Zpoken, OU.
Harju maakond, Tallinn,
Kesklinna linnaosa, Sakala tn 7-2, Estonia, 10141

ZKP BASED LIGHT CLIENT RESEARCH

By Zpoken team

Monthly progress (March 2023)

Zpoken, OU.
Harju maakond, Tallinn,
Kesklinna linnaosa, Sakala tn 7-2, Estonia, 10141

DOCUMENT GUIDE

SECTION I: Test case #4.1: recursive proof of the computational integrity of the chain of epochal blocks with verification of the correctness of digital signatures of producers and validators

The general scheme for the formation of a chain of recursive proofs is shown in fig. 2.

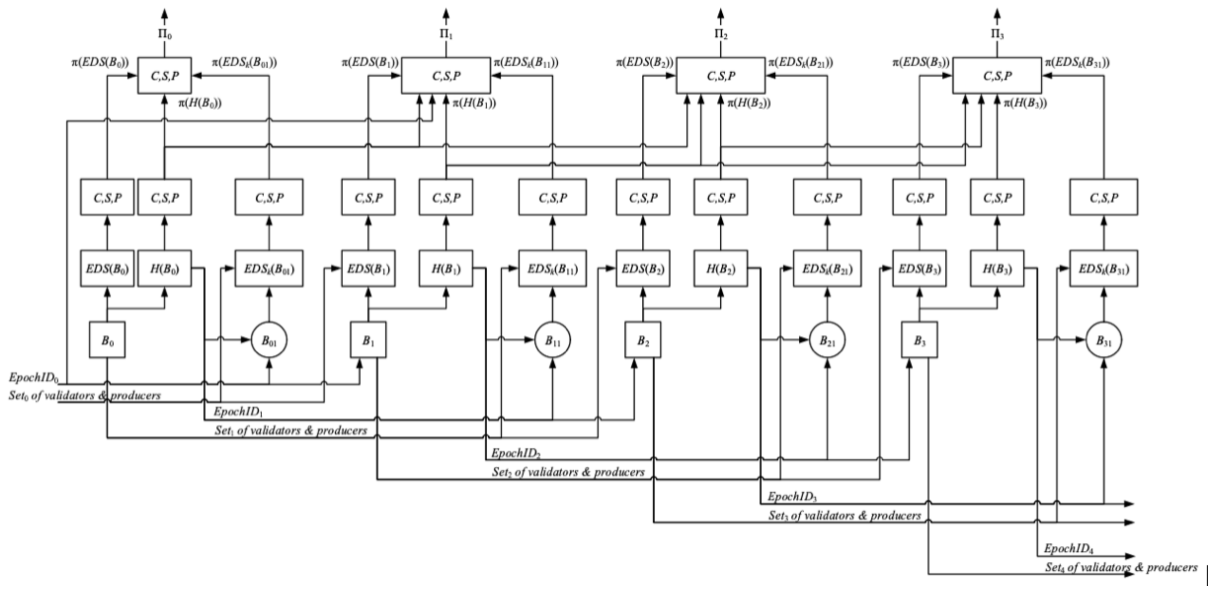

Fig. 1 - Cryptographic chain of epochal blocks (simplified scheme)

It includes:

1. Proof generation $\pi(H(B_i))$ of the CI of a hash $H(B_i)$ of each epochal block;
2. Proof generation $\pi(EDS(B_i))$ of the CI of a digital signature $EDS(B_i)$ of the producer of the epochal block. The list of block producers must match the list in *Set of validators & producers* of the penultimate epoch;
3. Proof generation $\pi(EDS_k(B_{i1}))$ of the CI of digital signatures of the validators of the first epoch block $EDS_k(B_{i1})$. The list of block validators must match the list in Set_{i-1} *of validators & producers* of the penultimate epoch;
4. Proof aggregation:
    a. generating proof of CI $\prod_0 = (\pi(H(B_0)), \pi(EDS(B_0)), \pi(EDS_k(B_{01})))$ of epoch block B_0 by aggregating:
        i. a proof of the correct hashing $\pi(H(B_0))$ of the epoch block B_0;
        ii. a proof of the correct digital signature $\pi(EDS(B_0))$ of the epoch block producer B_0;

iii. a proof of correct digital signatures $\pi(EDS_k(B_1))$ of first epoch block validators B_{01};

b. generating a proof of CI $\prod_i = (\pi(H(B_i)), EpochId, \pi(H(B_{i-1})), \pi(EDS(B_i)), \pi(EDS_k(B_{i1})))$ of each block by aggregating:

i. a proof of the correct hashing $\pi(H(B_i))$ of the epoch block B_i;

ii. *EpochId* is an initial epoch identifier (constant);

iii. a proof of the correct hashing $\pi(H(B_{i-1}))$ of the epoch block B_{i-1}, i.e. the correctness of the calculation of the epoch identifier $EpochId_{i+1} = H(B_{i-1})$;

iv. a proof of the correct digital signature $\pi(EDS(B_i))$ of the block producer B_i;

v. a proof of correct digital signatures $\pi(EDS_k(B_{i1}))$ of block validators B_{i1}, i.e. first epoch block with id $EpochId_{i+1} = H(B_{i-1})$.

In this test case, we assume that the Set_{i-1} *of validators & producers* contains a list of 3 validators for the epoch with id $EpochId_i$. Each validator is the producer of the corresponding block as in test case 4.

Note, that:

- the epoch block is the last block of the epoch with id $EpochId_{i-1}$, it contains the list of Set_{i-1} *of validators & producers* of validators and producers for the epoch with id $EpochId_{i+1}$;
- the list of validators and producers is predefined (not specified in any blocks) for epochs with identifiers $EpochId_0$ and $EpochId_1$.

It should be noted that when forming the proof $\prod_i = (\pi(H(B_i)), EpochId, \pi(H(B_{i-1})), \pi(EDS(B_i)), \pi(EDS_k(B_{i1})))$ of each block, there are two epoch identifiers $EpochId_{i-1}$ and $EpochId_i$, because block signatures from two consecutive epochs are required.

Implementation:

https://github.com/ZpokenWeb3/zk-light-client-implementation/tree/test_cases/plonky2_test_case4_1

SECTION II: Test case #4.2: recursive proof of the computational integrity of the NEAR blockchain.

The task is to implement a test case of a recursive proof of the computational integrity of the NEAR blockchain.

Unlike the previous example, we generate proofs for all blocks: epochal and ordinary. In fact, we reproduce the block formation protocol in DLT NEAR as a chain of aggregated recursive proofs.

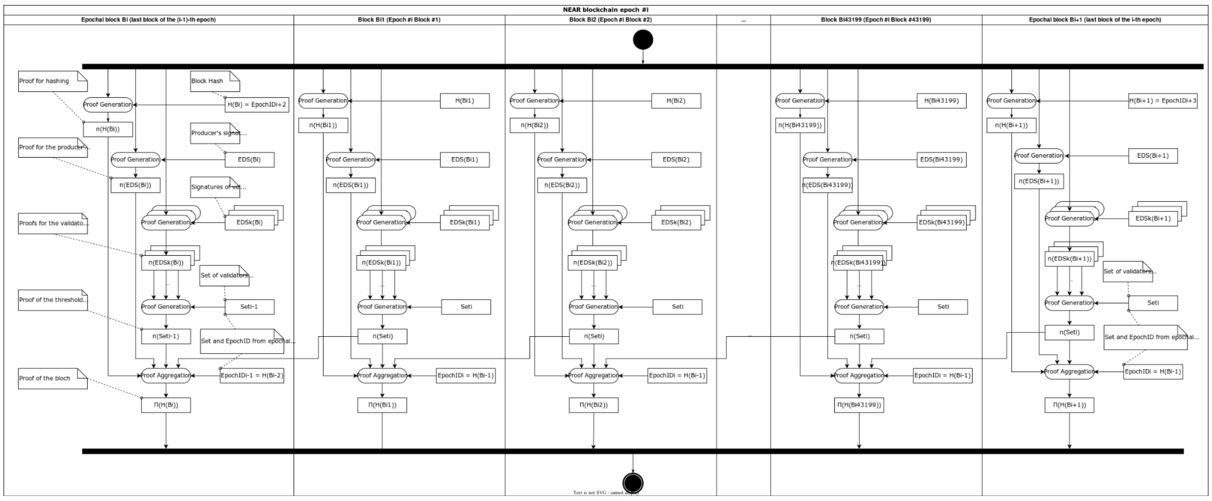

Fig. 2 - Cryptographic chain of blocks

The general scheme for building a chain of recursive proofs includes example 4.1 (fig. 1) with additional proofs for ordinary blocks:

1. Formation of proofs $\pi(EDS_k(B_{ij}))$ of the CI of digital signatures $EDS_k(B_{ij})$ of validators of ordinary blocks. The list of block validators must match the list in Set_{i-1} *of validators & producers* of the penultimate epoch;
2. Proof aggregation. Generation of proof of computational integrity of each block by aggregating:
    a. a proof of the correct hashing $\pi(H(B_{ij}))$ of the ordinary block B_{ij};
    b. a proof of the correct digital signature $\pi(EDS(B_{ij}))$ of the block producer B_{ij};
    c. $EpochId_{i-1}$ is an epoch identifier (constant);
    d. a proof of correct digital signatures $\pi(EDS_k(B_{ij}))$ of block validators B_{ij}.
    e. a final proof $\pi(EDS_k(B_{ij-1}))$ of the previous block B_{ij-1}.

Epoch block proofs additionally contain a proof of the correct hashing $\pi(H(B_{i-2}))$ of the epoch block B_{i-2}, i.e. the correctness of the calculation of the epoch identifier $EpochId_{i+2} = H(B_{i-2})$.

Implementation:

https://github.com/ZpokenWeb3/zk-light-client-implementation/tree/test_cases/plonky2_test_case4_2

SECTION III: Test case #4.2. Time and measurement results

All computations were made on 1,8 GHz Intel Core i5.

The scheme implementation:

https://github.com/ZpokenWeb3/zk-light-client-implementation/tree/test_cases/plonky2_test_case4_2

To get the final time you need to add all the time results.

To get the final proof size see Table 4.

Table 1 — Time and measurement results for proofs for hashes (1,8 GHz Intel Core i5)

№	Time to build a circuit, s	Time to make a proof, s	Proof size, bytes	Verification, s
0_{epoch}	2.0093	4.0053	132576	0.0862
1	4.3883	8.2438	146108	0.0315
2	–	7.5932	146108	0.0601
3_{epoch}	–	8.6319	146108	0.0533
4	–	8.6632	146108	0.0633
5	–	8.0812	146108	0.0412
6_{epoch}	–	7.5740	146108	0.0423
7	–	7.4234	146108	0.0613
8	–	8.4009	146108	0.0517
9_{epoch}	–	8.1406	146108	0.0767

Table 2 — Time and measurement results for proofs for signatures of block producers (1,8 GHz Intel Core i5)

№	Time to build a circuit, s	Time to make a proof, s	Proof size, bytes	Verification, s
0_{epoch}	82.3142	1195.2403	206392	0.0214

1	–	1741.8545	206392	0.0165
2	–	1884.9117	206392	0.0163
3_{epoch}	–	2078.0538	206392	0.0164
4	–	1774.8172	206392	0.0163
5	–	1973.3824	206392	0.0163
6_{epoch}	–	1704.7032	206392	0.0229
7	–	1937.9540	206392	0.0176
8	–	1751.7653	206392	0.0207
9_{epoch}	–	2201.9351	206392	0.0321

Table 3 — Time and measurement results for proofs for signatures of block validators (1,8 GHz Intel Core i5)

№	Time to build a circuit, s	Time to make a proof, s	Proof size, bytes	Verification, s
0_{epoch}	–	1728.5179	206392	0.0266
	–	1504.0904	206392	0.0232
	–	1937.6109	206392	0.0209
1	–	1821.4160	206392	0.0231
	–	1764.7149	206392	0.0193
	–	1974.2745	206392	0.0235
2	–	1870.8016	206392	0.0221
	–	2013.0869	206392	0.1215
	–	1828.4428	206392	0.0304
3_{epoch}	–	1819.0611	206392	0.0351
	–	2036.6267	206392	0.0326

	–	1826.7138	206392	0.0535
4	–	1900.8944	206392	0.0290
	–	1754.4752	206392	0.0278
	–	1950.2008	206392	0.0250
5	–	2049.7232	206392	0.0283
	–	1720.1431	206392	0.0249
	–	1820.8648	206392	0.0227
6_{epoch}	–	1811.6764	206392	0.0302
	–	1958.8086	206392	0.0255
	–	1975.9973	206392	0.0268
7	–	1847.8790	206392	0.0265
	–	1747.4057	206392	0.0231
	–	1862.5951	206392	0.0385
8	–	2024.4520	206392	0.1972
	–	1849.4671	206392	0.0245
	–	2067.1717	206392	0.0312
9_{epoch}	–	–	–	–

Table 4 — Time and measurement results for final proofs, i. e. aggregation with the proof of the previous block and aggregation with the proofs for hash of epoch blocks for epoch blocks only (1,8 GHz Intel Core i5)

№	Block type	Time to build a circuit, s	Time to make a proof, s	Proof size, bytes	Verification, s
0_{epoch}	epoch block	–	–	–	–
	epoch block	–	–	–	–

	previous block	5.3308783	10.2792	146588	0.0894
1	epoch block	–	–	–	–
	epoch block	–	–	–	–
	previous block	4.7095766	9.9709	146348	0.1333
2	epoch block	–	–	–	–
	epoch block	–	–	–	–
	previous block	5.733359	9.3847	146348	0.0905
3_{epoch}	epoch block	2.9163713	4.8917	–	0.0624
	epoch block	–	–	–	–
	previous block	5.314643	10.0085	146348	0.0536
4	epoch block	–	–	–	–
	epoch block	–	–	–	–
	previous block	5.017295	9.4335	146348	0.0731
5	epoch block	–	–	–	–
	epoch block	–	–	–	–
	previous block	4.711299	8.7290	146348	0.1351
6_{epoch}	epoch block	4.780585	8.6937	–	0.3104
	epoch block	2.1493838	4.1571	–	0.0818
	previous block	4.43709	7.9627	146348	0.1044
7	epoch block	–	–	–	–
	epoch block	–	–	–	–
	previous block	5.1883388	9.6124	146348	0.0850
8	epoch block	–	–	–	–

	epoch block	–	–	–	–
	previous block	4.762961	8.9360	146348	0.3070
9_{epoch}	epoch block	4.7457557	10.1135	–	0.3247
	epoch block	5.838455	9.2063	–	0.2690
	previous block	5.076543	8.7094	146348	0.0118

SECTION IV: Integration of NEAR block structure

This report presents an analysis of the computational integrity for block headers based on the implementation detailed in the article linked below and the results obtained for 9 consecutive blocks.
Source: Section 2 and 3 (Recursion method only):
https://github.com/ZpokenWeb3/zk-light-client-implementation/blob/main/Report%2031.01.2023%20ZKP%20for%20Near%20Blocks%20Proving.pdf
The focus is on build times, proof times, and verification times.
The ultimate goal is to extrapolate this implementation for epoch blocks to prove up to 2050 blocks.

```
pub struct BlockV2 {
    pub header: BlockHeader,
    pub chunks:
Vec<ShardChunkHeader>,
    pub challenges:
Challenges,
    // Data to confirm the
correctness of randomness
beacon output
    pub vrf_value:
near_crypto::vrf::Value,
    pub vrf_proof:
near_crypto::vrf::Proof,
}
```

```
pub struct BlockHeaderV3 {
    pub prev_hash: CryptoHash,

    /// Inner part of the
block header that gets hashed,
split into two parts, one that
is sent
    ///  to light clients, and
the rest
    pub inner_lite:
BlockHeaderInnerLite,
    pub inner_rest:
BlockHeaderInnerRestV3,

    /// Signature of the block
producer.
    pub signature: Signature,

    /// Cached value of hash
for this block.
    #[borsh_skip]
    pub hash: CryptoHash,
}
```

```
pub struct
BlockHeaderInnerLite {
    /// Height of this block.
    pub height: BlockHeight,
    /// Epoch start hash of
this block's epoch.
    /// Used for retrieving
```

```
pub struct
BlockHeaderInnerRest {
    /// Root hash of the chunk
receipts in the given block.
    pub chunk_receipts_root:
MerkleHash,
    /// Root hash of the chunk
validator information
    pub epoch_id: EpochId,
    pub next_epoch_id:
EpochId,
    /// Root hash of the state
at the previous block.
    pub prev_state_root:
MerkleHash,
    /// Root of the outcomes
of transactions and receipts.
    pub outcome_root:
MerkleHash,
    /// Timestamp at which the
block was built (number of
non-leap-nanoseconds since
January 1, 1970 0:00:00 UTC).
    pub timestamp: u64,
    /// Hash of the next epoch
block producers set
    pub next_bp_hash:
CryptoHash,
    /// Merkle root of block
hashes up to the current
block.
    pub block_merkle_root:
CryptoHash,
}
```

```
headers in the given block.
    pub chunk_headers_root:
MerkleHash,
    /// Root hash of the chunk
transactions in the given
block.
    pub chunk_tx_root:
MerkleHash,
    /// Number of chunks
included into the block.
    pub chunks_included: u64,
    /// Root hash of the
challenges in the given block.
    pub challenges_root:
MerkleHash,
    /// The output of the
randomness beacon
    pub random_value:
CryptoHash,
    /// Validator proposals.
    pub validator_proposals:
Vec<ValidatorStakeV1>,
    /// Mask for new chunks
included in the block
    pub chunk_mask: Vec<bool>,
    /// Gas price. Same for
all chunks
    pub gas_price: Balance,
    /// Total supply of tokens
in the system
    pub total_supply: Balance,
    /// List of challenges
result from previous block.
    pub challenges_result:
ChallengesResult,

    /// Last block that has
full BFT finality
    pub last_final_block:
CryptoHash,
    /// Last block that has
doomslug finality
    pub last_ds_final_block:
CryptoHash,

    /// All the approvals
included in this block
```

	``` pub approvals: Vec<Option<Signature>>, /// Latest protocol version that this block producer has. pub latest_protocol_version: ProtocolVersion, }```

```
    pub approvals:
Vec<Option<Signature>>,

    /// Latest protocol
version that this block
producer has.
    pub
latest_protocol_version:
ProtocolVersion,
}
```

Table 5 — Results on 9 consecutive blocks (1,900GHz AMD Ryzen 7 5800U (16))

№	Time to build, s	Time to prove, s	Verification, s	∑, s
0	2.5633	4.6440	0.0070	7.2143
1	2.0640	5.1424	0.0076	14.4283
2	2.4430	4.8360	0.0071	21.7144
3	2.0789	4.3056	0.0076	28.1065
4	2.1833	4.4122	0.0069	34.7089
5	2.1611	4.4126	0.0071	41.2897
6	2.1334	4.1623	0.0071	47.5925
7	2.1054	4.2416	0.0070	53.9465
8	2.1219	4.1295	0.0072	60.2051
9	2.2060	4.6440	0.0070	7.2143
Avg.	2.2060	4.4762	0.0072	–