Cairo Placeholder Verification

Technical Reference

Ilia Shirobokov
i.shirobokov@nil.foundation
=nil; Crypto3 (https://crypto3.nil.foundation)

Mikhail Komarov
nemo@nil.foundation
=nil; Foundation (https://nil.foundation)

June 22, 2022

Contents

Chapter 1

Introduction

This document is a technical reference to the Solana's state proof verification with Cairo project.

1.1 Overview

The project's purpose is to provide StarkNet users with reliable Solana's cluster state and necessary transactions proof source.

The project UX consists of several steps:

1. Retrieve Solana's state proof from `=nil;`.
2. Generate a STARK proof for it.
3. Verify the proof with in-StarkNet Cairo-based verifier.

Such a UX defines projects parts:

1. Solana's state proof retriever.
2. Auxiliary STARK proof generator.
3. Cairo-based Placeholder Verifier.

Each of these parts will be considered independently.

Chapter 2

State Proof Generator

The basic idea is to wrap the original proofs provided by DBMS replication protocol trustless I/O extension-based cluster to the StarkNet-compatible one.

Using these proofs, StarkNet/StarkEx (and Cairo in general) users will become capable to rely on third-party clustered databases data in a trustless manner by receiving `=nil;` DBMS query data, and a STARK proof, proving its structure and contents. This would provide StarkNet/StarkEx users/developers with bridging capabilities with any protocol `=nil;` DBMS supports (e.g. Bitcoin, Solana, others).

Remind that there are two types of proofs in `=nil;` DBMS Cluster:

- State Proofs provide compressed state updates from the tracked clusters. The circuits of these proofs are fixed and known in advance.
- Query Proofs show that some data is contained in the compressed state. The query proofs' details are not known in advance. However, they can be represented by relatively simple circuits that can be generated in runtime.

Some additional setup is required to integrade a tracked cluster into StarkNet via `=nil;` DBMS Cluster

1. Prepare Placeholder verification circuit for state proofs in PLONK-arithmetization.
2. Translate this circuit into AIR StarkNet-compatible representation.

The wrapping mechanism slightly change intercluster transcation **Read** operation[1].

Algorithm 1 Specialized Read Operation

```
Input: from, query, need_proof
1. Generate circuit $C$ from query.
2. Query data response required by query from database from.
3. If need_proof $= 1$ (otherwise $\pi_{\texttt{state}} = \pi_{\texttt{resp}} = 0$):
    3.1 If there is not anchor for the state, generate Placeholder proof $\pi'_{\texttt{state}}$ of the consistent state
        state of from
    3.2 Generate STARK proof $\pi_{\texttt{resp}}$ of $C$.
    3.3 Generate STARK proof $\pi_{\texttt{state}}$ that verifies $\pi'_{\texttt{state}}$.
4. Write $(\pi_{\texttt{state}}, \pi_{\texttt{resp}}, \texttt{response})$ to $\mathcal{M}$. Validators of $\mathcal{M}$ can verify response using either $\pi_{\texttt{state}}, \pi_{\texttt{resp}}$
   or the tracked state of from.
```

2.1 Circuit Conversion

We can move arithmetic constraints from PLONK into AIR representation straightforwardly. However, STARKs do not provide separate Permutation and Lookup arguments. It means that we need to convert copy- and lookup-constraints into basic AIR constraints.

[1]For details on intercluster transaction process, see https://dbms.nil.foundation/io.pdf

1. Split gates into separate constraints.
2. Represent each constraint as AIR constraint on the rows with the rows where selector values are equal to 1.
3. Represent copy-constraint as AIR constraints with the corresponding shift.
4. Represent lookup-constraints other separate lookup-argument as AIR constraints with the inner lookup table.

Chapter 3

In-EVM State Proof Verifier

This introduces a description for Solana's state proof Cairo-based verifier.
Crucial components which define this part design are:

1. Verification architecture description.
2. Verification logic API reference.
3. Input data structures description.

3.1 Verification Logic Architecture

Verification contains the following steps:

1. Get input: proof π and new state S.
2. Verify placeholder proof π (see placeholder verification below).
3. Update the last confirmed Solana state with S (see 3.1.1).

Placeholder verificaton part contains the following components:

1. **Proof Deserialization:** Handles the input data processing (marshalling/demarshalling) mechanisms.

   These mechanisms are defined within the `*_marshalling`-postfixed files.

   - `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/basic_marshalling.sol`
   - `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/basic_marshalling_calldata.sol`

2. **Proof Verification:** Includes a verification of the hash-based commitment scheme and the proof itself.

   The verification itself is defined within the directory `components`, each of which[1] defines a set of gates relevant to particular component. Verification algorithm contains:

   - Transcript (Fiat-Shamir transfomration to non-interactive protocol)
     `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/cryptography/transcript.sol`
   - Permutation argument:
     `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/placeholder/permutation_argument.sol`
   - Gate Argument depends on the circuit definition and is unique for each circuit. Example:
     - Circuit description:
       `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/components/poseidon_split_gen.sol.txt`

[1] For instance, `https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src//placeholder/verifier_non_native_field_add_component.sol`

    - Generated gate argument:
      https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/components/non_native_field_add_gen.sol
- Commitment Scheme verification
  https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/commitments

3.1.1 State Proof Sequence Maintenance

To verify the validator set within the state proof submitted is derived from original Solana's genesis data, it is supposed to maintain validator's set state proofs sequence on StarkNet side in a data structure as follows.

Let B_{n_1} be the last state confirmed on Ethereum. Let us say some prover wants to confirm a new B_{n_2} state. Denote by H_B the hash of a state B. So a Merkle Tree T_{n_1,n_2} from the set $\{H_{B_{n_1}}, ..., H_{B_{n_2}}\}$

The state proof sequence correctness statement contains (but not bounded by) the following points:

Algorithm 2 Proving Statement

```
1. Show that the validator set is correct.
2. Show that the $B_{n_1}$ corresponds to the last confirmed state on StarkNet.
3. for $i$ from the interval $[n_1 + 1, n_2 - 1]$:
    3.1 Show that $B_i$ contains $H_{B_{i-1}}$ as a hash of the previous state.
4. for $i$ from the interval $[n_2, n_2 + 32]$:
    4.1 Show that $B_i$ contains $H_{B_{i-1}}$ as a hash of the previous state.
    4.2 Show that there are enough valid signatures from the current validator set for $B_i$.
5. Build a Merkle Tree $T_{n_1,n_2}$ from the set $\{H_{B_{n_1}}, ..., H_{B_{n_2}}\}$.
```

T_{n_1,n_2} allows to provide a successful transaction from $\{B_{n_1}, ..., B_{n_2}\}$ to the StarkNet-based proof verificator later.

3.2 Verification Logic API Reference

Every call to Placeholder public API verification function eventually leads to a call of internal verification function for chosen circuit (for example, https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/placeholder/verifier_unified_addition_component.cairo#L66). These internal verification functions should be supplied with proof byteblob and initialized verification parameters.

For now there is test public API which execute basic logic consisting of:

1. parsing proof byteblob
2. verification parameters initialization
3. circuit specific internal verification function calling
4. verification result returning

Example of test public API function declaration intended for verification of unified addition circuit (see https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/placeholder/test/public_api_placeholder_unified_addition_component.sol#L34):

```
function verify(
    bytes calldata blob,
    uint256[] calldata init_params,
    int256[][] calldata columns_rotations
) public {...}
```

More details regarding public API input data structure see in the next section.
Other existing test public API functions could be found here:

- https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/placeholder/test/public_api_placeholder_non_native_field_add_component.sol#L34
- https://github.com/NilFoundation/cairo-placeholder-verification/blob/master/src/placeholder/test/public_api_placeholder_variable_base_scalar_mul_component.sol#L34

3.3 Input Data Structures

All input data divided into two parts:

1. Placeholder proof byteblob itself;
2. Verification parameters used to verify proof.

3.3.1 Placeholder Proof Structure

Placeholder proof consists of different fields and some of them are of complex structure types, which will be described in top-down order.

So, the first one Placeholder proof has the following structure, which is described in pseudocode:

```
struct PlaceholderProof {
    witness_commitment: vector<uint8>
    v_perm_commitment: vector<uint8>
    input_perm_commitment: vector<uint8>
    value_perm_commitment: vector<uint8>
    v_l_perm_commitment: vector<uint8>
    T_commitment: vector<uint8>
    challenge: uint256
    lagrange_0: uint256
    witness: LPCProof
    permutation: LPCProof
    quotient: LPCProof
    lookups: vector<LPCProof>
    id_permutation: LPCProof
    sigma_permutation: LPCProof
    public_input: LPCProof
    constant: LPCProof
    selector: LPCProof
    special_selectors: LPCProof
}
```

In turn proof of LPC algorithm has the following structure:

```
struct LPCProof {
    T_root: vector<uint8>
    z: vector<vector<uint256>>
    fri_proofs: vector<FRIProof>
}
```

The next one description is for structure of FRI algorithm proof:

```
struct FRIProof {
    final_polynomials: vector<vector<uint256>>
    round_proofs: vector<FRIRoundProof>
}
```

One of the components of the FRI algorithm proof is so called round FRI proof, which has the following structure:

```
struct FRIRoundProof {
    colinear_value: vector<uint256>
    T_root: vector<uint256>
    colinear_path: MerkleProof
    p: vector<MerkleProof>
}
```

The next important component is the merkle tree proof of the following structure:

```
struct MerkleProof {
    leaf_index: uint64
    root: vector<uint8>
    path: vector<MerkleProofLayer>
}

struct MerkleProofLayer {
    layer: vector<MerkleProofLayerElement>
}
```

In the simplest and used case of the merkle tree with arity 2 layer consists of only one element:

```
struct MerkleProofLayerElement {
    position: uint64
    hash: vector<uint8>
}
```

It is important to note that before sending Placeholder proof to EVM for verification it should be serialized into byteblob format, which is done using corresponding marshalling module (https://github.com/NilFoundation/crypto3-zk-marshalling/blob/01b531550a99232586e17c1e383e4693a4ddc924/include/nil/crypto3/marshalling/zk/types/placeholder/proof.hpp).

3.3.2 Verification Parameters

Verification parameters are used to parametrize Placeholder algorithm depending on chosen security parameters and specific circuit for which proof was created.

Following parameters are required to complete Placeholder verification procedure in-EVM:

```
uint256_t modulus; // modulus of chosen prime field
uint256_t r; // parameter of FRI algorithm
uint256_t max_degree; // parameter of FRI algorithm
uint256_t lambda; // parameter of LPC algorithm
uint256_t rows_amount; // parameter defined by chosen circuit
uint256_t omega; // parameter defined by chosen circuit
uint256_t max_leaf_size; // parameter dependent on specific instance of Placeholder algorithm,
std::vector<uint256_t> domains_generators; // parameter defined by chosen circuit
std::vector<uint256_t> q_polynomial; // parameter of Placeholder algorithm
std::vector<std::vector<uint256_t>> columns_rotations; //parameter defined by chosen circuit
```

Bibliography

1. Kattis A., Panarin K., Vlasov A. RedShift: Transparent SNARKs from List Polynomial Commitment IOPs. Cryptology ePrint Archive, Report 2019/1400. 2019. https://ia.cr/2019/1400.

2. Gabizon A., Williamson Z. J., Ciobotaru O. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. 2019. https://ia.cr/2019/953.

3. Fast Reed-Solomon interactive oracle proofs of proximity / E. Ben-Sasson, I. Bentov, Y. Horesh et al. // 45th international colloquium on automata, languages, and programming (icalp 2018) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

4. Gabizon A., Williamson Z. J. Proposal: The Turbo-PLONK program syntax for specifying SNARK programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.

5. Gabizon A., Williamson Z. J. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315. 2020. https://ia.cr/2020/315.

6. PLONKish Arithmetization - The halo2 book. https://zcash.github.io/halo2/concepts/arithmetization.html.

7. Lookup argument - The halo2 book. https://zcash.github.io/halo2/design/proving-system/lookup.html.

8. Chiesa A., Ojha D., Spooner N. Fractal: Post-Quantum and Transparent Recursive Proofs from Holography. Cryptology ePrint Archive, Report 2019/1076. 2019. https://ia.cr/2019/1076.