Designing the TVM side of the TON Trustless Bridge system smart contracts and exploring zk-proofs for verifying validator signatures

by RSquad Blockchain Lab on behalf of TON Foundation

v1.1, 15 May 2023

Contents

1 Ethereum light clients overview

1.1 Introduction on how Ethereum 2.0 light clients work

Light clients are essential components in blockchain ecosystems as they enable users to access and interact with a blockchain in a secure and decentralized way, without having to synchronize the full blockchain. A light client or light node is a software program that connects to full nodes to interact with the blockchain. Unlike full nodes, light nodes do not need to run continuously or read and write extensive amounts of data from the blockchain. Instead, they rely on full nodes as intermediaries. Light clients use full nodes for various operations, from requesting the latest headers to inquiring about an account's balance. The design of light client protocols allows them to interact with full nodes in a trust-minimized manner.

In particular, Sync Committees are an important feature for Ethereum light clients introduced with the Shanghai upgrade. Sync Committees are groups of full nodes that are selected at random intervals to generate and share block headers. These headers are then verified by light clients to ensure that they are valid and that the full nodes are honest. This process allows light clients to obtain information about the blockchain in a trust-minimized way, without relying on a single trusted source. Instead, they can verify information from multiple sources and make informed decisions based on the consensus of the network.

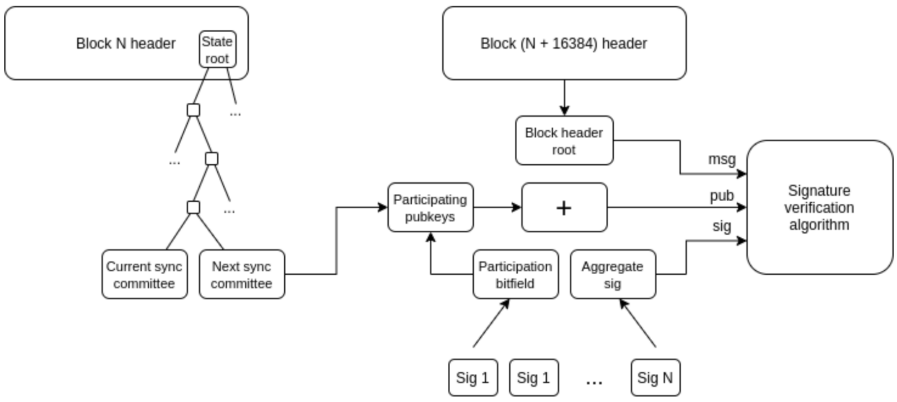

Figure 1: Light client follows to learn about more recent blocks

Figure 1 illustrates the basic procedure that a light client follows to learn about more recent blocks. As new blocks are added to the blockchain, the full nodes in the Sync Committee generate headers for these blocks and share them with the other full nodes in the committee. The light client then requests these headers from one or more full nodes and verifies them to ensure their validity. This process allows the light client to stay up to date with the latest blocks

while minimizing the amount of data it needs to synchronize.

1.2 What is Sync Committee and how does it work?

The recent Altair upgrade includes a key feature designed to support light client syncing called the sync committee. This committee of 512 validators is randomly selected every sync committee period (approximately 1 day) and is responsible for continually signing the block header that is the new head of the chain at each slot. The purpose of the sync committee is to allow light clients to keep track of the chain of beacon block headers.

The sync committee is updated infrequently and saved directly in the beacon state, which allows light clients to verify the sync committee with a Merkle branch from a block header they already know about. They can then use the public keys in the sync committee to directly authenticate signatures of more recent blocks. This function is critical for light clients to authenticate block headers since computing the proposer or attesters at a given slot requires a calculation on the entire active validator set, which light clients do not have access to.

Assuming a light client already has a block header at slot N, in period $X = \frac{N}{16384}$, and wants to authenticate a block header somewhere in period $X + 1$, the light client follows these steps:

1. Use a Merkle branch to verify the `next_sync_committee` in the slot N post-state. This is the sync committee that will be signing block headers during period $X + 1$.

2. Download the aggregate signature ofthe newer header that the light client is trying to authenticate.

3. Add together the public keys of the subset of the sync committee that participated in the aggregate signature (the bitfield in the signature will tell you who participated).

4. Verify the signature against the combined public key and the newer block header. If verification passes, the new block header has been successfully authenticated!

The minimum cost for light clients to track the chain is only about 25 kB per two days, which is the size of the sync committee and a few other necessary bytes. This low cost is intended to make the beacon chain light client-friendly for extremely constrained environments, such as mobile phones, embedded IoT devices, in-browser wallets, and other blockchains.

1.3 How does verification occur?

To begin, a light client must obtain the block headers of the blockchain. When making requests to a full node, the light client need not trust it completely.

Block headers contain a Merkle tree root, which acts like a unique identifier for all blockchain data concerning account balances and smart contract storage. Even the slightest change in the data will alter the fingerprint. Assuming that most miners are honest, the block headers and their fingerprints are deemed valid. While a light client may need to request information, such as an account's balance, from a full node, it can verify the information's authenticity using the fingerprints for each block. This provides a powerful means to authenticate information without prior knowledge of it.

1.4 How does synchronization occur?

A light node can synchronize with a blockchain much faster than a full node as it requires only a fraction of the information. While it takes about an hour to synchronize the entire Ethereum mainnet blockchain with a light client, anything more than a couple of seconds would be too much for any application. To enable quick syncing with the top of the blockchain, light clients have a trusted blockchain checkpoint built into their code. This checkpoint allows the client to download only the latest headers, resulting in a sync that takes just a few seconds. While this tradeoff requires users to trust the client developers to integrate a valid checkpoint, it is considered acceptable as users already need to trust the developers for the client implementation.

The sync protocol is essential for light clients, and the following details explain how it functions:

- The sync committee is a subset of validators attests to block $N - 1$ in every block N chosen from existing Ethereum validators.
- This subset of validators only rotates every approx. 27 hours, chosen as a happy medium between data load and opportunity to corrupt committee members.
- The sync committee is a relatively large and conservative size of 512 validators out of the approx. 138,000 currently on mainnet to ensure safety.
- The current and next sync committee is included on-chain.
- The light client syncs by tracking each sync committee across time, syncing forward committee by committee and ensuring that each committee handoff has a $\frac{2}{3}$ vote that verifies the next.
- Sync committee participants attest to the current state of the chain, specifically the previous block, and these attestations are aggregated into a single signature named "SyncAggregate". This signature is included in every new block, verifying its predecessor. It is different from "Attestations" put on chain via beacon committees.

Instead of initializing the node with the entire beacon state, the light client downloads a historical block header and the current and next sync committee at

that block. It can then download and track the current and next sync committee instead of the entire validator set. Finally, the light client can progress the chain by downloading only the last SyncAggregate from a sync committee (with $\frac{2}{3}$ of the committee signed) and a merkle proof to the next sync committee.

1.5 Specifics on the working process.

The working process of the light client involves the use of a REST API to fetch updates and request proofs. This process consists of three endpoints that provide functionality to get historical sync updates, get the latest sync update, and get multi-proofs for a beacon state. These endpoints create sync update objects and multi-proofs from beacon states, respectively. The REST API aims to become the new standard for light clients in Eth2, and its consensus is sought.

The light client can be initialized from a trusted state root or a trusted snapshot after startup. Once initialized, it requests updates from its current synced position up to the finalized state. The light client also exposes functionality to request proofs verified against the currently synced state root.

To retrieve the necessary data, the light client calls the `eth_getProof` method from the RPC provider. The beacon node implementation, which facilitates Phase 0 of the Ethereum consensus layer, exposes this API. The data obtained from this API is in JSON format.

All of Ethereum's execution layer's Merkle trees use a Merkle Patricia tree. From a block header, there are three roots from three of these trees: `stateRoot`, `transactionsRoot`, and `receiptsRoot`. Verification algorithms are used to verify the validity of the data.

In terms of data storage, the light client implements Verkle tree "Stateless" State Management and Proof Verification. The support for Verkle tree is not yet implemented, but it is planned for the architecture. Additionally, the light client supports EIP-4844 Shard Blob Transactions Spec Finalization, and the list of supported transaction types according to EIP-2718 is limited to two, so the addition of new transaction types does not affect it. The status of these updates can be implemented in testnets before moving to the mainnet.

2 Architectural design approach

Within the subsystem, smart contracts need to be developed. The basic structure consists of three types of contracts:

1. Ethereum Light Client
2. Bridge
3. Adapters

2.1 Smart Contract "Ethereum Light Client"

A light client stores the latest known state of Ethereum and the logic of updating this state. Essentially, the smart contract stores the current state of the blockchain without a detailed history and has a method `update` for verifying and updating this state.

Here's how the state is stored and updated[1]:

A light client maintains its state in a store object of type `LightClientStore` and receives update objects of type `LightClientUpdate`. Every update triggers `process_light_client_update(store, update, current_slot)` where `current_slot` is the current slot based on some local clock.

The `LightClientSnapshot` represents the light client's view of the most recent block header that the light client is convinced is securely part of the chain. The light client stores the header itself, so that the light client can then ask for Merkle branches to authenticate transactions and state against the header. The light client also stores the current and next sync committees, so that it can verify the sync committee signatures of newer proposed headers.

```
class LightClientSnapshot(Container):
    # Beacon block header
    header: BeaconBlockHeader

    # Sync committees corresponding to the header
    current_sync_committee: SyncCommittee

    next_sync_committee: SyncCommittee


class LightClientStore(object):
    snapshot: LightClientSnapshot

    valid_updates: Set[LightClientUpdate]


class LightClientUpdate(Container):
    # Update beacon block header
    header: BeaconBlockHeader
```

[1] https://github.com/ethereum/annotated-spec/blob/master/altair/sync-protocol.md

```
    # Next sync committee corresponding to the header
    next_sync_committee: SyncCommittee

    next_sync_committee_branch:
        Vector[Bytes32, floorlog2(NEXT_SYNC_COMMITTEE_INDEX)]

    # Finality proof for the update header
    finality_header:
        BeaconBlockHeader

    finality_branch:
        Vector[Bytes32, floorlog2(FINALIZED_ROOT_INDEX)]

    # Sync committee aggregate signature
    sync_committee_bits: Bitvector[SYNC_COMMITTEE_SIZE]

    sync_committee_signature: BLSSignature
    # Fork version for the aggregate signature

    fork_version: Version
```

The snapshot can be updated in two ways:

1. If the light client sees a valid `LightClientUpdate` containing a `finality_header`, and with at least $\frac{2}{3}$ of the sync committee participating, it accepts the `update.header` as the new snapshot header. Note that the light client uses the signature to verify `update.finality_header` (which would in practice often be one of the most recent blocks, and not yet finalized), and then uses the Merkle branch from the `update.finality_header` to the finalized checkpoint in its post-state to verify the `update.header`. If `update.finality_header` is a valid block, then `update.header` actually is finalized.

2. If the light client sees no valid updates via method for a sufficiently long duration (specifically, the length of one sync committee period), it simply accepts the speculative header in `valid_updates` with the most signatures as finalized.

To reduce the amount of data stored in the smart contract, it will be necessary to shorten the stored data. For example, there is no point in storing unsigned and incorrect updates. Therefore, valid updates can be finalized directly into the snapshot.

2.2 Smart Contract "Bridge"

The smart contract receives events from Ethereum and additional information. It then passes them to the light client for verification, and upon successful verification (proof that the event was actually included in the transaction in an Ethereum block), it passes the data for execution to the Adapter smart contract.

2.3 Smart Contract "Adapter"

The smart contract performs an action (such as minting a token) based on its code and the data passed to it. It works on the same principle as Adapter smart contracts in Ethereum. This is necessary in order to expand the functionality of the bridge for any operation.

2.4 Additional Information

In Ethereum, there are four trees, such as the transaction tree, which stores what the client signed and sent, which is not necessary for solving the problem. We only need the `receiptsRoot`, which is a Merkle tree with receipts. Through it, we can prove the result of the event (since logs or events are based on the `TransactionReceipts`, the only way to prove them is by proving the `TransactionReceipt` each event belongs to).

```
verifyMerkleProof(
    # Root
    blockHeader.transactionReceiptRoot,

    # Key or Path
    keccak256(proof.txIndex),

    # Serialized nodes starting with the root-node
    proof.merkleProof,

    # expected value
    transactionReceipt
)
```

3 List of improvements for TVM

This section provides a list of potential improvements to the TVM (TON Virtual Machine) that may be required to implement the TON Trustless Bridge. These enhancements are focused on the verification of Sync Committee signatures from Ethereum to validate a block. The following subsection also outlines the necessary steps for verifying the signatures.

3.1 Merkleization

The SSZ specification[2] provides guidelines for breaking down data into chunks for storage on a blockchain, as observed in Ethereum 2.0, to facilitate processing by the Ethereum Virtual Machine (EVM).

For TON Trustless bridge important to support the ability to perform merkleization is essential to derive a hash from a merkle tree for validating the correctness of event's signature in a user-transmitted block.

The excerpt from the specification mentioned below outlines the necessary functionality. It is worth noting that the functions are not mandatory to implement, but their inclusion in the TVM (TON Virtual Machine) would significantly reduce development and gas costs.

Note: The functions presented below are optional; however, they are desirable. Although the required functionality can be achieved using FunC, integrating these functions into TVM would considerably lower development and gas expenses.

We first define helper functions:

- `size_of(B)`, where `B` is a basic type: the length, in bytes, of the serialized form of the basic type.
- `chunk_count(type)`: calculate the amount of leafs for merkleization of the type.
  - all basic types: `1`
  - `Bitlist[N]` and `Bitvector[N]`: $\frac{N+255}{256}$ (dividing by chunk size, rounding up)
  - `List[B, N]` and `Vector[B, N]`, where `B` is a basic type: $\frac{(N*size\_of(B)+31)}{32}$ (dividing by chunk size, rounding up)
  - `List[C, N]` and `Vector[C, N]`, where `C` is a composite type: `N`
  - containers: `len(fields)`
- `pack(values)`: Given ordered objects of the same basic type:
  1. Serialize `values` into bytes.

[2] `https://github.com/ethereum/consensus-specs/blob/dev/ssz/simple-serialize.md#merkleization`

2. If not aligned to a multiple of `BYTES_PER_CHUNK` bytes, right-pad with zeroes to the next multiple.
3. Partition the bytes into `BYTES_PER_CHUNK`-byte chunks.
4. Return the chunks.

- `pack_bits(bits)`: Given the bits of bitlist or bitvector, get `bitfield_bytes` by packing them in bytes and aligning to the start. The length-delimiting bit for bitlists is excluded. Then return `pack(bitfield_bytes)`.
- `next_pow_of_two(i)`: get the next power of 2 of `i`, if not already a power of 2, with 0 mapping to 1. Examples: `0->1, 1->1, 2->2, 3->4, 4->4, 6->8, 9->16`
- `merkleize(chunks, limit=None)`: Given ordered `BYTES_PER_CHUNK`-byte chunks, merkleize the chunks, and return the root:
  - The merkleization depends on the effective input, which must be padded/limited:
    - if no limit: pad the `chunks` with zeroed chunks to `next_pow_of_two(len(chunks))` (virtually for memory efficiency).
    - if `limit >= len(chunks)`, pad the `chunks` with zeroed chunks to `next_pow_of_two(limit)` (virtually for memory efficiency).
    - if `limit < len(chunks)`: do not merkleize, input exceeds limit. Raise an error instead.
  - Then, merkleize the chunks (empty input is padded to 1 zero chunk):
    - If `1` chunk: the root is the chunk itself.
    - If `> 1` chunks: merkleize as binary tree.
- `mix_in_length`: Given a Merkle root `root` and a length `length` (`"uint256"` little-endian serialization) return `hash(root + length)`.
- `mix_in_selector`: Given a Merkle root `root` and a type selector `selector` (`"uint256"` little-endian serialization) return `hash(root + selector)`.

We now define Merkleization `hash_tree_root(value)` of an object `value` recursively:

- `merkleize(pack(value))` if `value` is a basic object or a vector of basic objects.
- `merkleize(pack_bits(value), limit=chunk_count(type))` if `value` is a bitvector.
- `mix_in_length(merkleize(pack(value), limit=chunk_count(type)), len(value))` if `value` is a list of basic objects.

- `mix_in_length(merkleize(pack_bits(value), limit=chunk_count(type)), len(value))` if `value` is a bitlist.
- `merkleize([hash_tree_root(element) for element in value])` if `value` is a vector of composite objects or a container.
- `mix_in_length(merkleize([hash_tree_root(element) for element in value], limit=chunk_count(type)), len(value))` if `value` is a list of composite objects.
- `mix_in_selector(hash_tree_root(value.value), value.selector)` if `value` is of union type, and `value.value` is not `None`
- `mix_in_selector(Bytes32(), 0)` if `value` is of union type, and `value.value` is `None`

3.2 Signature Verification and Curve Operations

To work with Sync Committees in TON Trustless Bridge, BLS is required. There are three possible options for BLS support.

3.2.1 Full BLS Support in TVM (Preferred Option)

Ideally, TON Trustless Bridge would have full BLS support, which would provide access to the entire BLS interface within TVM (except `Sign`). With full BLS support, TON developers could implement intended functionality and have expanded capabilities.

Ethereum uses BLS signatures as specified in the IETF draft BLS specification `draft-irtf-cfrg-bls-signature-02`[3] but with Hashing to Elliptic Curves - `draft-irtf-cfrg-hash-to-curve-07`[4] instead of `draft-irtf-cfrg-hash-to-curve-06`. The following interfaces are implemented with the `BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_` ciphersuite:

- ```
  def Sign(
  SK: int,
  message:  Bytes
  ) -> BLSSignature // optional
  ```
- ```
  def Verify(
  PK: BLSPubkey,
  message:  Bytes,
  signature:  BLSSignature
  ) -> bool
  ```
- ```
  def Aggregate(
  signatures:  Sequence[BLSSignature]
  ) -> BLSSignature
  ```

[3] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-02
[4] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-07
  ```

- ```
  def FastAggregateVerify(
  PKs: Sequence[BLSPubkey],
  message: Bytes,
  signature: BLSSignature
  ) -> bool
  ```
- ```
  def AggregateVerify(
  PKs: Sequence[BLSPubkey],
  messages: Sequence[Bytes],
  signature: BLSSignature
  ) -> bool
  ```

3.2.2 Only Necessary Functions

For the TON Trustless Bridge to work with Ethereum Sync Committee, only the `FastAggregateVerify`[5] function is required. If an implementation of this function is available, nothing else is required in the context of the current task.

```
// BLS12-381
FastAggregateVerify((PK_1, ..., PK_n), message, signature)
```

3.2.3 Only primitives

In this case, functions for pairing, addition, and multiplication operations are required to implement `FastAggregateVerify` or the entire BLS on FunC (pairing may be optional). So there is no `FastAggregateVerify`, the analogs of `ECADD`, `ECMUL`, and `pairing check` precompiled contracts on the `alt_bn128` elliptic curve[6] are required.

[5] `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-02#section-3.3.4`

[6] `https://eips.ethereum.org/EIPS/eip-1108`

4 Utilizing BLS to optimize delivery of TON validator signatures.

4.1 Problem

In the Ethereum part of the TON Trustless Bridge system, one of the use cases involves sending TON validator signatures to prove blocks directly in Ethereum.

The goal is to aggregate these signatures, using BLS or other methods, as ed21559 does not support non-interactive aggregation, with the aim of optimizing gas costs when performing iterations.

4.2 Overview of the subject area

The signatures used in TON are Ed25519-signatures generated with a validator's private keys of the sha256 of the concatenation of the 256-bit representation hash of the block and its 256-bit serialization hash `blk_serialize_hash`. The 64-bit keys in dictionary signatures represent the first 64 bits of the public keys of the corresponding validators[7].

TON utilizes two forms of elliptic curve cryptography: Ed25519 is used for cryptographic Schnorr signatures, while Curve25519 is used for asymmetric cryptography. These curves are used in the standard way. One unique adaptation of these curves for TON is that TON supports automatic conversion of Ed25519 keys into Curve25519 keys, so that the same keys can be used for signatures and for asymmetric cryptography[8]. Another source confirms that TON uses EDDSA ed25519 curve signature verification[9].

ECDSA stands for Elliptic Curve Digital Signature Algorithm, and EdDSA stands for Edwards-curve Digital Signature Algorithm. Both are used to create digital signatures.

The Edwards-curve Digital Signature Algorithm (EdDSA) is used to create a digital signature using an enhancement of the Schnorr signature with Twisted Edwards curves. One example of EdDSA is Ed25519, which is based on Curve 25519. It provides around 128-bit security and generates a 64-byte signature value of (R,s). Along with this, it has 32-byte values for the public and private keys.

ECDSA signatures change each time based on the nonce used, whereas EdDSA signatures do not change for the same set of keys and the same message. ECDSA public keys are (x,y) coordinates and thus have 512 bits (for secp256k1), while Ed25519 uses only the y-coordinate value for the point, and thus has 256 bits.

In theory, optimizing the algorithm for verifying a large number of signatures in TON can be achieved by caching intermediate results to avoid recomputing them for each signature. This means that verifying a set of 64 signatures allows

[7] `https://ton.org/tblkch.pdf` (5.1.8. Signed shardchain block)
[8] `https://ton.org/tblkch.pdf` (Appendix A. Elliptic curve cryptography)
[9] `https://medium.com/orbit-chain/orbit-bridge-now-supports-ton-network-c421c32c46b0`

developers to save time by storing intermediate results in registers and reusing them in the next iterations.

4.3 BLS

Schnorr signatures are great, but the BLS signatures are even better and newer. If we implement BLS signatures correctly, we can combine all signatures and public keys in a transaction into a single key and signature, and nobody will be able to identify multiple keys. Additionally, block validation can be faster since we can validate all signatures at once. However, there are some issues with Schnorr signatures that the BLS signatures can solve:

- A multisig scheme requires several communication rounds, which can be inconvenient with cold storage.
- With signature aggregation, we have to rely on a random number generator. In ECDSA, we can choose a random point R deterministically, but not in signature aggregation.
- The m-of-n multisig scheme is complex. We need to create a merkle tree of public keys, which can be large for large values of m and n.
- We can't combine all signatures in the block to a single signature.

BLS signatures can fix all of these issues. We don't need random numbers at all, and we can combine all signatures in the block into a single signature. The m-of-n multisig is also straightforward, and we don't need several communication rounds between signers. Moreover, BLS signatures are twice as short as Schnorr or ECDSA signatures. This is because the BLS signature is a single curve point, rather than a pair. This feature is critical to the scheme.

One great thing about the BLS scheme is that we don't need several communication rounds between devices. We just need to know who the other signers are, and we're all set. This is much simpler than the 3-round multisig scheme required for Schnorr signatures. Additionally, BLS signatures are a completely deterministic signature algorithm, so they don't rely on any randomness.

The extra structure provided by the bilinear map allows us to construct an efficient aggregate signature scheme. However, we cannot build efficient aggregate signatures from general gap groups.

Therefore, we cannot use existing ed25519 signatures for BLS aggregation. For the formation of a multiparty signature, based on the first paragraph, it is necessary to involve signers interactively with the transmission of relatively large amounts of data (public keys and random big numbers for each signer). Thus, even implementing such a scheme in TON and forcing validators to participate in this would not be expedient (it is better to switch to BLS immediately). Also, we cannot use existing signatures because they are not curve points, and the curve is different. Therefore, even if we come up with a way to aggregate public keys on this curve, we won't be able to aggregate signatures.

Since we cannot change the validator's working scheme, we need to look for a solution using ZK-SNARKs. Here, we need to construct a mathematical proof of the existence of valid signatures in such a way that verifying the proof of 50-100 signatures is cheaper than verifying the signatures themselves.

4.4 ZK-SNARKs

In the hypothesis, it's possible to verify EdDSA signatures using SNARKs. A brief overview of the technology is provided below (for verifying a single signature).

To verify an EdDSA signature, you need the signer's public key and the signature. The public key is a tuple (x,y) that represents a point on the twisted Edward curve, and it includes the parameters of the curve. The signature is a tuple (R,S), where R is a point on the curve, and S is a scalar used to perform a scalar multiplication on the curve.

The signature verification involves checking the following relation:

$$[2c * S]G = [2c]R + [2cH(R, A, M)]A$$

where G is the base point of the twisted Edward curve, k is the secret key of the signer, A is its public key, and H is the hash function used for signing. The variable c is either 2 or 3, depending on the twisted Edwards curve.

zk-SNARKs cannot be directly applied to any computational problem; you need to convert the problem into a "quadratic arithmetic program" (QAP), which is a highly nontrivial transformation. You also need a witness to the QAP, which corresponds to the input of the code. After that, you need to create the zero-knowledge proof for this witness, and a separate process for verifying a proof.

The QAP is converted into a rank-1 constraint system (R1CS), which is a sequence of groups of three vectors (a, b, c). The solution to an R1CS is a vector s that satisfies the equation $s.a * s.b - s.c = 0$.

There are many different algorithms for implementation, the most popular one being Groth16, invented in 2016, but it is currently not the most efficient. Below is a list of algorithms that can be used to solve the problem:

- Groth16[10]: Groth16 is currently the fastest and smallest data-volume zk-SNARK being used in Zcash, etc. CRS of Groth16 is not universal and its settings need to be bound to a specific circuit. It is often used by new zk-SNARKs to compare performance because of its speed and the small amount of data it proves.

- Sonic[11]: Sonic is an early universal zk-SNARK protocol that supports universal and upgradeable reference strings. The paper was published in January 2019. Sonic has a fixed proof size but high verification cost, which theoretically allows multiple proofs to be verified in batches for

[10] `https://eprint.iacr.org/2016/260`
[11] `https://eprint.iacr.org/2019/099`

better performance. Many of the new zk-SNARKs listed below are based on Sonic.

- Fractal[12]: Fractal is a zk-SNARK that allows recursion. The transparent setup is achieved by preprocessing the circuit. The maximum size of the proof is 250KB, which is much larger than the proofs generated by other builds.
- Halo[13]: Halo supports recursive evidence organization without trusted settings, and unlike other new zk-SNARK builds, Halo's verification time is linear.
- SuperSonic[14]: An improved version of Sonic, the first transparent zk-SNARK that is practical in terms of verification time and proof data volume.
- Marlin[15]: An improved version of Sonic, the proof time is reduced by 10 times, and the verification time is reduced by 4 times.
- Plonk[16]: An improved version of Sonic, the proof time is reduced by 5 times.

Conclusion — this approach requires a deeper study of possible protocols and test implementation, which is a separate object for research. For now, we can focus on optimizing the caching of intermediate calculations during EdDSA signature verification, which can help optimize verification and bring system resource consumption to acceptable values.

[12] https://eprint.iacr.org/2019/1076
[13] https://eprint.iacr.org/2019/1021
[14] https://eprint.iacr.org/2019/1229
[15] https://eprint.iacr.org/2019/1047
[16] https://eprint.iacr.org/2019/953