and Optimistic Rollups.

With the known trilemma[Mus23] of Ethereum Mainnet that it is decentralized, secure, and scalable, the L1 can not be more scalable without losing security or decentralization. Therefore, L2 was introduced which allows to process way more transactions without any changes to L1. Currently, rollups are the preferred solution for scalability which allows for decreasing the gas fees by up to 100x[Cry18].

Rollups combine multiple transactions into one bundler. Then transactions are executed on the L2 and posted to L1. The gas fee is spread among transaction creators making it cheaper for each user. Here are two different approaches to rollups: optimistic and Zero Knowledge they differ primarily on how this transaction data is posted to L1.

Both of the technologies come into place with the rapid growth of the Ethereum network. The network has already reached capacity with 1+ million transactions per day[EthA]. Which leads to the higher gas price[EthB] and pushes out users who cannot afford those fees. Another problem that was in place is that Ethereum Mainnet is only able to process roughly 15 transactions per second[Mis].

Since the work on L2 scalability was started in 2021 and a lot of projects still have additional trust assumptions as they work to decentralize their networks. Hence, it is not possible to make sure that any particular project is secure and bug-proof.

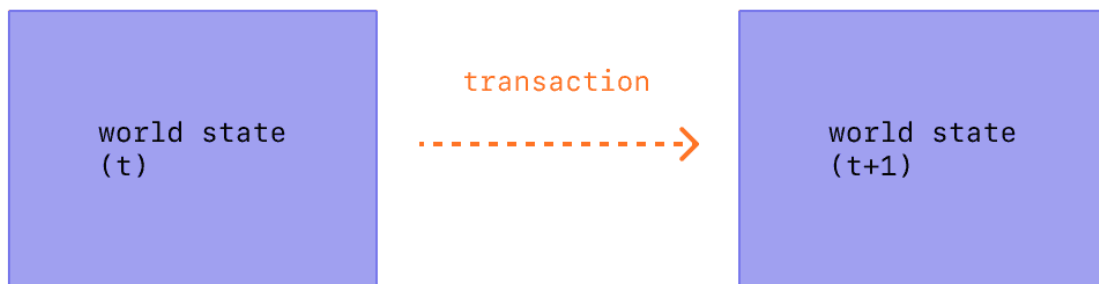
In this work, we are going to describe the current state of the L1 and L2, and how they work together to identify problems, check currently available solutions, and search for the way to make them better and more suitable. Also, possible solutions will be shown with description on how they are going to work.

1. Optimistic rollups

The main idea of optimistic rollups is to bundle up transactions, compress data and store it on Ethereum's main net. No proof of validity is required, this is why rollups are called optimistic. The security part relies on a fraud-proving scheme to detect cases where transactions are not calculated correctly.

1.1. Transactions

Transaction can be imagen as a connector between two states(Picture 1.):



Picture 1: *State changes*

[DJQ22]When participant 1 wants to transfer Ethereum to participant 2 he creates a transaction. The transaction has enough data to change the state of the system (e.g. deduct Ethereum from wallet 1 and add it to wallet 2). Transaction data consists of these parts:

- From – the address of the sender, that will be signing the transaction. This will be an externally-owned account as contract accounts cannot send transactions.
- Recipient – the receiving address (if an externally-owned account, the transaction will transfer value. If a contract account, the transaction will execute the contract code)

- Signature – the identifier of the sender. This is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction
- Nonce - a sequentially incrementing counter which indicates the transaction number from the account
- Value – the amount of ETH to transfer from sender to recipient (denominated in WEI, where 1ETH equals 1e+18wei)
- Data – optional field to include arbitrary data
- GasLimit – the maximum amount of gas units that can be consumed by the transaction. The EVM specifies the units of gas required by each computational step
- MaxPriorityFeePerGas - the maximum price of the consumed gas to be included as a tip to the validator
- MaxFeePerGas - the maximum fee per unit of gas willing to be paid for the transaction (inclusive of baseFeePerGas and maxPriorityFeePerGas)

3 types of transactions are currently in use on Ethereum:

1. User to user. The funds are transferred 'from' From to Recipient
2. Contract deployment transaction. Where the Recipient address is empty and the data block has a smart contract code.
3. Contract execution transaction. Where the Recipient address is an address of the smart contract

1.1.1. Block

All the transactions that are submitted to the sequencer are united into the block. The block itself has a sequence of transactions and adds some extra data in the header[Eik19]:

- parentHash - hash of the parent block.
- uncleHash - hash of the uncle block.
- coinbase - address that will receive the block reward.
- stateRoot - the Merkle root of the state tree.
- transactionsRoot - the Merkle root of the transactions list of the block.

- receiptsRoot - the Merkle root of the receipts list of the block.
- logsBloom - the Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.
- difficulty - the difficulty level of the block, calculated from the previous block's difficulty level, and the timestamp.
- number - the number of ancestor blocks (aka "height"). The genesis block has the number 0.
- gasLimit - current maximum amount of gas usable per block.
- gasUsed - the amount of gas used in this block.
- timestamp - a value equal to the reasonable output of Unix's time() at this block's inception.
- extraData - an arbitrary byte array containing data relevant to this block.
- mixHash - a hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block.
- nonce - a 64-bit scalar value which, combined with the mixHash, proves that a sufficient amount of computation has been carried out on this block.
- hash - hash of the block.

1.1.2. Channel

A channel is a sequence of sequencer batches (for sequential blocks) compressed together.

It can be split in frames in order to be transmitted via batcher transactions. The reason to split a channel into frames is that a channel might be too large to include in a single batcher transaction.

1.1.3. Batcher

A batcher is a software component (independent program) that is responsible to make channels available on a data availability provider. The batcher communicates with the rollup

node in order to retrieve the channels. The channels are then made available using batcher transactions.

1.1.4. Batcher transaction

A batcher transaction is a transaction submitted by a batcher to a data availability provider, in order to make channels available. These transactions carry one or more full frames, which may belong to different channels. A channel's frame may be split between multiple batcher transactions.

When submitted to Ethereum calldata, the batcher transaction's receiver must be the sequencer inbox address. The transaction must also be signed by a recognized batch submitter account.

1.1.5. Batcher Transaction Format

Batcher transactions are encoded as `version_byte ++ rollup_payload` (where `++` denotes concatenation). Where `version_byte` indicates the version of the batcher transaction and `rollup_payload` is a concatenation of several frames.

Unknown versions of the batcher transaction are ignored by the node. If any of the frames inside the `rollup_payload` fail to parse, then the whole transaction is rejected.

1.1.6. Sequencer Batch

A sequencer batch is a list of L2 transactions (that were submitted to a sequencer) tagged with an epoch number and an L2 block timestamp (which can trivially be converted to a block number, given our block time is constant).

Each batch represents the inputs needed to build one L2 block (given the existing L2 chain state) — except for the first block of each epoch, which also needs information about deposits.

1.1.7. Channel Frame

A channel frame is a chunk of data belonging to a channel. Batchers transactions carry one or multiple frames. The reason to split a channel into frames is that a channel might too large to include in a single batcher transaction.

1.1.8. Frame Format

The frame encoded as:

```
frame = channel_id ++ frame_number ++ frame_data_length ++ frame_data ++ is_last
```

Where:

channel_id = bytes16

frame_number = uint16

frame_data_length = uint32

frame_data = bytes

is_last = bool

All of the fields are fixed size except for frame_data.

- channel_id is an opaque identifier for the channel. It should not be reused and is suggested to be random; however, outside of timeout rules, it is not checked for validity
- frame_number identifies the index of the frame within the channel
- frame_data_length is the length of frame_data in bytes. It is capped to 1,000,000 bytes.
- frame_data is a sequence of bytes belonging to the channel, logically after the bytes from the previous frames
- is_last is a single byte with a value of 1 if the frame is the last in the channel, 0 if there are frames in the channel. Any other value makes the frame invalid (it must be ignored by the rollup node).

1.1.9. Channel Format

Channel is encoded in the way, defined as:

```
rlp_batches = []
for batch in batches:
    rlp_batches.append(batch)
channel_encoding = compress(rlp_batches)
```

where:

- batches is the input, a sequence of batches byte-encoded
- rlp_batches is the concatenation of the RLP-encoded batches
- compress is a function performing compression, using the ZLIB algorithm (as specified in RFC-1950) with no dictionary
- channel_encoding is the compressed version of rlp_batches

There is a limit for decompressed data. Currently, it is 10,000,000 bytes. The goal of the limit is to prevent zip bomb attacks, where the compressed data is small, but the decompressed version is abnormally huge compared to the compressed version.

1.1.10. Batch Format

The batch contains a list of transactions that will be included in a specific L2 block.

The difference between an L2 block and a batch is subtle but important: the block includes an L2 state root, whereas the batch only commits to transactions at a given L2 timestamp (equivalently: L2 block number). A block also includes a reference to the previous block.

A batch is encoded as batch_version ++ content, where content depends on the batch_version.

The content is looks like that:

rlp_encode([parent_hash, epoch_number, epoch_hash, timestamp, transaction_list]). Where:

- batch_version is a single byte, prefixed before the RLP contents, alike to transaction typing.
- rlp_encode is a function that encodes a batch according to the RLP format, and [x, y, z] denotes a list containing items x, y and z
- parent_hash is the block hash of the previous L2 block

- epoch_number and epoch_hash are the number and hash of the L1 block corresponding to the sequencing epoch of the L2 block
- timestamp is the timestamp of the L2 block
- transaction_list is an RLP-encoded list of EIP-2718 encoded transactions.

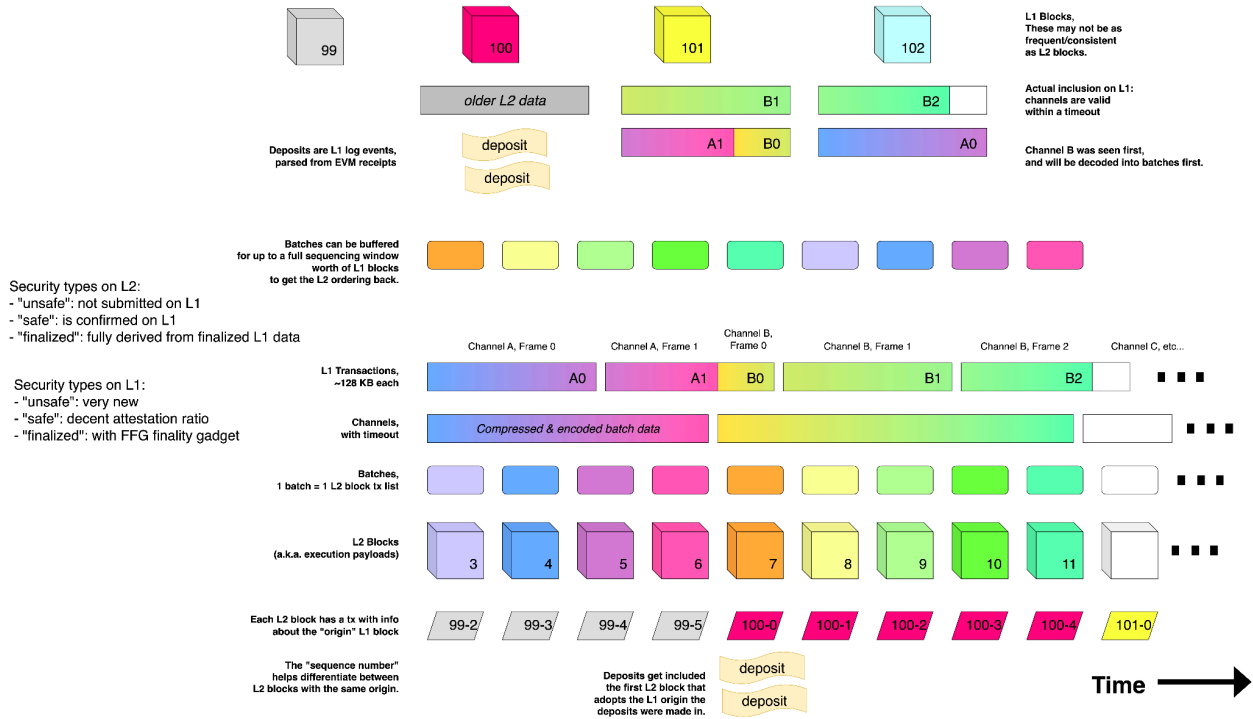
1.2. Sequencing & Batch Submission Overview

The sequencer except transactions from users on L2. These transactions are united inside the block. Then for each block, it creates the sequencer batch. Sequencer is also responsible for submitting each batch to the data availability layer, which is done using batcher.

The batcher submits batcher transactions to a data availability provider. These transactions contain one or multiple channel frames, which are chunks of data belonging to a channel.

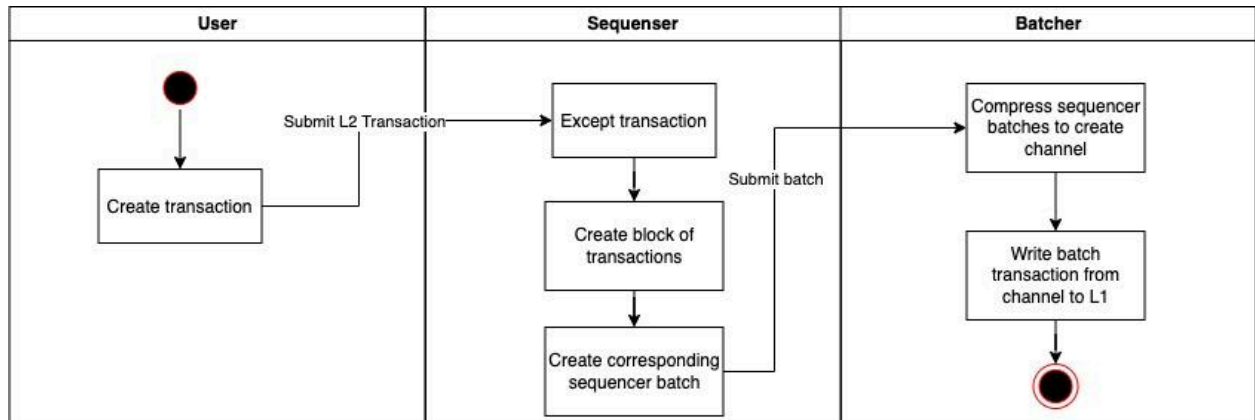
Channels might be too large to fit in a single batcher transaction, hence we need to split it into chunks known as channel frames. A single batcher transaction can also carry multiple frames (belonging to the same or to different channels).

Let's take a look at the image representation(Picture 2):



Picture 2: *Transaction Diagram* [Opt22]

1. The first line represents L1 blocks with their numbers
2. The second line represents batch transactions included inside the block. A1, B0, A0, B1, and B2 - are frames inside the batch. The squiggles under the L1 blocks represent deposits
3. The rounded boxes on the third line represent individual sequencer batches that were extracted from the channels
4. On the fourth line, there are batcher transactions.
5. On the fifth line, there is a reconstructed channel in the right order.
6. The sixth line shows the batches extracted from the channel. The channel is ordered so as batches.
7. The seventh line shows the L2 block which was constructed based on the batch.
8. The eighth line shows the L1 transactions that match the epochs of the L2 blocks and keep attributes for the L2 block.
9. Finally, the ninth line shows user-deposited transactions derived from the deposit contract event mentioned earlier.

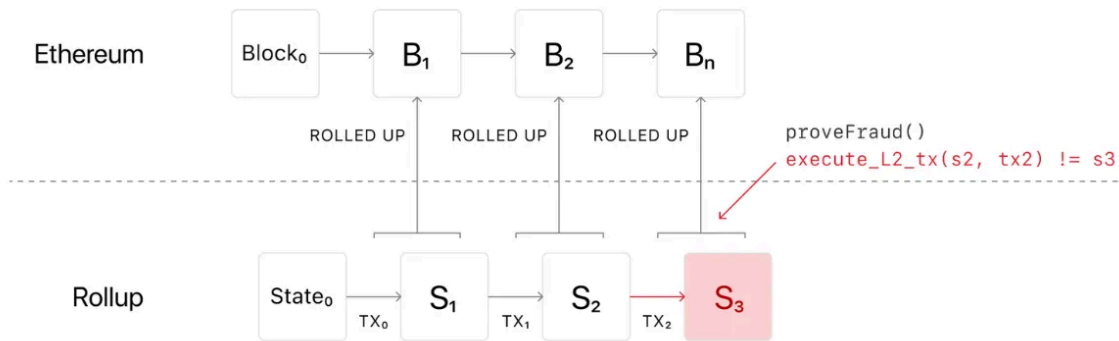


Picture 3: Transaction flow diagram

1.3. Transaction proving

There are 2 requirements to commit a rollup of transactions. The first one is very strict and relies on L2 computations. It is updating the state root of transactions. All of the transactions stored on L1 which are coming from L2 are organized as a Merkle tree. New transactions are going to the root of transactions expanding the tree and creating new dependencies. When the new block of transactions is proposed to be stored there must be provided the current root for the tree and the new root of the tree will represent the new transactions tree after they are executed and stored on L1. At this point, the contract should verify that the current state root matches the old state root listed by the sequencer, and if everything is correct the new state root will replace the old one and will be stored on L1. After that, the block of transactions is committed on the L1. Here is where the less strict validation takes place, transaction verification. The sequencer does not provide validity proof for any of the transactions, the old transactions are considered valid as soon as their tree root is replaced by the new one. From so on it is time for the verifiers to play their role. All of the new transactions are considered valid until someone can prove that any of the transactions are not. This is why this requirement is less strict than tree root one.

At this point in time, we know the old state root and the new state root. Anyone has access to all the data so the validator can take the old state root and perform transactions from the new state root and compare the outcome. If the verifier discovers that after processing new transactions the state root that he got does not match the state root proposed by the sequencer he can trigger a fraud-proof computation.



Picture 4: Transaction proving

1.4. Fraud proof

There are two approaches for fraud-proof, non-interactive and interactive. Above we described some of the steps which lead to the fraud-proof mechanism. The non-interactive one is not used for optimistic rollups because it requires a lot of computation power which leads to higher gas prices. Also, there are a lot more limitations from the EVM(Ethereum virtual machine) for this type of proof, which requires a lot of developers' work. The interactive one is preferred because it requires validation of only transactions which look incorrect and do not need a huge amount of work. This is how it works: The validator and sequencer have disagreed on the new state root, but it does not mean that they have disagreed on all of the transactions. So instead of running all of them, they break all of the transactions into two sets, they agree on the set which should have invalid transactions and repeat the process. In the logarithmic time base on the number of transactions, both will agree on the single transactions that create the problem. This leads to the single step which is required to validate the whole state root. This single step is basically the verification of the transaction on the main net. After the transaction is executed on the L1 it is clear if the new state root can be excepted or should be reverted.

1.5. Validators

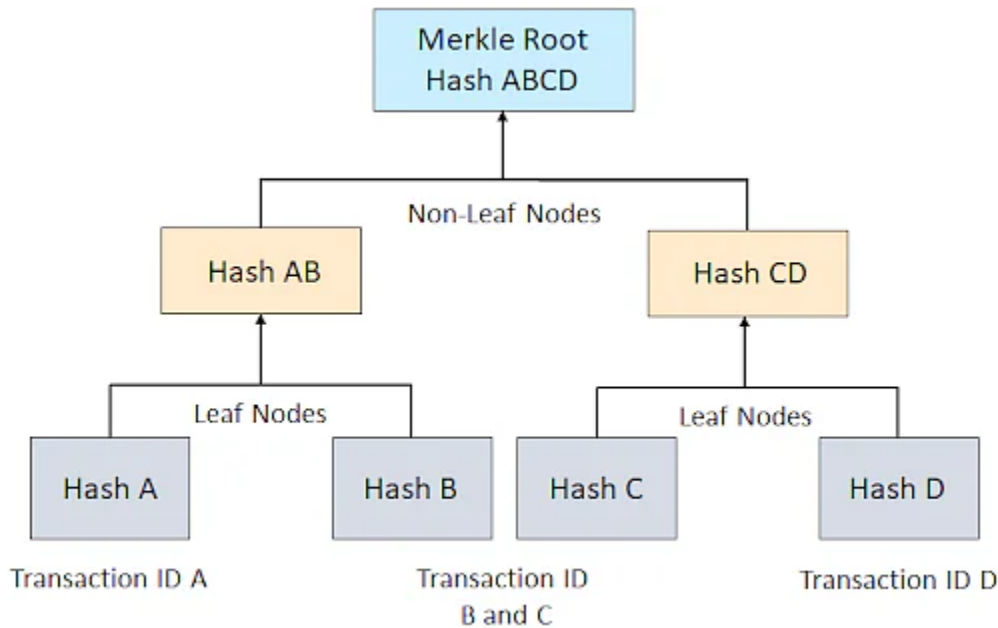
Validators are people who are validating transactions. By the idea of optimistic rollups, anyone can become a validator. However, not all of the projects allow anyone to do so. To be able to participate in transaction validation on L2 the user is required to run a node on the pc. Which can require some additional technical knowledge. Another requirement that is commonly presented by the L2 developers is staking coins. It is required to avoid creating a massive amount of false fraud proofs.

1.6. Data storage

One of the important questions for the optimistic rollups is “how the data is stored?”. Why do we need L1 if all the transactions are executed out of the L1? The answer is simply that it is safe to store data on L1 and it is cheap to execute transactions on L2. We have described what data is stored on the main net. The L2 does not touch any state of the L1, all the transactions there exist separately from the transactions from L2. The crucial part of Ethereum which is used by rollups is calldata. In simple words, the calldata is a place where users can store the variables which will be passed to the contract. Calldata is a non-modifiable, non-persistent area in a smart contract that behaves mostly like memory. The calldata persist on on-chain as part of the blockchain's history logs and do not interact with Ethereum states so it is cheap and safe to use.

The new state root and all of the transactions since the last checkpoint are passed to calldata and the contract is waiting for execution, it is called the dispute period. After that period the contract is executed and a new state root is stored on the main net.

1.7. Markle Tree



Picture 5: *Marklee Tree*

[Kas19]The Markle tree(Picture 5) represents how the block of transactions is organized to be used by the blockchain. Each leaf represents a single transaction. The creation of the tree starts from the bottom and goes all the way up to the root. The algorithm for creation of binary Marklee tree looks like that:

1. Hash each transaction.
2. Take all the transactions and split them into groups of 2 transactions. If there is an odd number of transactions duplicate the last hash for one transaction
3. Hash together hashes of groups for 2 transaction hashes.
4. Group hashes into groups of 2.
5. Hash hashes of the group.
6. Repeat 4-5 until there is 1 hash left.
7. The hash left is the tree root.

The above example represents the Marklee Tree. There are 4 transactions: A, B, C, D. These transactions were hashed and now have Hash A, Hash B, Hash C, and Hash D. In the next step the hashes are hashed together in order to get the parent hashes Hash AB and Hash CD. Then parent hashes are hashed together into Hash ABCD which is the tree root.

The tree root hash is stored on the L1 which allows validators to validate the batch of transactions.

The data is available for some period of time. The optimis states that data should be available for the validators and sequencers to perform operations. However, data availability doesn't guarantee that the data will be there for the long term.

Currently, the only supported data availability provider is Ethereum call data. Ethereum data blobs will be supported when they get deployed on Ethereum.

1.7. Data required for optimistic rollups operations

1. Transaction data. All the transactions are stored on the L2. Each transaction is added to the Markle tree where the leaf is the actual transaction and the root represents the hash for all of the transactions. When the number of transactions reaches some limit the sequencer can verify them, compress and submit one transaction to the L1 which will represent the new state of the L2 chain.
2. State data. The rollup chain maintains the current state of the rollup with the current balances of the users. There are two source of information that is provided to the users. One is the L2 chain which has the latest transactions but may have some transactions which will not be included in the next state update. The second one is the L1 state, which has all the transactions which are already committed to the network and the list of transactions that are planned to become part of the transactions but not yet executed due to the dispute period.
3. Markle tree data. This is the tree where all the transactions are stored in the compressed form on the L2. The root of the Markle tree is stored on the L1 with all the necessary data to recreate the whole tree which allows validators to validate the block of transactions.
4. Validator data. This data represents all the validators that are registered to the network, their voting power and the reward allocation.
5. Dispute resolution data. Since fraud-proof requires the possibility to dispute the transactions this data is needed to store all of the disputes on the network, their outcomes, and their current state.

6. Chain metadata. Chain metadata includes information about block height, and block timestamp.

1.8. Data availability

All of the data is available on both Layer 1 and Layer 2. Optimistic rollups uses Layer 2 network to process transactions provided by sequencer, this network can be accessed using Etherscan or Alchemy API. The same works for Layer 1 transactions, the correct network can be selected on both data providers.

2. Zero-Knowledge rollups

There is another interesting approach for the roll-ups: ZK-rollups. They work almost the same as optimistic rollups, but instead of having a fraud-proof window and storing all transactions on the L1, ZK-rollups store the new state of the Markle tree and some cryptographic proof that those changes are correct.

2.1. Data availability

ZK-rollups publish the state data to the main chain. The same as for optimistic rollups the calldata on the smart contract is used for this purpose. However, instead of publishing the transactions themselves, the state for every transaction and validity proof is published. This data is enough to validate the whole chain state.

All of the transactions are stored on L2 and L1 just keeping the state changes and transaction ids which allows user to verify their account balance. The way to retrieve the transaction depends on the specific implementation of the ZK-rollup, but it can be expected that there will be a contract on L1 that allows retrieving all of the transaction data based on the hash. Also, there are several applications that will do the work or the user can try to link L1 proof and L2 transaction by Id.

2.2. Transaction finality

Again, the same as for optimistic rollups, the final state of transactions is when everything is approved on L1. When it comes to ZK-rollups, submitted state and validity proof are validated by the contract on the L1 and if everything is correct they both are stored on the L1. At this point, the transaction is considered done or finalized, which means that there is no way to revert or somehow impact this transaction.

2.3 Transactions

Transactions(Picture 6.) allow users to transfer funds from one account to another. There are two ways that can be used by the user to transfer funds with transactions. First, is to submit the L2 transaction to the sequencer, he will create the batch of transactions, L2 block, and publish data to the smart contract on L1. The second way is used when the sequencer ignores the user for some reason, the user can submit a transaction to the smart contract on L1 directly, this will be more expensive than using a sequencer, but it is guaranteed that the transaction will be processed.

```
{
  blockHash: '0x412f43399e64f03f4eb4eff071a7970ced83763ad97ec8096078d49f6f372da3',
  blockNumber: '0x5adc90',
  hash: '0x4963d1bd8bdcd4e5b934017e6f206cb0270cdc9f7097dafc87b859619f866519',
  transactionIndex: '0x0',
  type: '0x0',
  nonce: '0xd7793',
  input: '0x',
  r: '0x4baad25f6be9c5ae8b1b0b6edc714b04c53b3ba239d879ab893d448448c85bab',
  s: '0x7cc4b77255fce8aea4bca2cdbdd9acbceb48d6d39d3b46d109871330bd8d7709',
  chainId: '0xaa36a7',
  v: '0x1546d72',
  gas: '0x14820',
  from: '0xa7e4ef0a9e15bdef215e2ed87ae050f974ecd60b',
  to: '0xdc87165102f13bcecb76e8f7b443fdd41d642817',
  value: '0x16345785d8a0000',
  gasPrice: '0xc4bfd37c4'
}
```

Picture 6: *Transaction example*

2.4. State commitments

The ZK-rollup's state, which includes L2 accounts and balances, is represented as a Merkle tree. A cryptographic hash of the Merkle tree's root (Merkle root) is stored in the

on-chain contract, allowing the roll-up protocol to track changes in the state of the ZK-rollup.[Eth22]

The roll-up transitions to a new state after the execution of a new set of transactions. The initiator of the transition must provide the new state root and submit it to the L1 contract with the validity proof. If the validity proof is correct, the new Marklee root becomes the canonical ZK-rollup state root.

2.5. Validity proofs

The transaction process is relatively simple. We can describe it with the following example:

Bob wants to transfer 10 ETH to Allise. He creates the transaction and submits it to the sequencer. The sequencer decreases the Bob balance by 10 ETH and increases the Allise balance by 10 ETH. Then sequencer needs to compute the new state root of the Marklee tree, so he hashes all the new account data, rebuilds the Maeklee tree, gets the root, and submits it to the L1.

However, the contract will not accept this data without one small detail - validity proof. Which is a succinct cryptographic commitment verifying the correctness of batched transactions.

Validity proofs allow participants to prove the correctness of the statement without revealing the statement. ZK-rollups use validity proof to confirm off-chain state transitions without re-executing all of the transactions on the Ethereum main net.

These proofs can come in the form of a ZK-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) or ZK-STAR (Zero-Knowledge Scalable Transparent Argument of Knowledge).

2.6. Proof generation

After the transaction is submitted to the operator he will perform the usual checks, to confirm that:

- The sender and receiver accounts are part of the state tree.
- The sender has enough funds to process the transaction.

- The transaction is correct and matches the sender's public key on the rollup.
- The sender's nonce is correct, etc.

Once the ZK-rollup node has enough transactions, it aggregates them into a batch and compiles inputs for the proving circuit to compile into a succinct ZK-proof. This includes[BLN23]:

- A Merkle tree comprising all the transactions in the batch.
- Merkle proofs for transactions to prove inclusion in the batch.
- Merkle proofs for each sender-receiver pair in transactions to prove those accounts are part of the rollup's state tree.
- A set of intermediate state roots, derived from updating the state root after applying state updates for each transaction (i.e., decreasing sender accounts and increasing receiver accounts).

Then proving circuit computes the validity proof by "looping" over each transaction and performing the same checks the operator completed before processing the transaction. It verifies that the sender account is part of the Marklee tree, checks if it has sufficient balance, reduce the balance, updates nonse, recalculates the hash, and combines it with the Marklee tree root to calculate the new one. Then all the operations are repeated for the sender's account.

When all the transactions inside the batch are processed, the last computed Marklee tree becomes the new canonical state root.

2.7. Proof verification

When the proving circuits verified the correctness of update L2 operator submit the batch of transactions to the L1 smart contract. The contract verifies proof validity and check the public data that was used to generate the proof, such as[LZH24]:

- Pre-state root: The ZK-rollup's old state root (i.e., before the batched transactions were executed), reflecting the L2 chain's last known valid state.
- Post-state root: The ZK-rollup's new state root (i.e., after the execution of batched transactions), reflecting the L2 chain's newest state. The post-state root is the final root derived after applying state updates in the proving circuit.
- Batch root: The Merkle root of the batch, derived by merklizing transactions in the batch and hashing the tree's root.

- Transaction inputs: Data associated with the transactions executed as part of the submitted batch.

If the proof is valid, it means that exists a sequence of transactions that transitions the pre-root to the new root. Also, if the pre-state root matches the root stored on the contract, the contract root got updated, and all of the new transactions are finalized.

In order to generate the proof transaction data is used to generate the circuit that later will be used to generate execution trace. The transaction data may include sender/receiver accounts, and amounts. All of this depends on the specific implementation.

Then the execution trace is used to generate a constraint system that constrains the values of the inputs and outputs to the values in the execution trace. This constraint system can be represented as a set of polynomial equations.

Then polynomial commitment scheme is used to commit to the polynomials that represent the constraint system.

3. Sharding and EIP-4844

Sharding is a technique used to increase the efficiency and scalability of a blockchain network. It involves dividing the network into smaller units, or "shards," allowing each shard to process transactions in parallel. This allows the network to process more transactions per second and reduces the strain on individual nodes, making the network faster and more efficient.

One of the ways to make transactions cheaper was proposed with eip-4844[Loe22]. Which is one of the steps to implement sharding inside the Ethereum network. The idea is to create a new type of transaction that will have some additional data storage called a blob.

Blobs are 4096 field elements of 32 bytes each, with a long-term maximum of 16 blobs per block. $4096 * 32 \text{ bytes} * 16 \text{ per block} = 2 \text{ MiB per block maximum}$.

Blobs are pruned after ~2 weeks. Available long enough for all actors of an L2 to retrieve it, short enough to keep disk use manageable. This allows blobs to be priced cheaper than calldata which is stored in history forever.

For the optimistic rollups blobs will be used as data storage instead of calldata. The list of transactions for the optimistic rollups needs to be stored for a short period of time, which is a fraud-proof window. After that, there is no need to keep all transactions.

The call data is not needed because all the data is available on L2, and L1 is just used for the state transition and verification. For the optimistic rollups, L1 is needed to store the current state of the blockchain and transactions that are planned to become part of the chain. Since the time that is required for the transactions to be verified is less than the time that blob can leave, there is no need to use callData.







Similar situation with ZK-rollups. The L1 is used to keep the state of the system and to verify that proofs are legit. The amount of time needed for verification is smaller than the time of blob life, so the new blobs will appear fast enough not to lose any data.

4. Possible solutions

With all pros and cons of Optimistic Rollups and ZK-rollups, we can have several possible solutions, but all of them will have some common questions. Do we want to combine completely different technologies or do we want to stick to one particular?

4.1. Optimistic and Optimistic

Optimistic rollups have brought us better scaling and simple architecture that is clear for all users and bug-proof for developers. It doesn't mean that it is perfect since they still have problems(Picture 7.) but it is something that we are already using.

NAME	STATE VALIDATION ⓘ	DATA AVAILABILITY ⓘ	UPGRADEABILITY ⓘ	SEQUENCER FAILURE ⓘ	PROPOSER FAILURE ⓘ
 Arbitrum One 🛡️	Fraud proofs (INT)	On chain	~12d 9h or no delay	Self sequence	Self propose
 Optimism 🛡️	In development	On chain	Yes	Self sequence	Cannot withdraw
 zkSync Era 🛡️	ZK proofs	On chain (SD)	Yes	Enqueue via L1	Cannot withdraw
 dYdX	ZK proofs (ST)	On chain	9d or 2d delay	Force via L1	Use escape hatch
 Metis Andromeda 🛡️	In development	Optimistic (MEMO)	Yes	Enqueue via L1	Cannot withdraw
 Loopring	ZK proofs (SN)	On chain	Yes	Force via L1	Use escape hatch

Picture 7: Layer 2 problems [L2b23]

If we take a look into common problems of Optimistic solutions, we can see that some systems allow only whitelisted participants to be part of fraud-proof, and some of the systems can be blocked by sequencer shutdown, or the system permits invalid state roots. Some of these problems can be addressed by combining two systems together. However, It would not bring any benefits to the approaches. From the perspective of ideas and possible problems, the combination would result in a performance which no higher than the performance of the worst application in pair. In other words, if solution A permits invalid state roots and solution B does not, there always will fail when see one. Even though this fact should increase the stability and security of solution A the average stability will be the same.

4.2. Optimism and ZK-rollups

This combination can have more potential than optimism + optimism. Both are completely different technologies with different problems. Based on Picture 7 the ZK-rollups are more secure, but they are not bug-free. There is no way so far to prove that a particular implementation of ZK-rollups is always working correctly. This is why the combination of optimism and zk-rollup might be helpful. Here the only benefit is securing the ZK network by submitting the same transactions to the optimistic network. However, there is not much difference in performance, like for the optimism + optimism solution. Withdrawals will be slow, transactions will be slow. All of this is because of optimism. On the other hand, if we can have the same Merkle tree for both Zk-rollup and Optimism, the fraud-proof period can be reduced. Another way to use this combination is to have a sort of production testing for ZK-rollups. Yes, there will be slower than having just ZK-rollup, but if there is something wrong with the ZK-proof mechanism, it will be possible to detect that with interactive fraud-proof of Optimistic rollups.

4.3. Zero-Knowledge and Zero-Knowledge

Combination of two Zero-Knowledge rollups. Probably one of the most interesting combinations out of the three. Two zk-rollups virtual machines can work together to prove each other fraud proofs, process the same transactions and combine result verification to verify the block. Most ZK solutions don't have obvious bugs and it is hard to detect if they have any problems at all, so combining two such systems can help identify the problems that are not common for both. Having two different approaches for creating ZK-proof should reduce the chance that some fraud actions are passed undetected, keeping the speed of transaction processing in a high level.

5. Existing Solutions

There are several existing solutions that trying to get the best from both approaches, below we will list two of them. The first one is a separate project, the second one is a possible way of evolution for the Optimism project.

5.1. RiskZero

One of the existing projects in this area is RiskZero. According to the article[RL23], this project is a zk-rollup, however, it can be used as an on-top layer of an existing optimistic rollup solution. The fraud-proof algorithm for optimistic rollups can require replaying all transactions on L1 which is expensive. RiskZero offers a solution where the transactions can be replayed on their EVM and the result will be backed up by zk-proof. After that, only zk-proof will be validated on L1.

5.2. OP Stack Zero Knowledge Proof

Meanwhile, optimism inviting developers to work on their version of ZK-rollups[Opt23]. The goal, as always, is to guarantee that the state transition from one block to another, that is stored on L1 is correct. At the moment, there is no clear vision of how it is going to work since the project is at its first stage and the company is waiting for developers to make proposals.

6. Combination decision

As discussed in Section 4 there are several possible ways to implement Multi-prover solution. We have considered all of them and made a final decision to use Optimism and ZK together.

An Optimistic rollups combination doesn't make any difference. Several optimistic solutions are already available to the customers and supported by big projects like Binance. None of the solutions has trust issues or any other problems that can be addressed by combining several of them.

ZK rollups combination may work together but has a lot of questions. The implementation is not proven to be bug-free and might have some unexpected problems. Also, there are not a lot of solutions supported as widely as optimistic. A combination of several ZK rollups might be used for testing both solutions, but not guarantee anything.

Optimistic and ZK combination. This solution was considered as the main concept to proceed. Optimistic rollups are already tested but slow, on the other hand, ZK rollups are faster but not trusted that much. The ETH community expects ZK rollups to be the future of the L2 solutions. For example, the Optimism project is already working on implementing ZK solution for the network. Based on this, even though the overall combination will have problems from both solutions, this will allow us to test ZK and direct the way for the migrating existing network without drastic changes.

7. Program implementation

7.1. Tornado Cash

Tornado Cash is a coin mixer that can be used to anonymize Ethereum transactions. Because of the logic of the blockchain, every transaction is public. If someone has some ETH on their account, they cannot transfer it anonymously, because anybody can follow the transaction history on the blockchain. Coin mixers, like Tornado Cash, can solve this privacy problem by breaking the on-chain link between the source and the destination address by using ZK proof.

When someone needs to make an anonymous payment, he can submit some amount of the ETH to the Tornado Cash contract (ex. 1 ETH). After some time, the contract will get a critical number of transactions. From this moment, the money can be released to the destination account. When there are thousands of transactions with similar amounts that are coming to the contract, and thousands of transactions that are made by the contract it is impossible to tell which money goes in which direction. This mechanism prevents anybody from tracking that pass that transaction went through. The technical challenge is that smart contract transactions are also public like any other transaction on the Ethereum network. This is the point where ZK proof will be relevant.

When someone deposits 1 ETH on the contract, he has to provide a “commitment”. This commitment is stored by the smart contract. When someone withdraws 1 ETH on the other side, he has to provide a “nullifier” and a zero-knowledge proof. The nullifier is a unique ID that is in connection with the commitment and the ZK proof proves the connection, but nobody knows which nullifier is assigned to which commitment (except the owner of the depositor/withdrawal account).

Here, it is possible to prove that a commitment is assigned to a nullifier without revealing the commitment. This part is critical for further program implementation since the implementation for the ZK proof was partially taken from this project.

7.2. Optimistic rollups

Another part of the implementation will be optimistic rollups. We have described the main details in Chapter 1. Here the interesting part is how transactions are stored and how to access the data that have been provided.

For the program implementation, we have taken the existing optimism network as a baseline. The Layer 2 chain provides all the details about new transactions that were submitted to the network. It is possible to view the block of transactions that is currently pending approval. As we know, the approval process for the optimistic rollups is not very complicated but time-consuming. This is why we are going to implement ZK proof for the block of transactions.

7.3. Data access

Probably one of the most important parts is data access. We know that the initial data obtained from a Sequencer is stored on the Layer 2 Optimism chain. Then the data is verified and the result is stored on Layer 1 ETH main net. This data can be extracted using several publicly available APIs. Such as Alchemy and Optimism Etherscan. Both of them provide access to both networks and Alchemy has a great set of API endpoints to get all of the publicly available data from the blockchain.

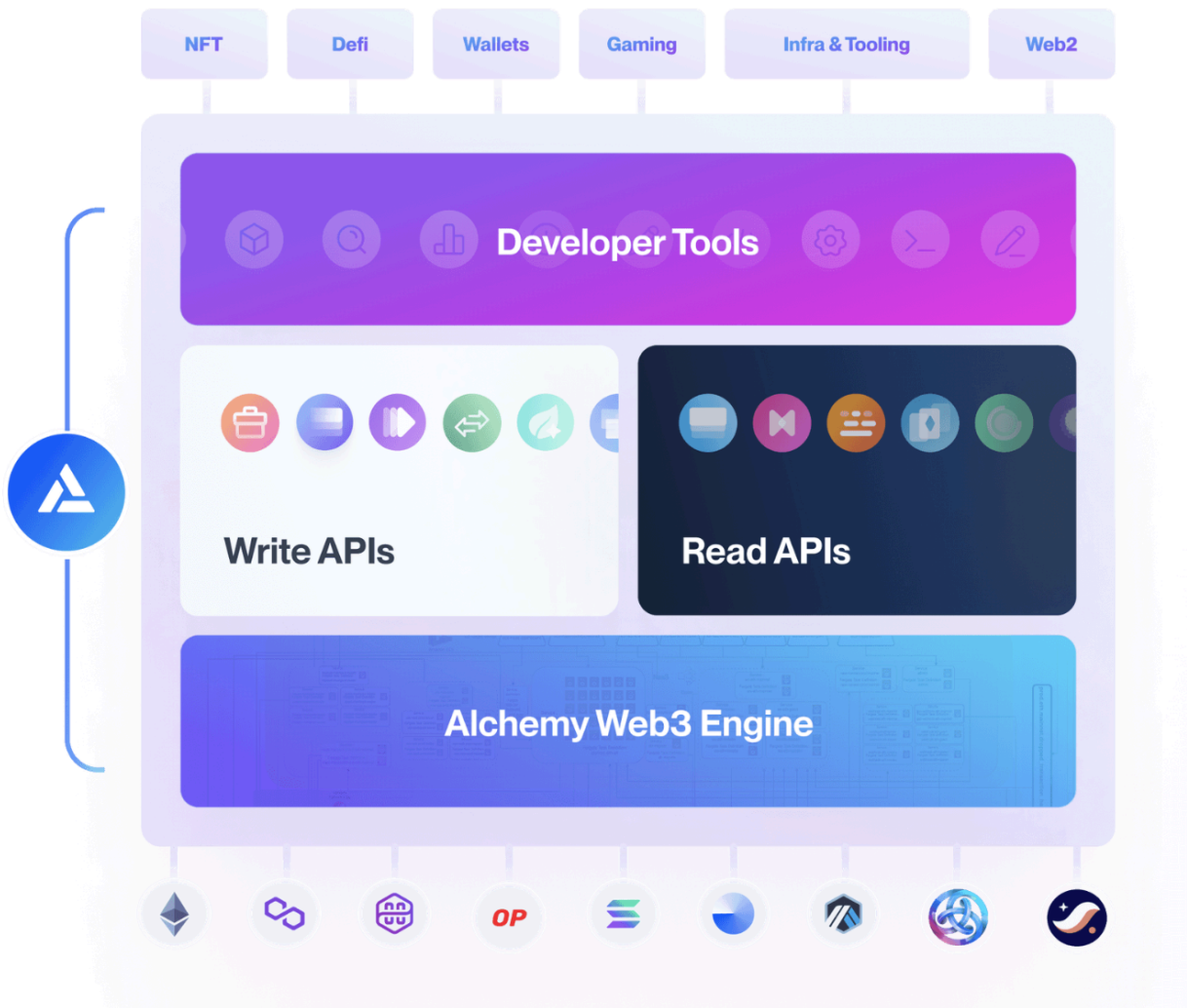
7.4. Alchemy

This is how the company describes itself: “Alchemy(Picture 8.) provides the leading blockchain development platform powering millions of users in 197 countries worldwide. Our mission is to provide developers with the fundamental building blocks they need to create the future of technology.

The Alchemy team draws from decades of deep expertise in massively scalable infrastructure, AI, and blockchain from leadership roles at technology pioneers like Google, Microsoft, Facebook, Stanford, and MIT. Backed by Stanford University, Coinbase, the Chairman of Google, Charles Schwab, and founders and executives of globally leading

organizations, Alchemy powers billions of dollars of transactions for top companies around the world.

The computer and internet fundamentally improved human life on planet Earth. We're excited to help enable the global opportunity of blockchain - the next tectonic shift." [AT24]



Picture 8: Alchemy

7.5. Optimism Etherscan

The company describes a project as: “Optimism Etherscan(Picture 9.) is the leading blockchain explorer, search, API and analytics platform for Optimism, an optimistic rollup Layer 2 scaling solution for Ethereum.

As a means to provide equitable access to blockchain data, we've developed the Optimism Etherscan Developer APIs to empower developers with direct access to Optimism Etherscan's block explorer data and services via GET/POST requests.

Optimism Etherscan's APIs are provided as a community service and without warranty, so please use what you need and no more.”



Picture 9: *Optimism logo*

7.6. Choice of data provider

As data provider was chosen Alchemy, since it is provided with some testing tokens and allows one to make up to a million requests without a paid subscription. Also, it has a powerful SDK which makes it easier to make requests and perform operations on the data. Moreover, they have great support on Discord, which responds any time during the day resolving issues with API. A couple of problems that were encountered is that SDK did not return the same data as a direct request, which was addressed by the team.

7.7. ZK solutions

In order to validate and generate the proof the snarkjs(Picture 10.) library was used.



Picture 10: *snarkJs* logo

“This is a JavaScript and Pure Web Assembly implementation of zkSNARK and PLONK schemes. It uses the Groth16 Protocol (3 point only and 3 pairings), PLONK and FFLONK.

This library includes all the tools required to perform trusted setup multi-party ceremonies: including the universal powers of tau ceremony, and the second phase circuit specific ceremonies.”[ST24]

It provides great API to generate proves, the documentation had all the necessary steps to process data and the most critical part was the possibility of generating smart contracts out of the box. The proves are non-interactive wich creates the possibility to deploy a smart contract on the chain and to use it for computation verification.

Snarkjs is used with Circom language[CT24]. Circom is a compiler written in Rust for compiling circuits written in the Circom language. The compiler outputs the representation of the circuit as constraints and everything needed to compute different ZK proofs.

It allows us to perform the ceremony with witnesses and generate the required files.

7.8. Smart Contract

The smart contract was created on solidity. Solidity is an object-oriented programming language for implementing smart contracts on various blockchain platforms, most notably, Ethereum. Solidity is licensed under GNU General Public License v3.0. Solidity was designed

by Gavin Wood and developed by Christian Reitwiessner, Alex Beregszaszi, and several former Ethereum core contributors. Programs in Solidity run on Ethereum Virtual Machine or on compatible virtual machines.

7.9. Remix

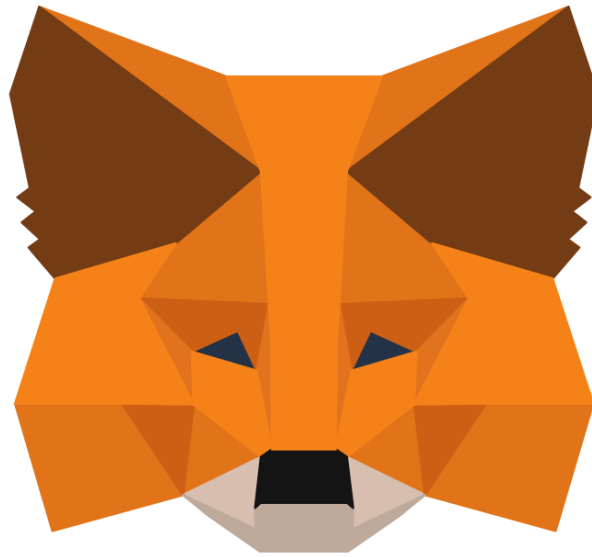
The smart contract was compiled and deployed using Remix. “Remix IDE is used for the entire journey of smart contract development by users at every knowledge level. It requires no setup, fosters a fast development cycle, and has a rich set of plugins with intuitive GUIs. The IDE comes in two flavors (web app or desktop app) and as a VSCode extension.”[RT24]

Remix provides a great tool for contract creation, validation, and deployment. After the contract is deployed, it allows running the code of the contract with direct access using Remix IDE.

7.10. Metamask

In order to deploy a contract to the network we are required to pay for the transaction. To do that we have chosen a Metamask(Picture 11.) wallet that is integrated into the browser using an extension and can be easily accessed by any website.

“MetaMask is a global community of developers and designers dedicated to making the world a better place with blockchain technology. Our mission is to democratize access to the decentralized web, and through this mission, to transform the internet and world economy to one that empowers individuals through interactions based on consent, privacy, and free association.”[MM24]

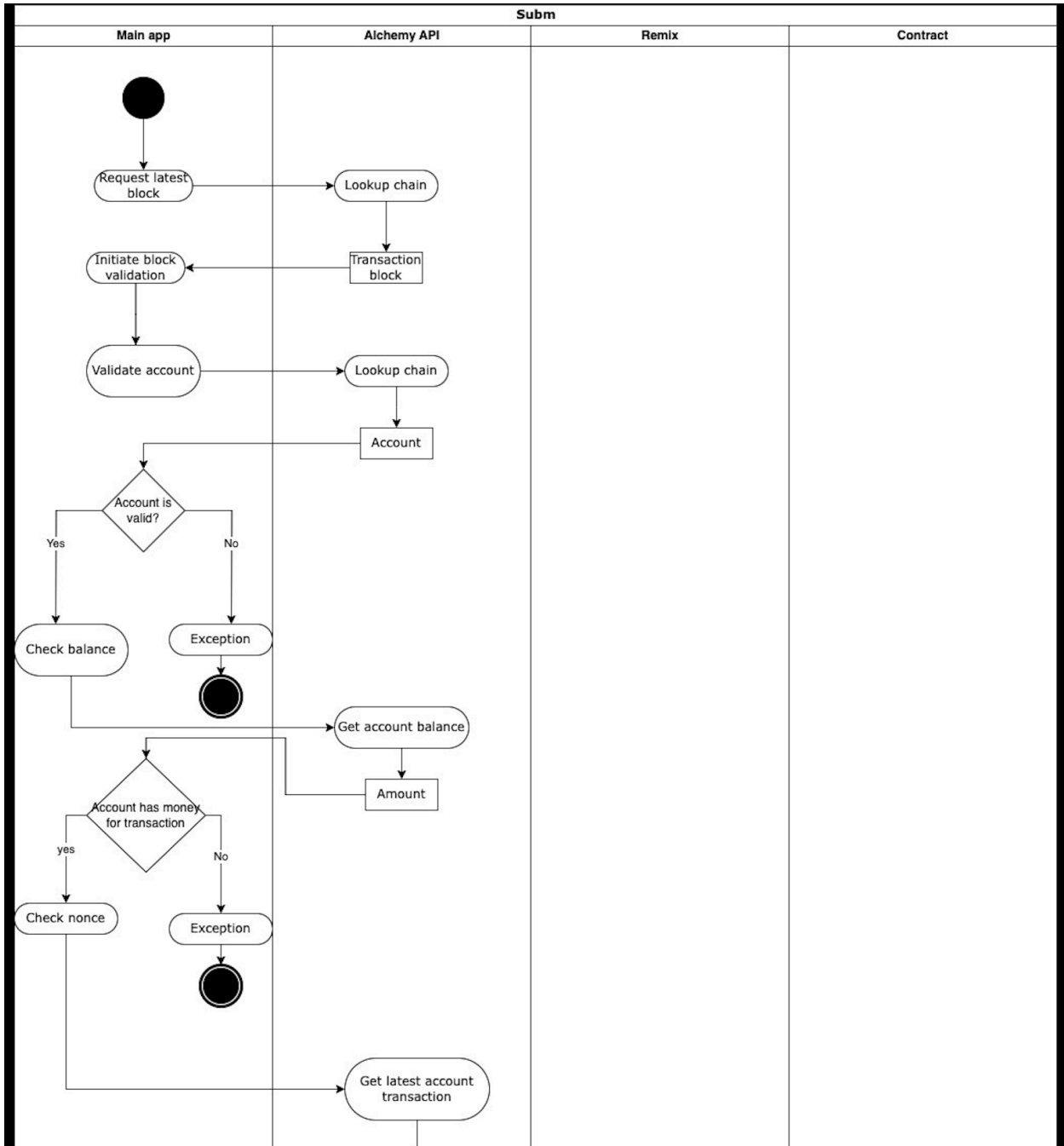


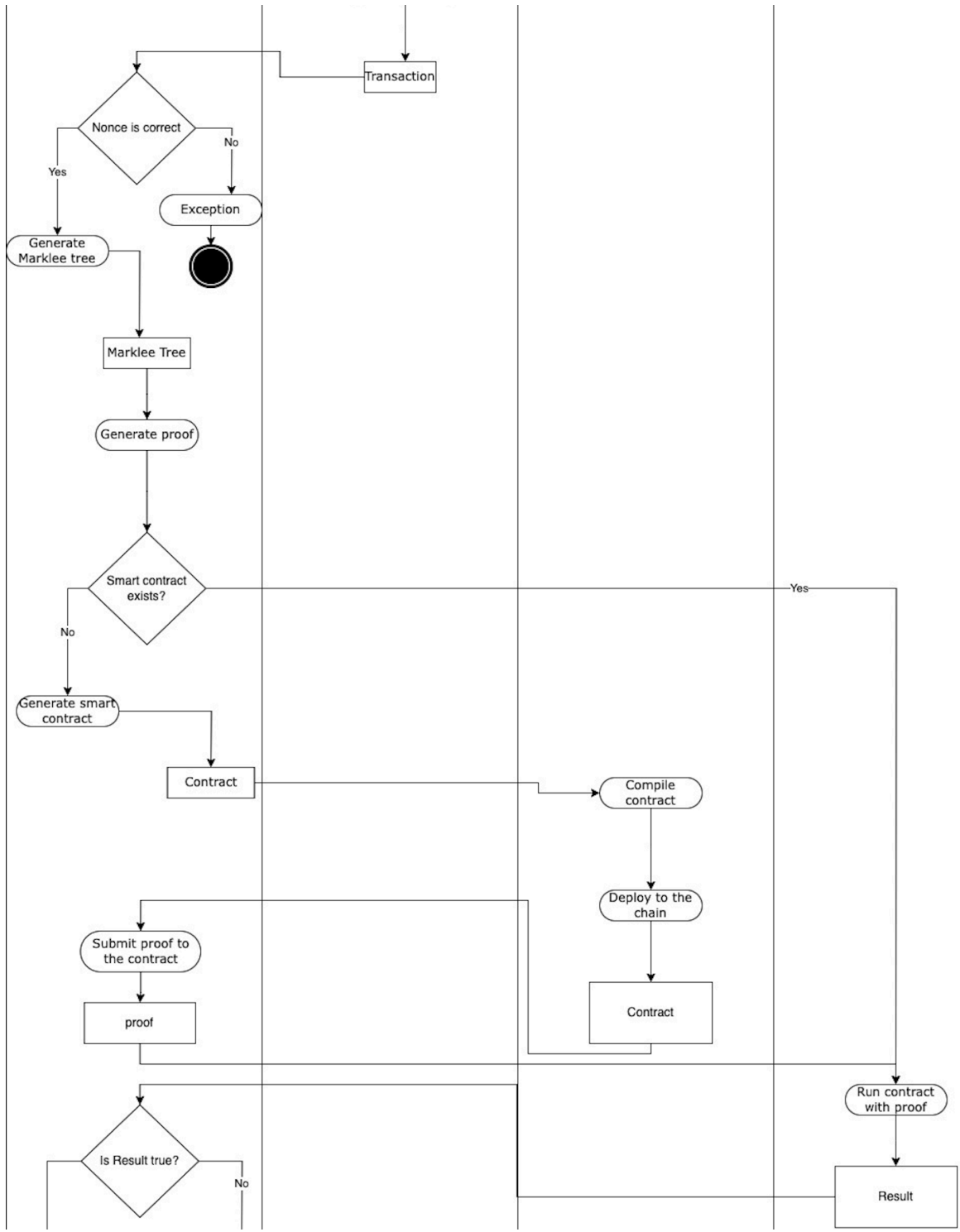
Picture 11: *Metamask*

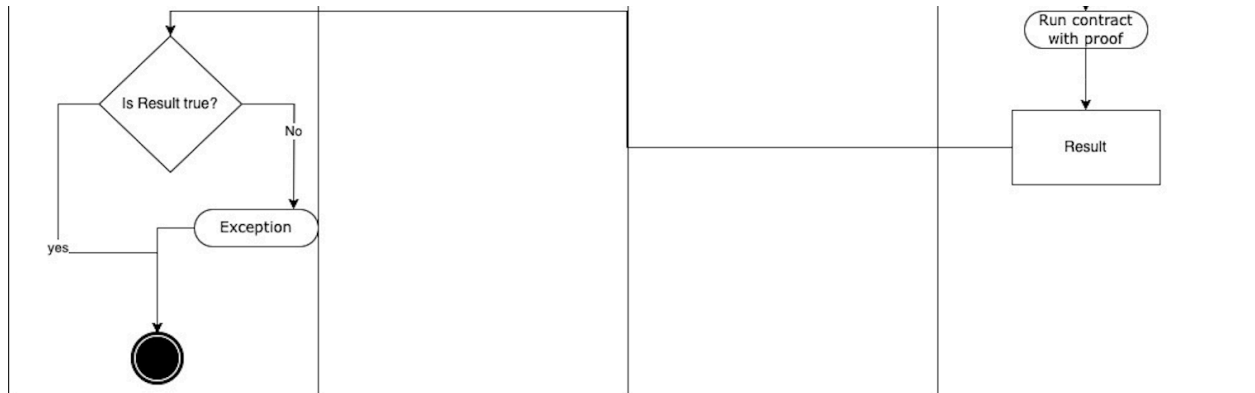
7.11. The program

7.11.1 Sequence diagram

A sequence diagram shows different processes or objects that live simultaneously, and messages exchanged between them. (picture 12.)





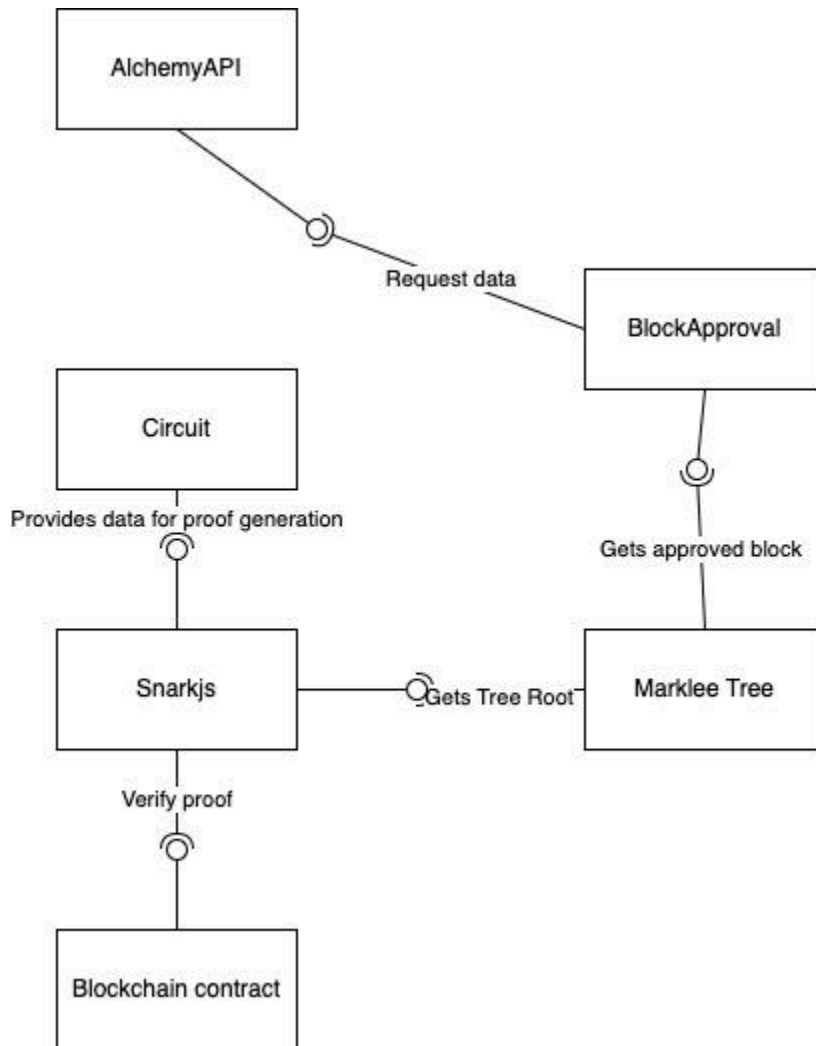


picture 12: *Sequence diagram*

The diagram above shows the flow of the program. At the start, we get the latest transaction block using AlchemyAPI. The block contains all of the transactions that are part of the block plus other metadata. We are interested in transactions. They are required to validate the block. We go over all of the transactions and check if the accounts that are used for the transaction is correct. Meaning they exist and all of the data is valid. This is done only for money transfer transactions since other types of transactions have only from address. This step is done using the AlchemyAPI method for validating an account. After that we check if the account has enough data to perform a transaction. If the account is valid and has enough money we need to make sure that transaction belongs to the account provided. In order to do that, we check the nonce of the transaction and account. Then we proceed to the most interesting step, generating a Marklee Tree. We take all of the transactions, apply the hash function then take all of the results, combine by pair, and apply the hash function. We repeat this step until there is only one hash, which is the root of the Markeele Tree. We need this root in order to generate a proof. When we have the proof there are two possibilities. First, we already deployed a smart contract to the network, then we can submit the proof to the network contract and check the result. Otherwise, there are some extra steps required. We are generating a smart contract using Snarkjs and deploying the contract using Remix.

7.11.2 Functional view

The Functional view of a system defines the architectural elements that deliver the system's functionality. (picture 13.)



picture 13: *Functional viewpoint*

AlchemyAPI is required to get blockchain data such as transactions, blocks, and accounts. This is used for BlockApproval that validates a block. The validated block is then transferred to the MarkleeTree module that creates a tree and provides a Tree Root to the Snarkjs library in order to create a proof. The proof is created using Circuit and Snarkjs. Also, Snarkjs is creating a contract that will be deployed to the blockchain and used for proof verification.

7.11.3 Proff generation

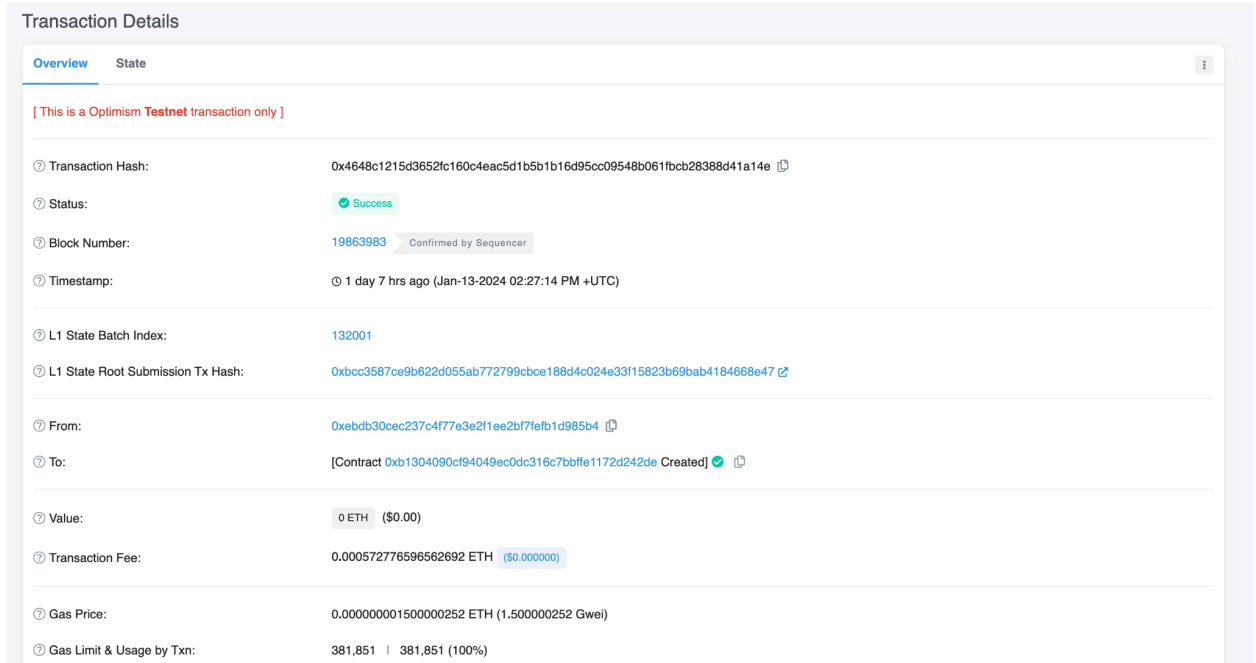
As already mentioned before the proof generation is done by Snarkjs and Circuit. The steps that are required for successful proof generation are:

1. We have written a program that can generate a proof using Circuit language and compiler.
2. Compiling the program we will get two files wasm and r1cs. Wasm file is a binary instruction format for a stack-based virtual machine. R1CS is a system of equations used to represent computations and is particularly useful in scenarios where we want to prove that a certain computation was done correctly, without revealing any other information about the inputs or the computation itself.
3. Then we need to generate a ptau file which has initial parameters for the zk-snark. Since it is not production level we have downloaded the generated file from snarkjs.
4. Having our input (Markeele Tree root) we can also generate witnesses that are required for creating proof.
5. Now we can generate proving keys by using the snarkjs, circuit, groth16 algorithm, and the ptau file. The groth16 algorithm enables a quadratic arithmetic program to be computed by a prover over elliptic curve points derived in a trusted setup, and quickly checked by a verifier.
6. After that we have enough data to generate proof. We provide snarkjs.groth16.fullProve function with the input data (Markeele Tree root), wasm file, and proving keys. This will generate our proof and file with public data.
7. We can verify our proof locally using snarkjs.groth16.verify function which requires a verification key, public information, and proof. This step is not mandatory but good to have.
8. Having verification keys we are creating a smart contract. It is written in solidity and generated by snarkjs.
9. The code for the smart contract is then deployed to the chain using Remix. For the development purpose, we decided to use Ethereum test net.
10. Using snarkjs we simulate the verification call in order to receive all of the input parameters for the contract.

11. With the given parameters we call a smart contract using Remix and get the result.

7.11.4 Smart contract

A Smart contract (picture 14.) was generated and deployed to the chain. It can be found with transaction id 0x4648c1215d3652fc160c4eac5d1b5b1b16d95cc09548b061fbc28388d41a14e.



picture 14: Contract

8. Experiment Conducted

In order to conduct an experiment, we have taken the block of transactions from the Sepolia test net. This is the test network that allows users to create transactions without paying real money. Then, the block was broken into transactions. Each transaction was verified, by checking account validity, account funds, number of transactions from the account. When the block of transactions was checked for correctness, we were able to create a Marklee Tree for the transactions inside the block. The root of the tree was used for the ZK-proof. Using snarkjs we were able to initiate tau ceremony, provide it with random data and several contributions, generate a circuit, and create the proof using the groth16 algorithm. At this point, it is possible to verify the proof locally. The next step was to generate a smart contract that could be used on a

chain so that anyone could prove the correctness of the computation. the verification contract for the blockchain was created. The contract was successfully deployed to the Sepolia network and obtained proof was submitted to the contract, which verified correctness of the proof.

8.1. Experiment hardware

MacBook Pro 2019

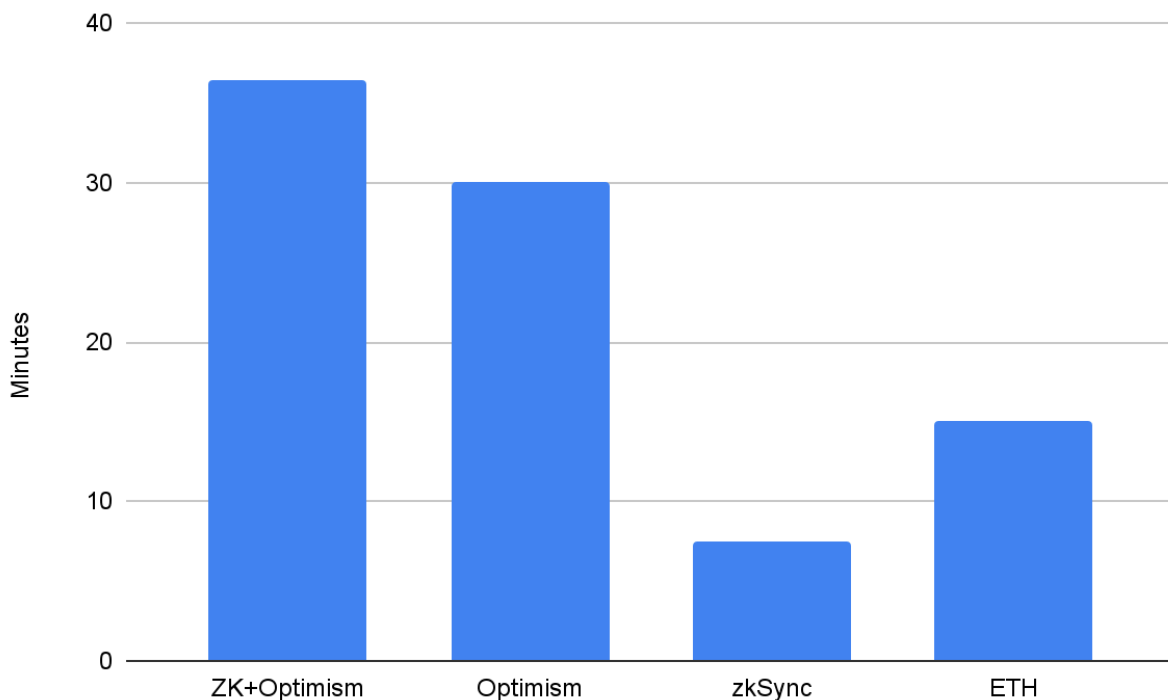
1.7 GHz Quad-Core Intel Core i7

Intel Iris Plus Graphics 645 1536 MB

16 GB 2133 MHz LPDDR3

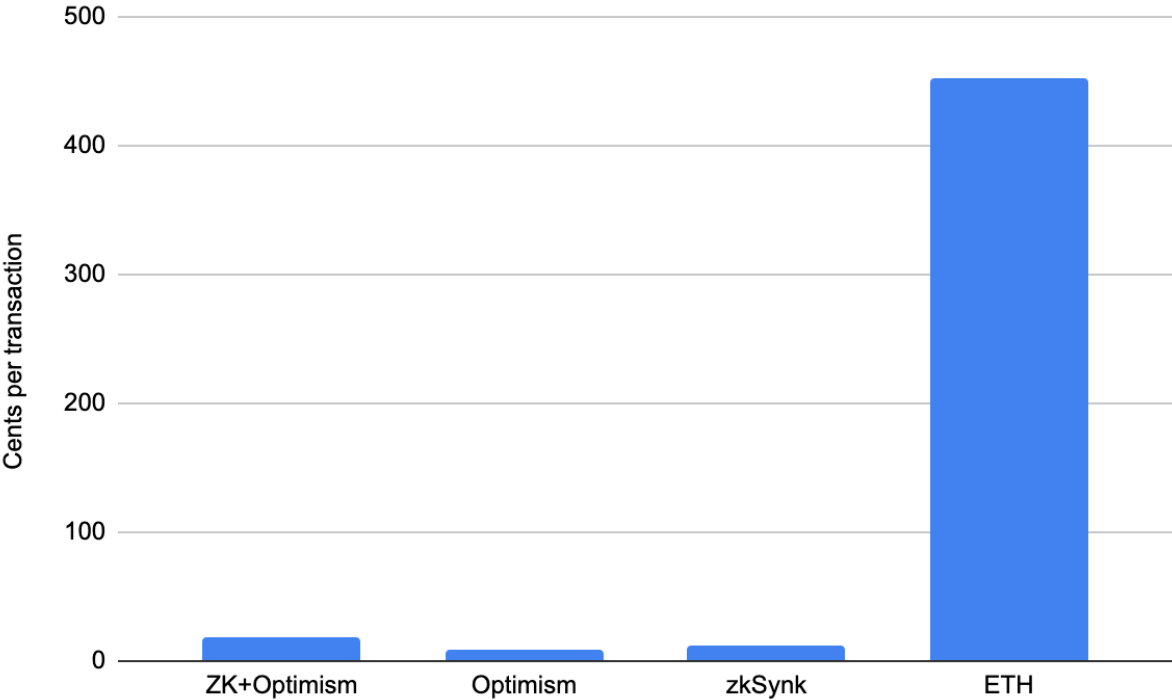
8.2 Experiments results

Approval of the transactions block for ZK part takes approximately 6 min. The time required to approve transaction on the Optimism L2 is around 30 min (picture 15.). Total less then 40 min which is more than any other network (5min zkSync, 15 min ETH).



picture 15. Transaction approval time

The transaction cost is not much higher than for separate Optimism or ZK solutions (picture 16.). Cost for transactions is computed by taking the cost of Optimism transaction (0.09\$), adding the cost for contract deployment divided by avg number of transactions in block, adding the same amount for contract execution transaction, plus max cost for Alchemy API requests per block verification and electricity cost. Total is around 18 cents which is still lower than the price of ETH transaction.



picture 16. Transaction cost

9. Results and Conclusions

As a result, we have a working prototype that uses an Optimistic implementation of the chain with the real transactions data and Tornado Cash implementation of Zero Knowledge rollups. This takes some benefits from both systems but also has some problems. The main benefit is that a combined solution can be used as a more secure way of processing transactions and testing ZK implementation. This doesn't bring any benefits to the network, but allows developers to gain trust in ZK solutions and gives us a benefit for the next possible use case.

The Optimism project already collaborated with RiscZero in order to develop a ZK solution based on the Optimism network. Our solution shows that it can be done without changing the whole network. It is possible to keep the current solution, which will save a great amount of time for the developers and build ZK on top of it. By deploying a smart contract for verification Optimism should be able to decrease the time required for transaction processing keeping the security and availability. ZK proofs are slightly more expensive than optimistic but way faster since optimism is required to have a challenge window compared to which ZK proofs are almost instant.

Definitions of the terms

Layer 1 - Ethereum main net

Layer 2 - Layer on top of Ethereum main net

Abbreviations

L1 - Layer 1

L2 - Layer 2

ZK - Zero Knowledge

OP - Optimistic

SDK - Software Development Kit

API - Application Programming Interface

Wasm - WebAssembly

R1CS - Mastering Rank-1 Constraint System

Resources

[Mus23] MUSHARRAF, Mohammad, 2023, What is the blockchain trilemma? *Ledger* [online]. 29 May 2023. [Accessed 21 June 2023]. Available from: <https://www.ledger.com/academy/what-is-the-blockchain-trilemma>

[Cry18] CRYPTOSTATS, 2018, L2fees.info. *L2Fees* [online]. 1 July 2018. [Accessed 31 January 2023]. Available from: <https://l2fees.info/>

[EthA] ETHERSCAN TEAM, [no date], Ethereum Average Gas Price Chart | Etherscan. *Etherscan* [online]. [Accessed 31 January 2023]. Available from: <https://etherscan.io/chart/tx>

[EthB] ETHERSCAN TEAM, [no date], Ethereum Average Gas Price Chart | Etherscan. *Etherscan* [online]. [Accessed 31 January 2023]. Available from: <https://etherscan.io/chart/gasprice>

[Mis] MISTER_ETH, [no date], Ethereum. *Live Ethereum TPS data* [online]. [Accessed 31 January 2023]. Available from: <https://ethstats.info/Network/Ethereum>

[DJQ22] LIN, Dan, WU, Jiaping, YUA, Qi and ZHENG, Zibin, 2022, Modeling and Understanding Ethereum Transaction Records via A Complex Network Approach. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS*. 11 November 2022, vol.67.

[Eik19] EIKI, 2019, Ethereum block structure explained. *Medium* [online]. 19 July 2019. [Accessed 7 April 2023]. Available from: <https://medium.com/@eiki1212/ethereum-block-structure-explained-1893bb226bd6>

[Opt22] OPTIMISM TEAM, 2022, L2 Chain Derivation Specification. *GitHub* [online]. 23 August 2022. [Accessed 18 April 2023]. Available from:

<https://github.com/ethereum-optimism/optimism/blob/develop/specs/derivation.md#batch-submission-wire-format>

[Kas19] KASIREDDY, Preethi, 2019, How does ethereum work, anyway? *Medium* [online]. 29 October 2019. [Accessed 12 June 2023]. Available from: <https://preethikasireddy.medium.com/how-does-ethereum-work-anyway-22d1df506369>

[Eth22] ETHEREUM COMMUNITY, 2022, Zero-knowledge rollups. *ethereum.org* [online]. 30 June 2022. [Accessed 24 April 2023]. Available from: <https://ethereum.org/en/developers/docs/scaling/zk-rollups/>

[BLN23] BERENTSEN, Aleksander, LENZI, Jeremias and NYFFENEGGER, Remo, 2023, An Introduction to Zero-Knowledge Proofs in Blockchains and Economics. *Federal Reserve Bank of St Louis*. Louis, United States, 12 May 2023.

[LZH24] LIN, Xin, ZHANG, Yuanyuan, HUANG, Changhai, XING, Bin, CHEN, Liangyin, HU, Dasha and CHEN, Yanru, 2024, An Access Control System Based on Blockchain with Zero-Knowledge Rollups in High-Traffic IoT Environments. *MDPI*. Basel, Switzerland, 24 March 2024. p. 13–18.

[Loe22] LOERAKKER, Diederik, 2022, 4844: Shard blob transactions. *EIP* [online]. 20 February 2022. [Accessed 26 March 2023]. Available from: <https://www.eip4844.com/>

[L2b23] L2BEAT RESEARCH TEAM, 2023, The state of the Layer Two ecosystem. *L2BEAT* [online]. 2023. [Accessed 12 June 2023]. Available from: <https://l2beat.com/scaling/risk>

[RL23] RISC ZERO and LAYER N, 2023, Zero-knowledge fraud proofs. *Layer N* [online]. 20 May 2023. [Accessed 12 June 2023]. Available from: https://www.layern.com/blog/zkfp?utm_source=substack&utm_medium=email

[Opt23] RFP OPTIMISM FOUNDATION, 2023: Op Stack Zero Knowledge proof. *GitHub* [online]. 30 May 2023. [Accessed 12 June 2023]. Available from: https://github.com/ethereum-optimism/ecosystem-contributions/issues/61?utm_source=substack&utm_medium=email

[AT24] Alchemy team, [2024] Alchemy Insights, Inc [online]. [Accessed 10 January 2024].
Available from: <https://www.alchemy.com/company>

[ST24] Snarkjs team, [2024] Github [online]. [Accessed 10 January 2024]
Available from: <https://github.com/iden3/snarkjs?ref=hackernoon.com>

[CT24] Circom team, [2024] Circom.io [online] [Accessed 10 January 2024]
Available from: <https://docs.circom.io/>

[RT24] Remix team, [2024] Remix [online] [Accessed 10 January 2024]
Available from: <https://remix-ide.readthedocs.io/en/latest/>

[MM24] [2024] MetaMask A Consensys Formation [online] [Accessed 12 December [2023]
Available from: <https://metamask.io/about/>

Appendix

Source code: <https://github.com/Nick9707/RWProject>