Volume Pool Doc

Maru [*]

1 References

Polygon EVM Implementation: `https://github.com/mir-protocol/plonky2/tree/main/evm`
Description of Merkle Patricia Tree: `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/#merkle-patricia-trees`
NIST Keccak Spec: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`
Keccak Team Keccak Spec: `https://keccak.team/files/Keccak-implementation-3.2.pdf`
Mir-protocol EVM doc: `https://github.com/mir-protocol/plonky2/blob/main/evm/spec/zkevm.pdf`

2 Scheme

2.1 Purpose and Design

The part of our scheme was taken from repository of plonky2/evm by Mir-protocol. The main purpose of taking Mir-evm implementation to use cross table lookups and implementation of **AllStark** with recursive aggregation of proofs with CTL to implement our idea of count trade pool. The initial version of mir EVM contains logic of proving CPU, memory, Keccak256, arithmetic, logic. In our implementation we implement logic to prove sum of volume, searching subarray in array, leafs, nodes, roots, contract data and method signature in patricia merkle path.

The **AllStark** is a structure that contains all the circuits of the scheme in order to achieve three things:

- Check the correct integration between circuits via the shared croos table lookups, to verify that the table layouts match.
- Allow having a one structure of proofs for which a proofs can be generated and verified if we build regular proofs.

[*]Version of September 18, 2023. Email: `stanislav@proxima.one`

- Allow having a single structure for which a proof can be generated and verified recursively using aggregation. The main idea to aggregate each child STARK proof to one proof that we will name **"root proof"** in terms of SNARK and verify it.

Each STARK, such as DataStark, SumStark, SearchStark... etc. are implemented as a sub STARK proof of the AllStark structure, that performs the verification of a specific part of counted volume proving of trade pool by using Patricia Merkle Tree from Node.

#Todo write about CTLs

2.2 Patricia Merkle Tree and pool data

MPT is a prefix Patricia tree in which the keys are byte sequences. In this tree, edges consist of sequences of nibbles (half-bytes). Consequently, one node can have up to seventeen children (corresponding to branches from 0x0 to 0xF). Nodes are divided into three types:

- Leaf node: A node that contains a portion of the path and a stored value. It is the endpoint in the chain.

- Branch node: A node used for branching. It contains from 1 to 17 references to child nodes. It may also contain a value.

- Extension node: An addition node that stores a portion of the path shared by multiple child nodes, as well as a reference to a branch node further along the path.

This data structure allows for efficient storage of key-value pairs while ensuring data integrity verification. You can fully read description of MPT at the following link in references. For volume proving of trade we use receipts trie because every block has its own receipts trie. A path here is: rlp(transactionIndex). transactionIndex is its index within the block it's mined. To query a specific receipt in the Receipts trie, the index of the transaction in its block, the receipt payload and the transaction type are required. The returned receipt can be of type Receipt which is defined as the concantenation of TransactionType and ReceiptPayload. To prove volume of pool trades we need to get access data from MPRT of blocks, where the data for each trade, in deserialized form, contains the following fields:

- `sold_token_name`: `String` - the name of the token being exchanged in the trade.

- `bought_token_name`: `String` - the name of the token received as a result of the exchange.

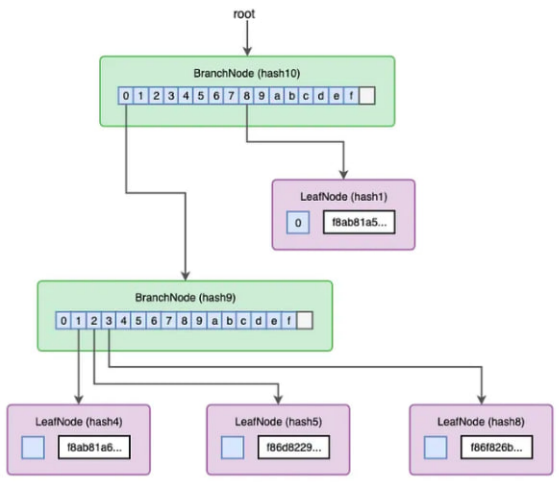


Figure 1: Merkle Patricia Receipt Trie in Ethereum

- `sold_token_volume`: `Decimal` - the amount of tokens (a floating-point decimal number) being exchanged.
- `bought_token_volume`: `Decimal` - the amount of tokens (a floating-point decimal number) received as a result of the exchange.

For each token, trades are filtered where either `sold_token_name` or `bought_token_name` matches the name of that token. After, the trade data is serialized and included in the Event Log, where the following fields are important for verification:

- `address`: The address of the contract in which the event was saved in this log. The data type of the address is `H160`, which is a fixed-size uninterpreted hash type with a size of 20 bytes (160 bits).
- `topics`: This field contains arbitrary auxiliary data in the form of a vector of data of type `H256`. `H256` is a fixed-size uninterpreted hash type with a size of 32 bytes (256 bits). The first element in the topics vector contains the hash of a string that includes the event's name, its parameters, and their types. For example, it could be computed as `keccak("TokenExchange (index_topic_1 address buyer, uint256 sold_id, uint256 tokens_sold, uint256 bought_id, uint256 tokens_bought)")`.
- `data`: This is a sequence of bytes (wrapped as a raw bytes wrapper) containing the values of the event's parameters.
- `transaction_index`: The index of the transaction, which determines the order of execution of transactions within a block.

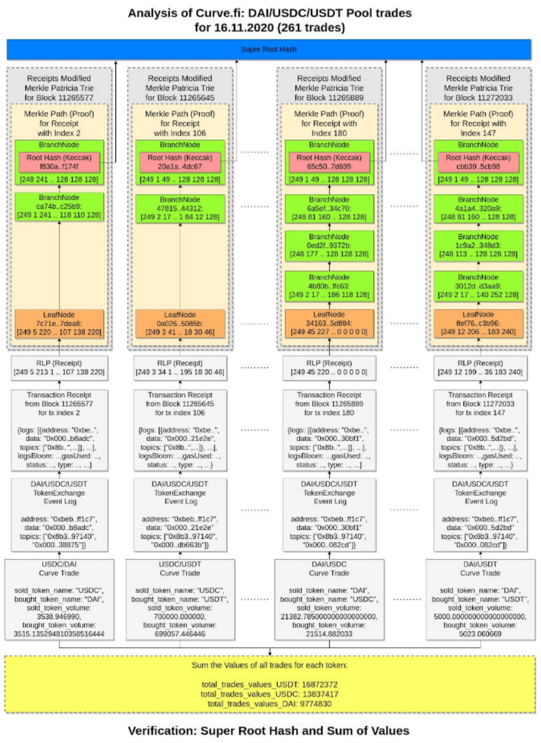
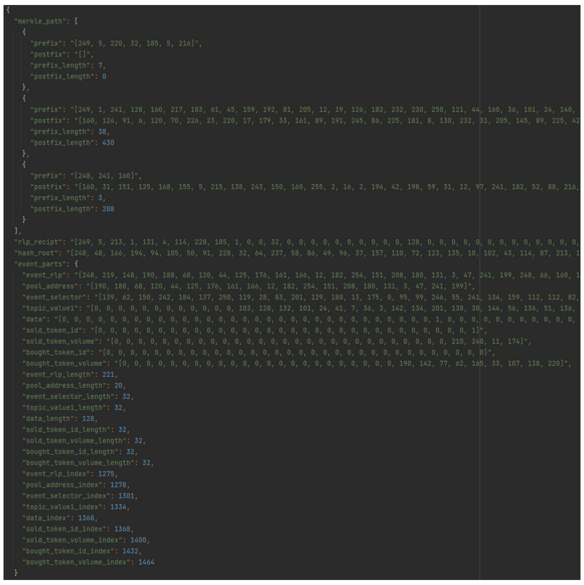

Analysis of Curve.fi: DAI/USDC/USDT Pool trades for 16.11.2020 (261 trades)
Super Root Hash
Receipts Modified Merkle Patricia Trie for Block 11265577
Merkle Path (Proof) for Receipt with Index 2
BranchNode
Root Hash (Keccak) f830a..f174f
[248 241 .. 128 128 128]
BranchNode ca74b..c25b9: [249 1 241 .. 118 110 128]
LeafNode 7c71e..7dea8: [249 5 220 .. 107 138 220]
RLP (Receipt) [249 5 213 1 .. 107 138 220]
Transaction Receipt from Block 11265577 for tx index 2
{logs: [{address: "0xbe..", data: "0x000..b8adc", topics: ["0x8b..",...]}, ...], logsBloom: ..,gasUsed: .., status: .., type: .., ...}
DAI/USDC/USDT TokenExchange Event Log
address: "0xbeb..ff1c7", data: "0x000..b8adc", topics: ["0x8b3..97140", "0x000..38875"]}
USDC/DAI Curve Trade
sold_token_name: "USDC", bought_token_name: "DAI", sold_token_volume: 3538.946990, bought_token_volume: 3515.135294810358516444
Receipts Modified Merkle Patricia Trie for Block 11265645
Merkle Path (Proof) for Receipt with Index 106
BranchNode
Root Hash (Keccak) 20a1a..4dc67
[249 1 49 .. 128 128 128]
BranchNode 47815..44312: [249 2 17 .. 1 84 12 128]
LeafNode 0a026..5085b: [249 3 41 .. 18 30 46]
RLP (Receipt) [249 3 34 1 .. 195 18 30 46]
Transaction Receipt from Block 11265645 for tx index 106
{logs: [{address: "0xbe..", data: "0x000..21e2e", topics: ["0x8b..",...]}, ...], logsBloom: ..,gasUsed: .., status: .., type: .., ...}
DAI/USDC/USDT TokenExchange Event Log
address: "0xbeb..ff1c7", data: "0x000..21e2e" topics: ["0x8b3..97140", "0x000..db663b"]}
USDC/USDT Curve Trade
sold_token_name: "USDC", bought_token_name: "USDT", sold_token_volume: 700000.000000, bought_token_volume: 699057.446446
Receipts Modified Merkle Patricia Trie for Block 11265889
Merkle Path (Proof) for Receipt with Index 180
BranchNode
Root Hash (Keccak) 65c50..7d699
[249 1 49 .. 128 128 128]
BranchNode 6a5ef..34c70: [248 81 160 .. 128 128]
BranchNode 0ed2f..9372b: [248 177 .. 128 128 128]
BranchNode 4b80b..ffc63: [249 2 17 .. 186 118 128]
LeafNode 34163..5d884: [249 45 227 .. 0 0 0 0 0]
RLP (Receipt) [249 45 220 .. 0 0 0 0 0]
Transaction Receipt from Block 11265889 for tx index 180
{logs: [{address: "0xbe..", data: "0x000..30bf1", topics: ["0x8b..",...]}, ...], logsBloom: ..,gasUsed: .., status: .., type: .., ...}
DAI/USDC/USDT TokenExchange Event Log
address: "0xbeb..ff1c7", data: "0x000..30bf1" topics: ["0x8b3..97140", "0x000..082cd"]}
DAI/USDC Curve Trade
sold_token_name: "DAI", bought_token_name: "USDC", sold_token_volume: 21382.785000000000000000, bought_token_volume: 21514.882033
Receipts Modified Merkle Patricia Trie for Block 11272033
Merkle Path (Proof) for Receipt with Index 147
BranchNode
Root Hash (Keccak) cbb39..5cb98
[249 1 49 .. 128 128 128]
BranchNode 4a1a4..320a9: [248 81 160 .. 128 128]
BranchNode 1c9a2..348d3: [248 113 .. 128 128 128]
BranchNode 3012d..d3aa9: [249 2 17 .. 140 252 128]
LeafNode 8ef76..c3b96: [249 12 206 .. 183 240]
RLP (Receipt) [249 12 199 .. 36 183 240]
Transaction Receipt from Block 11272033 for tx index 147
{logs: [{address: "0xbe..", data: "0x000..5d2bd", topics: ["0x8b..",...]}, ...], logsBloom: ..,gasUsed: .., status: .., type: .., ...}
DAI/USDC/USDT TokenExchange Event Log
address: "0xbeb..ff1c7", data: "0x000..5d2bd" topics: ["0x8b3..97140", "0x000..082cd"]}
DAI/USDT Curve Trade
sold_token_name: "DAI", bought_token_name: "USDT", sold_token_volume: 5000.000000000000000000, bought_token_volume: 5023.060669
Sum the Values of all trades for each token:
total_trades_values_USDT: 16872372
total_trades_values_USDC: 13837417
total_trades_values_DAI: 9774830
Verification: Super Root Hash and Sum of Values

Figure 2: Curve3Pool data scheme

- `transaction_hash`: The hash of the transaction.

After, Event logs are included in arrays of logs and are part of the Transaction Receipt. Afterward, the Receipt is serialized into RLP format. The most critical step in preparing data for our proof involves constructing a Merkle path (proof) on the Patricia Merkle Trie. In this example, the Merkle Path consists of three nodes: The bottom LeafNode, which includes the RLP (Receipt). The middle BranchNode, which can be visualized as having three parts, one of which should equal the hash of the LeafNode. The top BranchNode, which similarly can be represented as having three parts, one of which should equal the hash of the middle BranchNode. The data in the Merkle Path is represented as a `HashMap<H256, Bytes>`, where hash-table keys are represented as hashes (in the form of `H256`) corresponding to the values. The values are byte slices (wrapped as `Bytes`) of the corresponding Merkle Path nodes. Thus, the Merkle Path in this schema consists of three elements in the hash-table format:
`merkle_path = { leaf_hash: leaf_bytes, middle_branch_hash: middle_branch_bytes, root_branch_hash: root_branch_bytes, }`
For ZK proof, we adopt the following modified representation format for the Merkle Path:
Vec of `MerklePathElement`, where `struct MerklePathElement { prefix: Bytes, postfix: Bytes }`

A vector is used to order the path elements from the root to the leaf. Each element consists of a structure with a prefix and postfix, which are the left and right parts of the byte slice of the corresponding path element. The central part of each byte slice corresponds to the byte representation of the hash of the child element in the Merkle Path. For proving that each data of string contained in string we use start indexes of each data array from trusted setup. As example you can see the image of data structure for volume proving:



```
{
  "merkle_path": [
    {
      "prefix": "[249, 5, 220, 32, 185, 5, 216]",
      "postfix": "[]",
      "prefix_length": 7,
      "postfix_length": 0
    },
    {
      "prefix": "[249, 1, 241, 128, 160, 217, 103, 61, 45, 159, 192, 81, 205, 12, 19, 126, 182, 232, 230, 250, 121, 44, 160, 36, 101, 24, 140,
      "postfix": "[160, 124, 91, 6, 120, 70, 226, 23, 220, 17, 179, 33, 161, 89, 191, 245, 86, 225, 181, 8, 130, 232, 31, 205, 145, 89, 225, 42
      "prefix_length": 38,
      "postfix_length": 430
    },
    {
      "prefix": "[248, 241, 160]",
      "postfix": "[160, 31, 151, 125, 168, 155, 5, 215, 138, 243, 150, 160, 255, 2, 16, 2, 194, 42, 198, 59, 31, 12, 97, 241, 182, 52, 88, 216,
      "prefix_length": 3,
      "postfix_length": 208
    }
  ],
  "rlp_recipt": "[249, 5, 213, 1, 131, 4, 114, 228, 185, 1, 0, 0, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 128, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  "hash_root": "[248, 48, 166, 194, 94, 105, 50, 91, 228, 32, 64, 237, 58, 86, 49, 96, 37, 157, 110, 72, 123, 135, 10, 102, 43, 114, 87, 213, 1
  "event_parts": {
    "event_rlp": "[248, 219, 148, 190, 188, 68, 120, 44, 125, 176, 161, 166, 12, 182, 254, 151, 208, 180, 131, 3, 47, 241, 199, 248, 66, 160, 1
    "pool_address": "[190, 188, 68, 120, 44, 125, 176, 161, 166, 12, 182, 254, 151, 208, 180, 131, 3, 47, 241, 199]",
    "event_selector": "[139, 62, 150, 242, 184, 137, 250, 119, 28, 83, 201, 129, 180, 13, 175, 0, 95, 99, 246, 55, 241, 134, 159, 112, 112, 82,
    "topic_value1": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 103, 128, 132, 101, 24, 41, 7, 36, 3, 142, 134, 201, 138, 30, 144, 56, 136, 51, 136,
    "data": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    "sold_token_id": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]",
    "sold_token_volume": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 210, 240, 11, 174]",
    "bought_token_id": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]",
    "bought_token_volume": "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 190, 142, 77, 62, 165, 33, 107, 138, 220]",
    "event_rlp_length": 221,
    "pool_address_length": 20,
    "event_selector_length": 32,
    "topic_value1_length": 32,
    "data_length": 128,
    "sold_token_id_length": 32,
    "sold_token_volume_length": 32,
    "bought_token_id_length": 32,
    "bought_token_volume_length": 32,
    "event_rlp_index": 1275,
    "pool_address_index": 1278,
    "event_selector_index": 1301,
    "topic_value1_index": 1334,
    "data_index": 1368,
    "sold_token_id_index": 1368,
    "sold_token_volume_index": 1400,
    "bought_token_id_index": 1432,
    "bought_token_volume_index": 1464
}
```

Figure 3: Curve3Pool data scheme

2.3 Architecture

Our zkSNARK-proof system for the correctness of computed volume of pool trades is based on composing each STARK proofs: `ArithmeticStark`, `KeccakStark`, `KeccakSpongeStark`, `LogicStark`, `DataStark`, `SumStark`, `SearchStark` that we aggregate to one SNARK root proof with proving and verifying cross table lookups.
# To do write more

2.3.1 SumStark

To prove volume of pool trades we need to add each value of sold token volume in curve trades and prove that counted volume is correct. For this purpose we create `SumStark` using constraints and cross table lookups tables from `SumStark` and `ArithmeticStark` to prove correctness of each arithmetic addition of 256 bits elements (`U256`). In execution trace we have:

- `IS_INPUT`: indicate where in row the initial value of accumulated sum.
- `IS_OUTPUT`: indicate where in row the output of accumulated sum from previous operation will be as the operand in current row.
- `IS_RESULT`: indicate the last value of accumulated sum of `ChainResult` operations
- `ACUM_SUM`: consists of 16 limbs that represent each 16 bit per column.

The `SumStark` aims at constraining the following aspects:

- the first row of each sum operation (`ChainResult`) have to contain zero value of accumulated sum.
- `is_input` and `is_output` can be in range of $\{0, 1\}$

The main idea of proving relies on CTL, where we build two function for CTL:

- `ctl_previous_sum_equal_op1` prove that previous accumulated sum used as second operand in arithmetic operation.
- `ctl_acum_equals_result` guarantees that `acum_sum` columns have the exact value as result columns in arithmetic table.

2.3.2 SearchStark

To ensure that BranchNode hashes consist of previous hash of RLP BranchNode or LeafNode, we got the idea to search subarray of 32 bytes in array of 136 + 32 bytes (`block_size` + `postfix_bytes`) to ensure that searched subarray not will be between start of one and end of the next block. Using right cyclic shifts we create one row per shift until the first 32 bytes of `HAYSTACK_REGISTER` will be equal to 32 bytes in `NEEDLE_REGISTER`, where the searched subarray. Number of shifts will be 168 - offset in block that we got as input data from SearchOp. As we mentioned earlier as input from trusted setup we got start indexes of each data array (value, pool_address, ... etc) and offset will be equal to (`start_index_data` + `KECCAK_DIGEST_BYTES`) % `KECCAK_RATE_BYTES`. In execution trace we have:

- `HAYSTACK_SIZE`: length of array equal to 168 bytes (postfix of previous block + current block).
- `NEEDLE_SIZE`: searched subarray of 32 bytes.
- `IS_IN`: will be set to 1 in rows where the initial sub-array is present.
- `IS_OUT`: will be set to 1 where the first 32 bytes of `HAYSTACK_REGISTER` are equal to `NEEDLE_REGISTER`.

- `DUMMY_ROW`: indicates padding rows.
- `OFFSET`: number of offsets in block, after using cyclic shifts will reduced to one per shift operation .
- `IS_SHIFT`: will be set to 1 in rows where we used a right cyclic shift.
- `HAYSTACK_REGISTER`: register of array.
- `NEEDLE_REGISTER`: register of subarray.

The `SearchStark` aims to constrain the following aspects:

- Verify that the first 32 elements in the haystack are equal to the first 32 elements in the needle.
- `IS_IN` and `IS_OUT` can have values in the range of 0, 1.
- Confirm that the last element of the haystack's first row of each `SearchOp` equals the first element in the next row of the cyclic shift when `IS_IN` $= 1$, and `IS_IN` in the next row equals 0.
- Ensure that the previous element in the row of the cyclic shift equals the next element of the next row when `IS_IN` $= 0$, and the rows are not dummy rows.
- Check that the offset in the row equals 0 when `IS_OUT` $= 0$.
- Verify that the offset of the previous row is greater than one corresponding to the next row.

The main idea of the proof relies on CTL, where we build two functions for CTL proofing:

`ctl_substr_in_haystack` guarantees that the concatenated columns of `typed_data` with `zero_shift_num` (always zero) in the `Data` table, using a row filter where `child_hash_found`, `transfer_value_found`, `method_signature_found`, `contract_address_found`, and `root_hash_found` are equal to 1, equal to the concatenated columns of `NEEDLE_REGISTER` with `OFFSET` in the `Search` table using a filter where `IS_OUT` equals 1.

`ctl_haystack_equal_data` guarantees that the concatenated columns of `prefix`, `block_bytes`, and `shift_num` in the `Data` table, using a filter where `child_hash_found`, `transfer_value_found`, `method_signature_found`, `contract_address_found`, and `root_hash_found` equal 1, have the exact values of `HAYSTACK_REGISTER` in the `Search` table.

2.3.3 DataStark

The purpose of creating this STARK is to store data required to prove integrity of Keccak hashes of leafs, nodes, roots, super_root and connect other STARK while proving using CTLs. `DataOp` has structure, which consists of enum `DataType` that indicates what type of data in operation and `DataItem`, which contains additional data as offset in block and item - bytes array of hash or volume value :

```
pub(crate) struct DataOp {
    pub(crate) input: Vec<u8>,
    pub(crate) contract: Option<DataItem>,
    pub(crate) value: Option<DataItem>,
    pub(crate) child: Option<DataItem>,
    pub(crate) data_type: DataType,
    pub(crate) method_signature: Option<DataItem>,
    pub(crate) concat_data: Option<Vec<DataItem>>,
}

pub(crate) struct DataItem {
    pub(crate) offset: usize,
    pub(crate) offset_in_block: usize,
    pub(crate) item: [u8; KECCAK_DIGEST_BYTES],
}

pub enum DataType {
    Leaf = 0,
    Node = 1,
    Root = 2,
    RootData = 3,
    SuperRoot = 4,
    ConcatRoots = 5,
}
```

If we insert leaf data type then doesn't have child `DataItem` which contains of child hash and value of contract address, volume value and `method_signature` must be included. In other variants when we insert `Node`, `Root` we insert only input and `DataItem`. In execution trace we have column structure:

- `is_full_input_block`: will be 1 if this row represents a full input block, i.e. one in which each byte is an input byte, 0 otherwise.
- `len`: value of the original input length (in bytes).
- `already_absorbed_bytes`: value of input bytes that have already been absorbed prior to this block.
- `last_block`: is equal to 1 if is it the last block per `DataOp`.
- `is_leaf`: will be 1 if this operation has leaf data type.

- `is_node`: will be 1 if this operation has node data type.
- `is_root`: will be 1 if this operation has root data type.
- `is_root_data`: will be 1 if this operation has `root_data` data type.
- `is_super_root`: will be 1 if this operation has super root data type.
- `is_shadow`: has value of 1 if in one block we have `contract_address`, `method_signature`, `transfer_value` or `concat_data` (when we prove concatenation of roots in `super_root`).
- `prefix_bytes`: contain the last 32 bytes of previous block or zero bytes if this the first row of `DataOp`.
- `block_bytes`: contain the block being absorbed, which may contain input bytes and/or padding bytes.
- `contract_address_found`: will be 1 when contract address is present in block.
- `method_signature_found`: will be 1 when method signature is present in block.
- `transfer_value_found`: will be 1 when volume value is present in block.
- `child_hash_found`: will be 1 when child hash is present in block.
- `root_hash_found`: will be 1 when root hash is present in block.
- `offset_block`: equals to index indicating start of sub-array in input block with required data.
- `shift_num`: has values of cyclic shifts number required for `SearchStark` and respectively zero number of shifts.
- `zero_shift_num`: has zero value.
- `offset_object`: has value of index indicating start of sub-array in input.
- `is_concat_root`: will be 1 if this operation has concatenation roots data type.
- `typed_data`: has values of 32 bytes array containing hash, value, contract address or method signature.

The DataStark aims to constrain the following aspects:

- Each flag (full-input block, final block or implied dummy flag) must be boolean.
- Ensure that `zero_shift_num` is always zero.
- Ensure that full-input block and final block flags are not set to 1 at the same time.

- If this is a final block, the next row's original sponge state should be 0 and `already_absorbed_bytes` = 0.
- If this is a full-input block, the next row's `already_absorbed_bytes` should be ours plus 136.
- Verify that the offset of the previous row is greater than one corresponding to the next row.
- Verify that if this a `is_shadow row` the next rows have equal `already_absorbed_bytes`.
- A dummy row is always followed by another dummy row, so the prover can't put dummy rows "in between" to avoid the above checks
- Ensure that if we have more than 2 objects of 32 length in one block, `is_shadow will` be 1 and next row has the same prefix, `block_bytes` as previous row
- Ensure that if we have `method_signature_found` is equal to 1 then `typed_data` have to be equal to hard-coded sell signature of Curve 3pool trade.
- Ensure that if we have `contract_address_found` is equal to 1 then `typed_data have` to be equal to hard-coded contract address of Curve 3pool trade.
- Ensure that the distance between end of contract and start of method signature is equal to 3 (feature of location elements in Curve 3pool trade receipts).
- Ensure that the distance between end of method signature and start of volume value is equal to 67 (feature of location elements in Curve 3pool trade receipts).

The main idea of the proof relies on CTL, where we connect columns data from other STARKs:

- `ctl_data`: `block_bytes` using filter when we have leaf, node, root_data is equal to concatenation columns of `xored_rate_u32s`, `original_capacity_u32s`, `updated_state_u32s` for input block.
- `ctl_keccak_sponge`: `typed_data` of child hash have to be equal `updated_state_bytes` when it's the final input block.
- `ctl_substr_in_haystack`: The lookup guarantees that the concatenated columns of typed_data with `zero_shift_num` (always zero) in the Data table, using a row filter where `child_hash_found`, `transfer_value_found`, `method_signature_found`, `contract_address_found`, and `root_hash_found` are equal to 1, equal to the concatenated columns of `NEEDLE_REGISTER` with `OFFSET` in the `Search` table using a filter where `IS_OUT` equals 1.

- `ctl_haystack_equal_data`: The lookup guarantees that the concatenated columns of `prefix`, `block_bytes`, and `shift_num` in the `Data` table, using a filter where `child_hash_found`, `transfer_value_found`, `method_signature_found`, `contract_address_found`, and `root_hash_found` equal 1, have the exact values of `HAYSTACK_REGISTER` in the Search table.

- `ctl_concat_filled`: The lookup guarantees that roots in `typed_data` for concatenation to `super_root` are equal to the same roots in Patricia Merkle Path.

2.3.4 KeccakStark

This implementation of Stark was taken from mir-protocol respectively `LogicStark`, `KeccakSpongeStark`. The purpose of this Stark is to prove integrity of permutations as a part of building Keccak. Columns that are used in trace rows generation:

- `START_PREIMAGE` are used to hold the original input to a permutation, i.e. the input to the first round.

- `START_A` # TODO

- `START_C` are used to hold `xor(A[x, 0], A[x, 1], A[x, 2], A[x, 3], A[x, 4])` operation

- `START_C_PRIME` are used to hold `xor(C[x, z], C[x - 1, z], C[x + 1, z - 1])` operation

- `START_A_PRIME` are used to hold `xor(A[x, y], C[x - 1], ROT(C[x + 1], 1))` operation

- `START_A_PRIME_PRIME` are used to hold `xor(B[x, y], and (B[x + 1, y], B[x + 2, y]))` operation

- `START_A_PRIME_PRIME_0_0_BITS` #TODO

- `REG_A_PRIME_PRIME_PRIME_0_0_LO` are used to hold lower bits of `START_A_PRIME_PRIME_0_0_BITS`

- `REG_A_PRIME_PRIME_PRIME_0_0_HI` are used to hold higher bits of `START_A_PRIME_PRIME_0_0_BITS`

- `REG_FILTER` is a register which indicates if a row should be included in the CTL. Should be 1 only for certain rows which are final steps, i.e. with ‘`reg_step(23) = 1`‘.

The KeccakStark aims to constrain the following aspects:

- The `REG_FILTER` must be 0 or 1 and if this is not the final step, the REG_FILTER must be off.
- If this is not the final step, the local and next preimages must match.
- $C'[x, z] = C[x, z] \oplus C[x - 1, z] \oplus C[x + 1, z - 1]$.
- Check that the input limbs are consistent with A' and D. $A[x, y, z] = A'[x, y, z] \oplus D[x, y, z] = A'[x, y, z] \oplus C[x-1, z] \oplus C[x+1, z-1] = A'[x, y, z] \oplus C[x, z] \oplus C'[x, z]$.
- $$\bigoplus_{i=0}^{4} A'[x, i, z] = C'[x, z]$$

  so for each x, z, diff $\cdot(\text{diff} - 2) \cdot (\text{diff} - 4) = 0$, where diff $= \sum_{i=0}^{4} A'[x, i, z] - C'[x, z]$.
- $A''[x, y]$ = xor(B[x, y], and (B[x + 1, y], B[x + 2, y])).
- $A'''[0, 0] = A''[0, 0] \oplus \text{RC}$.
- Enforce that previous round's output equals the next round's input.

The `ctl_keccak` function is used in proving STARK using CTL to prove that concatenation columns of `xored_rate_u32s`, `original_capacity_u32s`, `updated_state_u32s` in `KeccakSpongeStark` equal to concatenated permutation input and output in `KeccakStark`.

2.3.5 KeccakSpongeStark

The purpose of this Stark is to prove integrity of sponge operations as a part of building Keccak. Columns that are used in trace rows generation:

- `is_full_input_block`: 1 if this row represents a full input block, i.e. one in which each byte is an input byte, not a padding byte; 0 otherwise.
- `timestamp`: the timestamp at which inputs should be read from memory,
- `len`: The length of the original input, in bytes,
- `already_absorbed_bytes`: the number of input bytes that have already been absorbed prior to this block,
- `is_final_input_len`: If this row represents a final block row, the 'i'th entry should be 1 if the final chunk of input has length 'i' (in other words if 'len - already_absorbed == i'), otherwise 0. If this row represents a full input block, this should contain all 0s,
- `original_rate_u32s`: the initial rate part of the sponge, at the start of this step,

- `original_capacity_u32s`: the capacity part of the sponge, encoded as 32-bit chunks, at the start of this step,
- `block_bytes`: the block being absorbed, which may contain input bytes and/or padding bytes,
- `xored_rate_u32s`: the rate part of the sponge, encoded as 32-bit chunks, after the current block is xor'd in,but before the permutation is applied.,
- `updated_state_u32s`: the entire state (rate + capacity) of the sponge, encoded as 32-bit chunks, after the permutation is applied.
- `updated_state_bytes`: the entire state of the sponge as 32 bytes.

The `KeccakSpongeStark` aims to constrain the following aspects:

- Each flag (full-input block, final block or implied dummy flag) must be boolean.
- Ensure that full-input block and final block flags are not set to 1 at the same time.
- If this is the first row, the original sponge state should be 0 and `already_absorbed_bytes` = 0.
- If this is a final block, the next row's original sponge state should be 0 and `already_absorbed_bytes` = 0.
- If this is a full-input block, the next row's "before" should match our "after" state.
- If this is a full-input block, the next row's `already_absorbed_bytes` should be ours plus 136.
- A dummy row is always followed by another dummy row, so the prover can't put dummy rows "in between" to avoid the above checks.
- If this is a final block, `is_final_input_len` implies '`len - already_absorbed == i`'.

The functions are used in proving STARK using CTL:

- `ctl_data`: `block_bytes` using filter when we have leaf, node, root_data in `Data` table is equal to concatenation columns of `xored_rate_u32s`, `original_capacity_u32s`, `updated_state_u32s` for input block in `KeccakSponge` table.
- `ctl_keccak_sponge`: `typed_data` of child hash in Data table have to be equal to `updated_state_bytes` in `KeccakSponge` table when it's the final input block.

- `ctl_keccak`: prove that concatenation columns of `xored_rate_u32s`, `original_capacity_u32s`, `updated_state_u32s` in `KeccakSpongeStark` equal to concatenated permutation input and output in `KeccakStark`.
- `ctl_logic`: prove that concatenation columns of `original_rate_u32s`, `block_bytes`, `xored_rate_u32s` and flag of xor operation in `KeccakSponge` table is equal to concatenation of `input0(256 bits)`, `input1(256 bits)` and flag of xor operation. This function guarantees that xor operation of state and `block_bytes` is correct.

2.3.6 LogicStark

The initial goal of using this Stark to prove logical operations(xor, and, or) using `CPUStark` in EVM, but for our needs we use the Stark only for xor operation proving with state and input blocks in `KeccakSpongeStark`. Columns used in generation of trace rows:

- `IS_AND`: flag that describes `AND` operation, must be boolean ;
- `IS_OR`: flag that describes `OR` operation, must be boolean ;
- `IS_XOR`: `IS_OR + 1`;
- `INPUT0`: left operand(256 bits) of operation that are decomposed from `U256`;
- `INPUT1`: right operand(256 bits) of operation that are decomposed from `U256`;
- `RESULT`: the result of operation that is packed in 32 bits from `U256`;

The `LogicStark` aims to constrain the following aspects:

- Ensure that all bits are indeed bits.
- The result will be 'input0 OP input1 = sum_coeff * (input0 + input1) + and_coeff * (input0 AND input1)'. AND => sum_coeff = 0, and_coeff = 1, OR => sum_coeff = 1, and_coeff = -1, XOR => sum_coeff = 1, and_coeff = -2

The functions are used in proving STARK using CTL:

- ctl_logic: prove that concatenation columns of `original_rate_u32s`, `block_bytes`, `xored_rate_u32s` and flag of xor operation in `KeccakSponge` table is equal to concatenation of `input0(256 bits)`, `input1(256 bits)` and flag of xor operation. This function guarantees that xor operation of state and `block_bytes` is correct.

2.3.7 ArithmeticStark

The goal of using this Stark to prove volume of trade pool as a chain of arithmetic addition operation using `SumStark` through CTL. Columns used in generation of trace rows:

- flags that describe arithmetic operation using boolean value: `IS_ADD`, `IS_MUL`, `IS_SUB`, `IS_DIV`, `IS_MOD`, `IS_MULMOD`, `IS_ADDFP254`, `IS_MULFP254`, `IS_SUBFP254`, `IS_SUBMOD`, `IS_LT`, `IS_GT`, `IS_BYTE`
- `SHARED_COLS`: there are shared columns which can be used by any arithmetic circuit, depending on which one is active this cycle.
- `INPUT_REGISTER_0`, `INPUT_REGISTER_1`, `OUTPUT_REGISTER` are columns to register left operand, right operand and result of operation as 256 bits.
- `AUX_INPUT_REGISTER_0`, `AUX_INPUT_REGISTER_1`, `AUX_INPUT_REGISTER_DBL` are columns for auxiliary input that are used two rows per operation (for example, `MULMOD`).
- `AUX_REGISTER_0`, `AUX_REGISTER_1`, `AUX_REGISTER_2` are used to help with overlapping of the general input columns because they correspond to the values in the second row for modular operations.

`ArithmeticStark` consists of parts of arithmetic operations that aims to constrain addition, multiplication, operations with bytes, sub, division, operation of is greater and lower. For each arithmetic operation we will describe constraints. Addcy.rs is used to prove addition, sub, operation of greater and lower. The structure of operations looks like: `X + Y = Z + CY *` 2^{256}, by an appropriate assignment of "inputs" and "outputs" to the variables `X`, `Y`, `Z` and `CY`.
Specifically, `ADD`: `X + Y`, inputs `X`, `Y`, output `Z`, ignore `CY`
`SUB`: `Z - X`, inputs `X`, `Z`, output `Y`, ignore `CY`
`GT`: `X` $>$ `Z`, inputs `X`, `Z`, output `CY`, auxiliary output `Y`
`LT`: `Z` $<$ `X`, inputs `Z`, `X`, output `CY`, auxiliary output `Y`
Operations proving aims to constrain:

- $\sum_i (x_i + y_i) \cdot 2^{16i} = \sum_i z_i \cdot 2^{16i} + \text{given\_cy} \cdot 2^{256}$.
- If `N_LIMBS = 1`, then this amounts to verifying that either $x_0 + y_0 = z_0$ or $x_0 + y_0 == z_0 + \text{cy} \cdot 2^{16}$.

Byte.rs is used to prove operations with bytes. The structure of operations looks like:
INPUTS: 256-bit values I and $X = \sum_{i=0}^{31} X_i B^i$, where $B = 2^8$ and $0 \leq X_i < B$ for all i.
OUTPUT: X_{31-I} if $0 \leq I < 32$, otherwise 0.

To prove that the bytes x and y are in the range [0, 2^8). Mir-protocol do the following: Instead of storing x and y, we store w = 256 * x and y. Then, to verify that x, y < 256 and the last limb L = x + y * 256, we check that L = w / 256 + y * 256. The proof of why verifying that L = w / 256 + y * 256 suffices is as follows:

1. The given L, w, and y are range-checked to be less than 2^{16}.
2. $y \cdot 256 \in \{0, 256, 512, \ldots, 2^{24} - 512, 2^{24} - 256\}$
3. $\frac{w}{256} = L - y \cdot 256 \in \{-2^{24} + 256, -2^{24} + 257, \ldots, 2^{16} - 2, 2^{16} - 1\}$
4. By inspection, for $w < 2^{16}$, if $\frac{w}{256} < 2^{16}$ or $\frac{w}{256} \geq P - 2^{24} + 256$ (i.e., if $\frac{w}{256}$ falls in the range of point 3 above), then $w = 256 \cdot m$ for some $0 \leq m < 256$.
5. Hence $\frac{w}{256} \in \{0, 1, \ldots, 255\}$
6. Hence $y \cdot 256 = L - \frac{w}{256} \in \{-255, -254, \ldots, 2^{16} - 1\}$
7. Taking the intersection of ranges in 2. and 6., we see that $y \cdot 256 \in \{0, 256, 512, \ldots, 2^{16} - 256\}$
8. Hence $y \in \{0, 1, \ldots, 255\}$

Bytes operations proving aims to constrain:

- Verify that idx0_hi is the high (11) bits of the first limb of idx (in particular idx0_hi is at most 11 bits, since idx[0] is at most 16 bits).
- Verify the layers of the tree.
- Check byte decomposition of last limb.
- idx_is_large can be 0 or 1.
- If hi_limb_sum is nonzero, then idx_is_large must be one.
- If idx_is_large is 1, then hi_limb_sum_inv must be the inverse of hi_limb_sum, hence hi_limb_sum is non-zero, hence idx is indeed "large". Otherwise, if idx_is_large is 0, then hi_limb_sum * hi_limb_sum_inv is zero, which is only possible if hi_limb_sum is zero, since hi_limb_sum_inv is non-zero.
- Check that the rest of the output limbs are zero.

Mul.rs is used to prove multiplication. The structure of operation looks like:

$$C = A \cdot B \pmod{2^{256}},$$

i.e. C is the lower half of the usual long multiplication $A \cdot B$. Inputs A and B, and output C, are given as arrays of 16-bit limbs. For example, if the limbs of A are $a[0], \ldots, a[15]$, then

$$A = \sum_{i=0}^{15} a[i]\beta^i,$$

where $\beta = 2^{16} = 2^{\text{LIMB\_BITS}}$. To verify that A, B and C satisfy the equation we proceed as follows. Define

$$a(x) = \sum_{i=0}^{15} a[i]x^i$$

(so $A = a(\beta)$) and similarly for $b(x)$ and $c(x)$. Then $A \cdot B = C \pmod{2^{256}}$ if and only if there exists q such that the polynomial

$$a(x) \cdot b(x) - c(x) - x^{16} \cdot q(x)$$

is zero when evaluated at $x = \beta$, i.e. it is divisible by $(x - \beta)$; equivalently, there exists a polynomial s (representing the carries from the long multiplication) such that

$$a(x) \cdot b(x) - c(x) - x^{16} \cdot q(x) - (x - \beta) \cdot s(x) = 0$$

As we only need the lower half of the product, we can omit $q(x)$ since it is multiplied by the modulus $\beta^{16} = 2^{256}$. Thus we only need to verify

$$a(x) \cdot b(x) - c(x) - (x - \beta) \cdot s(x) = 0$$

Multiplication aims to constrain:

- Constraint poly holds the coefficients of the polynomial that must be identically zero for this multiplication to be verified.
- $a(x) \cdot b(x) - c(x) = (x - \beta) \cdot s(x)$ where $s(x) = \sum_i \texttt{aux\_limbs}[i] \cdot x^i$.
- `constr_poly` holds the coefficients of the polynomial $a(x)b(x) - c(x) - (x - \beta)s(x)$. The multiplication is valid if and only if all of those coefficients are zero.

Modular.rs is used to prove `ADDMOD`, `MULMOD` and `MOD` operations. The structure of operation looks like: $C = \text{operation}(A, B) \mod M$.

Where operation can be addition, multiplication, or just returning the first argument (for MOD). Inputs A, B, and M, and output C, are given as arrays of 16-bit limbs. For example, if the limbs of A are $a[0], \ldots, a[15]$, then

$$A = \sum_{i=0}^{15} a[i]\beta^i,$$

where $\beta = 2^{16} = 2^{\text{LIMB\_BITS}}$. To verify that A, B, M, and C satisfy the equation, we proceed as follows. Define

$$a(x) = \sum_{i=0}^{15} a[i]x^i$$

(so $A = a(\beta)$) and similarly for $b(x)$, $m(x)$, and $c(x)$. Then, $\text{operation}(A, B) = C \mod M$ if and only if there exists q such that the polynomial

$$\text{operation}(a(x), b(x)) - c(x) - m(x) \cdot q(x)$$

is zero when evaluated at $x = \beta$; equivalently, there exists a polynomial s such that

$$\text{operation}(a(x), b(x)) - c(x) - m(x) \cdot q(x) - (x - \beta) \cdot s(x) = 0.$$

Modular aims to constrain:

- `mod_is_zero` is zero or one.
- `mod_is_zero` is zero if modulus is not zero (they could both be zero).
- `MODULAR_DIV_DENOM_IS_ZERO` is 1 if the operation is DIV and the denominator is zero.
- Verify that the output is reduced, i.e. `output` $<$ `modulus`.
- `constr_poly_copy` holds the coefficients of the polynomial

  $$\text{operation}(a(x), b(x)) - c(x) - q(x) \cdot m(x) - (x - \beta) \cdot s(x)$$

  where `operation` is `add`, `mul`, or $|a, b| \to a$. The modular operation is valid if and only if all of those coefficients are zero.