

# Smart Contract Audit Report

## Conducted by CryptoExperts

As part of our due process, we retained CryptoExperts to review the design document and related source code of the EVM STARK Verifier. We chose to work with CryptoExperts based on warm recommendations and our interaction with them.

We are happy to share the key findings below, followed by the full report.

## Vulnerability Severity Classification

The current version of the report is an update of our original report after counterauditing modifications from StarkWare.

Each observation is appended with a status:  
Resolved (✓), Partially resolved (≈), Unresolved (✗).

Most of our recommendations have been addressed. Unresolved or partially resolved issues are related to documentation or coding practices which are of minor importance.

Category	Number of findings	✓	≈	✗
High risk	0	-	-	-
Medium risk	1	1	-	-
Low risk	2	2	-	-
Coding Practices	17	8	4	5
Documentation	7	3	1	3
Total	27	14	5	8

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Overview of the code . . . . .	4
1.2	Methodology and summary of findings . . . . .	5
<b>2</b>	<b>Specifications of APIs and data formats</b>	<b>7</b>
2.1	Notions and notations . . . . .	7
2.2	Merkle commitment scheme . . . . .	10
2.2.1	Merkle queue . . . . .	10
2.2.2	Trace commitments . . . . .	11
2.2.3	Layer polynomial commitments . . . . .	11
2.3	FRI protocol . . . . .	11
2.3.1	Scaling . . . . .	11
2.3.2	FRI context . . . . .	12
2.3.3	FRI queue . . . . .	13
2.4	STARK verifier . . . . .	14
2.4.1	Parameters . . . . .	14
2.4.2	Proof format . . . . .	14
2.4.3	Verifier context . . . . .	15
2.5	External dependencies . . . . .	16
2.5.1	Memory mapping contract . . . . .	16
2.5.2	OODS contract . . . . .	17
2.5.3	Derived STARK verifier contract . . . . .	17
<b>3</b>	<b>Review and specific observations</b>	<b>19</b>
3.1	Arithmetic primitives . . . . .	19
3.1.1	Contract PrimeFieldElement0 . . . . .	19
3.1.2	Contract HornerEvaluator . . . . .	20
3.2	Verifier channel . . . . .	21
3.2.1	Contract Prng . . . . .	21
3.2.2	Contract VerifierChannel . . . . .	22
3.3	Merkle trees . . . . .	24
3.3.1	Contract IMerkleVerifier . . . . .	24
3.3.2	Contract MerkleVerifier . . . . .	25
3.3.3	Contract MerkleStatementContract . . . . .	26
3.3.4	Contract MerkleStatementVerifier . . . . .	27
3.4	FRI protocol . . . . .	27
3.4.1	Contract FriLayer . . . . .	27
3.4.2	Contract Fri . . . . .	31
3.4.3	Contract FriStatementContract . . . . .	31
3.4.4	Contract FriStatementVerifier . . . . .	32
3.5	STARK verifier . . . . .	34

3.5.1 Contract StarkVerifier . . . . .	34
<b>4 General observations</b>	<b>38</b>
<b>A Formal checking of doXFriSteps</b>	<b>44</b>

# 1 Introduction

STARKWARE is a company developing scalability and privacy technologies for blockchain applications. In particular, STARKWARE develops a STARK-powered level-2 scalability engine which uses cryptographic proofs to attest to the validity of a batch of transactions. In this context, STARKWARE was seeking for a code review of their STARK verifier in order to verify its correct implementation and its compliance to the protocol specification. The code of the STARK verifier to be reviewed is written in Solidity language and composed of 14 smart contract source files.

Upon business agreement, with STARKWARE, CRYPTOEXPERTS conducted the audit of the code in November and December 2021. The service consisted in a study of the protocol specification and a review of the code by two engineers, experts in cryptography. The present report contains the results of this audit.

We first give an overview of the code (Section 1.1) and a summary of the audit methodology and findings (Section 1.2). Then we specify some APIs and data formats which were necessary for the good understanding of the code (Section 2). Our review of the different smart contracts is depicted in Section 3, including for each of them a summary of the functionality and a list of observations and recommendations. General observations and recommendations (which affect several contracts) are finally provided in Section 4.

## 1.1 Overview of the code

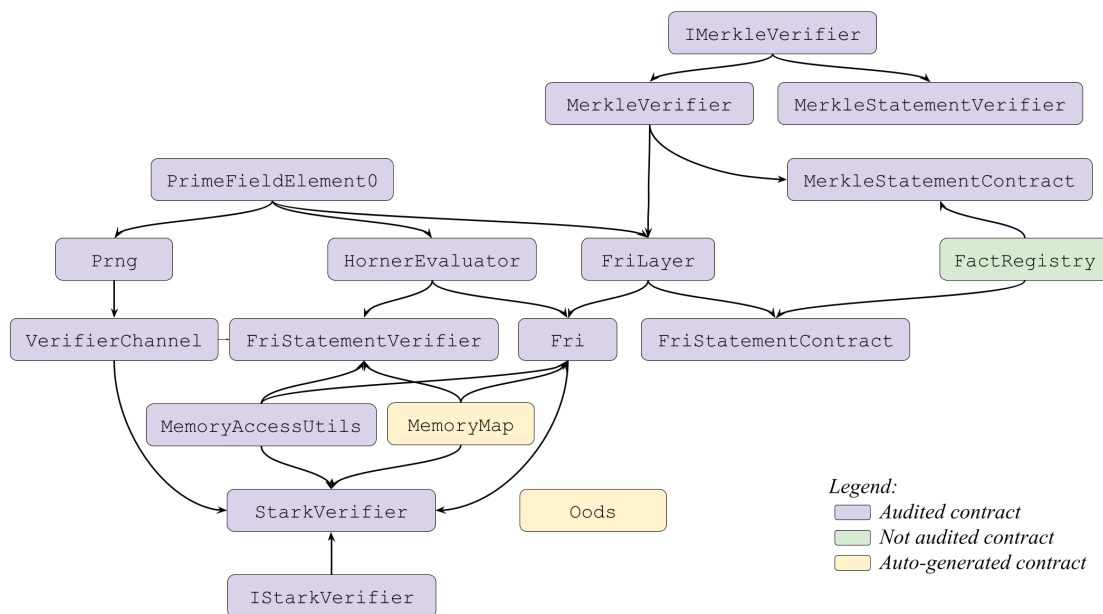


Figure 1: The inheritance relations between the smart contracts of the STARK verifier.

The audited code implements the STARK verifier of STARKWARE's scalability engine. It is composed of the 14 source files from the following GitHub repository:

<https://github.com/starkware-libs/starkex-contracts/tree/0efa9ce324b04226de5dcd7a0139b109bca8f074/evm-verifier/solidity/contracts>

The audited version of the code corresponds to the commit "StarkEx v4.0", from October 14th, 2021.

The different smart contracts are represented in Figure 1 with their inheritance relations. The contract in green is out of the scope of the present audit. The contracts in yellow are auto-generated and out of the scope of the audit.






## 1.2 Methodology and summary of findings

The main goal of this audit was to validate the soundness of the reviewed implementation of the STARK verifier. More precisely, this audit aims

- to confirm that the implemented verification process is compliant to the specification of the STARK verifier,
- to check the absence of flaw in the implementation which would allow an adversary to forge a valid proof for an invalid statement (with less effort than the target security level).

The audit methodology consisted in an in-depth review of the code by two different persons (engineers, junior and senior experts in cryptography), confronting our understanding of the code and keeping track of our observations. We did not rely on automatic tools or re-implementation (with one exception depicted in Section 3.4.1).

Our observations are categorized as follows:

- Observations that may impact the soundness of the verifier, rated as
  - high risk (flagged ) ,
  - medium risk (flagged ) ,
  - low risk (flagged ) .
- Observations related to coding practices and implementation choices (flagged ) .
  - These observations do not translate into a direct risk on the soundness of the verifier but addressing them would make the code clearer, more efficient and/or less prone to errors.
- Observations related to documentation, comments, variable naming (flagged ) .
  - These observations do not translate into a direct risk on the soundness of the verifier but addressing them would facilitate the understanding of the code by third parties (users, developers, auditors).

Each observation comes with an associated recommendation to fix or improve the underlying issue.

*Summary of findings.* As a general remark, we were provided with a clear set of documentations about the STARK protocol [4, 5, 8, 9] but sometimes missed detailed specifications for the reviewed implementation. To reach a global and confident understanding of the implementation, we wrote down the missing specifications according to our understanding of the code, which we include in the report (see Section 2).

Our findings are summarized in the table below. Besides observations related to coding practices and documentation, we only made three observations with potential impact on the verifier soundness, rated low for two of them (contract `VerifierChannel`) and medium for one of them (contract `Prng`). As explained in our recommendations, these issues can easily be addressed. Besides these points, and up to the limitations inherent to human code review, we conclude that the code is a sound implementation of the STARK verifier.

The current version of the report is an update of our original report after counter-auditing modifications from STARKWARE. Each observation is appended with a status: Resolved (✓), Partially resolved (≈), Unresolved (✗). Most of our recommendations have been addressed. Unresolved or partially resolved issues are related to documentation or coding practices which are of minor importance.

Category	Number of findings	✓	≈	✗
● High risk	0	-	-	-
● Medium risk	1	1	-	-
● Low risk	2	2	-	-
● Coding practices	17	8	4	5
■ Documentation	7	3	1	3
<b>Total</b>	<b>27</b>	14	5	8

## 2 Specifications of APIs and data formats

### 2.1 Notions and notations

We recall hereafter the notions and notations relevant to this report. We use the notations and terminology introduced in the ethSTARK documentation [10] (as well as a few additional notations).

- The *IOP Prover* and the *IOP Verifier* refer to the two parties of the interactive protocol (*i.e.* before the transformation to a non-interactive protocol).
- The *Execution Trace* uses  $W$  registers and has length  $N$ , where  $N$  is a power of two.
- The values in the trace cells are elements of the finite field  $\mathbb{F}_p$  with  $p$  a prime.
- The *Trace Evaluation Domain* is defined as a multiplicative subgroup  $\langle g \rangle$  of  $\mathbb{F}_p^\times$  of size  $N$ .
- Each trace column is interpreted as the  $N$  point-wise evaluations of a polynomial of degree smaller than  $N$  over the trace evaluation domain. These polynomials are referred to as the *Trace Column polynomials* and are denoted by  $f_0, \dots, f_{W-1}$ .
- An *Algebraic Intermediate Representation (AIR) Polynomial Constraint* on the trace (represented as a rational function) is denoted  $C_j$  and is of degree  $D_j$ . We denote  $D$  the smallest power of two which is greater than all the constraint degrees.
- The *Composition Polynomial* is denoted by  $h(x)$  and takes the form

$$h(x) = \sum_{j=1}^k C_j(x)(\alpha_j x^{D-D_j-1} + \beta_j) \quad (1)$$

where  $k$  is the number of constraints. Its degree is  $D-1$ . This polynomial can decompose into  $M_2$  polynomials  $h_0, \dots, h_{M_2-1}$  of degree  $N$  (called *Composition Polynomial Trace*) such that

$$h(x) = \sum_{i=0}^{M_2-1} x^i h_i(x^{M_2}) . \quad (2)$$

- To check the consistency between the execution trace and the composition polynomial trace,  $h$  is evaluated using the two above expressions in a single random point denoted  $z$  which is called the *OODS point*.
- To evaluate  $h$  in the OODS point  $z$  using (1), we need a set of values of the form  $f_j(zg^s)$  which are involved in the expressions of the  $\{C_j(z)\}_j$ . This set of values is called the *mask*. It is denoted  $\{y_\ell\}_\ell$  and its size is denoted  $M_1$ .
- To evaluate  $h$  in the OODS point  $z$  using (2), we need the values  $h_i(x^{M_2})$  for  $0 \leq i < M_2$ . Those values are denoted  $\hat{y}_i := h_i(z^{M_2})$ .

- The elements of  $\{y_\ell\}_\ell \cup \{\hat{y}_i\}_i$  are called the *OODS Values*.
- The *DEEP Composition Polynomial*, denoted  $p_0(x)$  is defined as:

$$p_0(x) = \sum_{\ell=0}^{M_1-1} \gamma_\ell \cdot \frac{f_{j_\ell}(x) - y_\ell}{x - zg^{s_\ell}} + \sum_{i=0}^{M_2-1} \gamma_{M_1+i} \cdot \frac{h_i(x) - \hat{y}_i}{x - z^{M_2}}$$

where  $\{\gamma_0, \dots, \gamma_{M_1+M_2-1}\}$  are random coefficients sampled by the IOP verifier, called *OODS Coefficients*.

- To achieve a secure protocol, the polynomials  $f_0, \dots, f_{W-1}$  and  $h_0, \dots, h_{M_2-1}$  are evaluated over a domain  $L_0$ , larger than and disjoint from the trace evaluation domain, which we call the *evaluation domain*. We refer to this evaluation as *the trace Low Degree Extension (LDE)* and the ratio  $|L_0|/N$  (ratio between the size of the evaluation domain and the size of the trace evaluation domain) is further referred to as the *blowup factor*, denoted  $\beta$ . In practice,  $L_0$  is a non-unit coset  $c_{\text{LDE}} \cdot \langle g_{\text{LDE}} \rangle$  of the multiplicative subgroup  $\langle g_{\text{LDE}} \rangle \subseteq \mathbb{F}_p^\times$  of size  $\beta N$ .
- The FRI protocol is composed of different *layers*. We denote  $K$  the number of FRI layers. Except for the last one, each layer performs a number of *FRI steps*. For the  $i$ th layer, the number of steps is denoted  $\ell_i$ , which is stored in a list denoted `fri_step_list[]`. Namely,

$$\ell_i := \text{fri\_step\_list}[i] .$$

We further denote by  $s_i$  the total number of steps from layers 1 to  $i$ , that is

$$s_i := \sum_{j=1}^i \ell_j .$$

- At the  $i$ th layer, the IOP prover starts with a polynomial  $p_{i-1}$ . She gets a random challenge  $\zeta_{i-1} \in \mathbb{F}_p$  from the verifier and computes the new polynomial  $p_i$  as

$$p_i(x) = \sum_{j=0}^{2^{\ell_i}-1} \zeta_{i-1}^j \cdot p_{i-1}^{[j]}(x) \quad (3)$$

where the  $\{p_{i-1}^{[j]}\}_j$  are the polynomials defined such that

$$p_{i-1}(x) = \sum_{j=0}^{2^{\ell_i}-1} x^j \cdot p_{i-1}^{[j]}(x^{2^{\ell_i}}) .$$

- Let  $L_i$  the evaluation domain of  $p_i$ . Then, for every  $i \geq 1$ , we have

$$L_i := \left\{ x^{2^{\ell_i}}, x \in L_{i-1} \right\}$$

which by definition of  $L_0$  implies

$$L_i = c_{\text{LDE}}^{2^{s_i}} \cdot \langle g_{\text{LDE}}^{2^{s_i}} \rangle .$$



- The polynomial  $p_0$  at the input of the FRI protocol is the DEEP composition polynomial and we have, for  $i \geq 0$ ,

$$\deg p_i = \frac{N}{2^{s_i}}$$

where  $s_i$  is defined as above.

- All the coefficients of the final polynomial  $p_{K-1}$  are sent to the IOP Verifier.
- Once the polynomials  $\{p_i\}_i$  committed, the FRI protocol evaluates these polynomials on a set of evaluation points, called *queries*. The set of queries for the polynomial  $p_i$  is denoted  $\mathcal{Q}_i \subset L_i$ , and for every  $i \geq 1$ , we have

$$\mathcal{Q}_i := \left\{ x^{2^{\ell_i}}, x \in \mathcal{Q}_{i-1} \right\}$$

where  $\mathcal{Q}_0 \subseteq L_0$  is the set of random queries sampled by the IOP verifier at the beginning of the query phase of the FRI protocol.

- Each element  $x$  of  $L_i$  has an index  $\text{index}_i(x)$  defined such as

$$x = (c_{\text{LDE}}^{2^{s_i}}) \cdot (g_{\text{LDE}}^{2^{s_i}})^e \iff \text{index}_i(x) = \text{bit-reverse}_{\log_2 |L_i|}(e)$$

where  $\text{bit-reverse}_{\log_2 |L_i|}(e)$  stands for the bit reverse of the exponent  $e$  on  $\log_2 |L_i| = \log_2(\beta \cdot N) - s_i$  bits. We further denote

$$\overline{\text{index}_i}(x) := \text{index}_i(x) + |L_i|.$$

- We introduce the *scaled* versions of several FRI notions. For all  $0 \leq i \leq K-1$ ,
  - the *scaled* layer polynomial  $p'_i(x)$  is defined as  $p'_i(x) := p_i(c_{\text{LDE}}^{2^{s_i}} \cdot x)$ ;
  - the scaled layers polynomials  $p'_{i-1}$  and  $p'_i$  verify a similar relation than in the Equation 3 but for the *scaled* challenges defined as

$$\zeta'_{i-1} := \left( c_{\text{LDE}}^{2^{s_{i-1}}} \right)^{-1} \cdot \zeta_{i-1} ;$$

- the *scaled* evaluation domain  $L'_i$  and the *scaled* set  $\mathcal{Q}'_i$  of queries for the polynomial  $p'_i$  are defined as

$$L'_i := (c_{\text{LDE}}^{2^{s_i}})^{-1} \cdot L_i = \langle g_{\text{LDE}}^{2^{s_i}} \rangle \quad \text{and} \quad \mathcal{Q}'_i := \{ (c_{\text{LDE}}^{2^{s_i}})^{-1} \cdot v ; v \in \mathcal{Q}_i \}.$$

*Remark.* In the above notations, the following are not from [10] but introduced for the purpose of the present report:  $s_i$ ,  $\ell_i$ ,  $K$ ,  $\text{index}_i(x)$ ,  $c_{\text{LDE}}$ ,  $g_{\text{LDE}}$ ,  $\mathcal{Q}_i$  and all the scaled notions.

## 2.2 Merkle commitment scheme

All the commitments in the protocol are realized thanks to Merkle trees. This structure enables to commit several inputs as leaves of the tree to get a single commitment value (the root of the tree). Then, it is possible to show the consistence of a small subset of the revealed inputs with the committed root without having to communicate all the other inputs. The principle is to reveal the sibling paths of the revealed inputs in the Merkle tree.

In the STARK verifier, only *complete* Merkle trees are used, *i.e.* Merkle trees with  $2^n$  leaves for some  $n \in \mathbb{N}$ . Moreover, only the verification primitive is implemented: the STARK verifier does not need to commit but only to decommit data.

### 2.2.1 Merkle queue

The implementation uses a structure called a *Merkle queue*. It is a *circular* queue containing some nodes of the tree (leaves and internal nodes). At the beginning of the verification the queue only contains the decommitted leaves. Along the verification, the queue is fed with the encountered nodes on the Merkle paths to the decommitted leaves (either computed from previous queue elements or fetched from the proof).

Each node is represented by a pair composed of

- The index of the node, which is stored on 32 bytes. The nodes of depth  $i$  have indexes  $2^i, \dots, 2^{i+1} - 1$ . Thus the root has index “1” and the leaf indexes are  $\{2^n, \dots, 2^{n+1} - 1\}$ .
- The label of the node in the Merkle tree (*i.e.* the hash of the child nodes for a internal node or a hash digest of one committed value), which is stored on 32 bytes.

By definition of a Merkle tree, the label corresponding to index  $i$  is defined as

$$\text{label}_i := \text{Hash}(\text{label}_{2i} \parallel \text{label}_{2i+1}) ,$$

for internal nodes and

$$\text{label}_{2^n+i} := \begin{cases} \text{Hash}(\text{input}_i) & \text{if } \text{input}_i \text{ is bigger than 256 bits} \\ \text{input}_i & \text{otherwise} \end{cases}$$

for leaves, where  $\text{input}_0, \dots, \text{input}_{2^n-1}$  are the  $2^n$  inputs of the tree. The used hash function  $\text{Hash}(\cdot)$  is KECCACK-256 truncated to the 160 most significant bits.

The nodes in the Merkle queue must respect the following order:

- All the present nodes of the same depth must be contiguous in the queue and must have their indices in the increasing order;
- A node of depth  $i$  must be before a node of depth  $j$  if  $i > j$ .

### 2.2.2 Trace commitments

In the STARK protocol, two Merkle trees are used to commit traces: the *Execution Trace*  $f_0, \dots, f_{W-1}$  and the *Composition Polynomial Trace*  $h_0, \dots, h_{M_2-1}$  which are both defined on the *Low Degree Extension*  $L_0$  of size  $\beta \cdot N$ .

The leaves of the corresponding Merkle are defined as follow:

- For the Execution Trace:

$$\text{input}_i = f_0(x) \parallel f_1(x) \parallel \dots \parallel f_{W-1}(x) \quad \text{where} \quad x \in L_0, \text{index}_0(x) = i$$

- For the Composition Trace:

$$\text{input}_i = h_0(x) \parallel h_1(x) \parallel \dots \parallel h_{M_2-1}(x) \quad \text{where} \quad x \in L_0, \text{index}_0(x) = i$$

*Remark.* If the *Randomized AIR with Preprocessing* [7] is used, the execution trace is committed in two times, using two Merkle trees (instead of a single one). The first tree contains the trace columns committed before the additional round, and the second tree contains the remaining trace columns.

### 2.2.3 Layer polynomial commitments

In the FRI protocol, a polynomial  $p_i$  is committed at the layer  $i + 1$  (except for the last layer). The evaluation domain for the commitment of  $p_i$  is  $L_i$ . To compute an evaluation of  $p_{i+1}$  for a point of  $L_{i+1}$  using the Equation (3), one needs  $2^{\ell_{i+1}}$  evaluations of  $p_i$ . Thus these evaluations are committed together. The commitment of  $p_i$  is realized thanks to a Merkle tree with  $\frac{|L_i|}{2^{\ell_{i+1}}}$  leaves, and the corresponding leaves are defined as follows:

$$\text{input}_j = p_i(x_0) \parallel p_i(x_1) \parallel \dots \parallel p_i(x_{2^{\ell_{i+1}}-1})$$

where

$$x_k \in L_i \quad \text{and} \quad \text{index}_i(x_k) = j \cdot 2^{\ell_{i+1}} + k .$$

## 2.3 FRI protocol

The implementation of the FRI verification process is based on a trick that we call *scaling* (explained in Section 2.3.1) and it makes use of two data structures: the *FRI context* (specified in Section 2.3.2) and the *FRI queue* (specified in Section 2.3.3).

### 2.3.1 Scaling

Instead of performing the standard FRI protocol on the DEEP composition polynomial  $p_0$ , the implementation runs the protocol on the *scaled DEEP composition polynomial*  $p'_0$  (see Section 2.1 for the definition of scaled notions). This choice enables to avoid all the offsets  $\{c_{\text{LDE}}^{2^s i}\}_i$  of the evaluation domains  $\{L_i\}_i$ . Instead, the protocol is performed on the scaled evaluation domains  $\{L'_i\}$  which are subgroups of  $\mathbb{F}_p^\times$  easier to handle.

This change of representation does not impact the commitments defined in Section 2.2.3, since  $p'_i(x') = p_i(x)$  whenever  $x = x' \cdot c_{\text{LDE}}^{2^{s_i}}$  (with  $x \in L_i$  and  $x' \in L'_i$ ). It only implies that the IOP verifier samples scaled challenges  $\{\zeta'_i\}_i$  instead of  $\{\zeta_i\}_i$  (but the two distributions are actually equivalent) and that the IOP prover sends the scaled version of the last layer polynomial  $p'_{K-1}$  (instead of  $p_{K-1}$ ).

### 2.3.2 FRI context

The FRI context is a data structure which contains three different lists. The two last lists contain elements which are computed once for all at the beginning of the verification. The first list is the only one to be updated along the proof verification. The three lists are described hereafter:

- The *FRI evaluations*. This list contains the  $2^{\ell_{i+1}}$  evaluations

$$\{p_i(v \xi^j), 0 \leq j < 2^{\ell_{i+1}}\},$$

which will be used to compute  $p_{i+1}(v^{2^{\ell_{i+1}}})$ , where  $\xi$  denotes a generator of the multiplicative subgroup of  $\mathbb{F}_p^\times$  of order  $2^{\ell_{i+1}}$ .

- The *FRI group*. This is the subgroup of  $\mathbb{F}_p^\times$  with  $2^{\text{MAX\_FRI\_STEP}}$  elements, sorted in a specific order. Let  $\xi$  be a generator of this group.<sup>1</sup> Then the  $i$ th element of the group, denoted `friGroup[i]`, is defined as

$$\text{friGroup}[i] := \xi^e \iff i = \text{bit-reverse}_{\text{MAX\_FRI\_STEP}}(e).$$

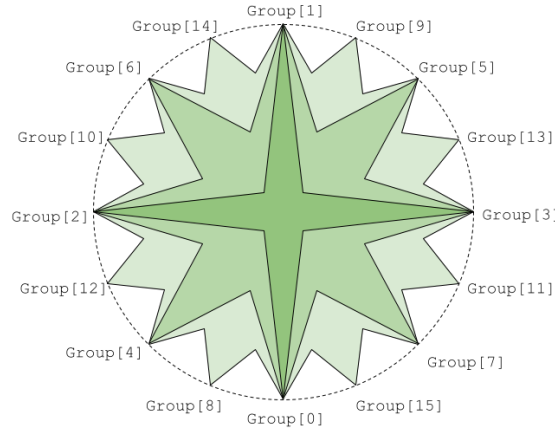


Figure 2: The FRI group when `MAX_FRI_STEP` = 4.

Figure 2 represents this group for `MAX_FRI_STEP` = 4 (the current choice in the audited code). This order implies the two following properties:

<sup>1</sup>In the code, this generator is implemented via the constant `FriLayer.FRI_GROUP_GEN`.

- We have  $\text{friGroup}[2i+1] = -\text{friGroup}[2i]$ .
- Let  $\text{order}(\cdot)$  denote the multiplicative order of an element of  $\mathbb{F}_p^\times$ . If two group elements  $x, y$  verify  $\text{order}(x) < \text{order}(y)$ , then  $x$  is stored before  $y$ . It results that the  $2^\ell$  first elements form the multiplicative subgroup of  $\mathbb{F}_p^\times$  with  $2^\ell$  elements, for any  $0 \leq \ell \leq \text{MAX\_FRI\_STEP}$ .
- The *FRI half group of inverses*. This list contains half of the inverse elements from the previous list:

$$\text{friHalfInvGroup}[i] := \frac{1}{\text{friGroup}[2i]} ,$$

for every  $i < 2^{\text{MAX\_FRI\_STEP}-1}$ .

### 2.3.3 FRI queue

The FRI queue is a structure which contains the FRI queries, *i.e.* evaluations of layer polynomials which must be verified by Merkle decommitment. The FRI queue is updated along the FRI verification process. At the beginning the queue only contains evaluations of the DEEP composition polynomial  $p_0$ . Then while processing the  $i$ th layer, for  $i$  from 1 to  $K-1$ , the layer- $(i-1)$  queries are replaced by layer- $i$  queries. Finally the last layer checks the consistency of the evaluations of  $p_{K-1}$  present in the queue at the end.

A layer- $i$  FRI query corresponding to an evaluation point  $v = c_{\text{LDE}}^{2^{s_i}} \cdot g_i^e \in L_i$ , with scaled value  $v' = g_i^e \in L'_i$ , is represented as a triplet

$$(\overline{\text{index}}_i(v), p_i(v), 1/v')$$

where (as introduced in Section 2.1)

$$\overline{\text{index}}_i(v) = \text{index}_i(v) + |L_i| = \text{bit-reverse}_{\log |L_i|}(e) + |L_i| .$$

Each element of the FRI query triplet is stored on 32 bytes.

*Remark.* There are several reasons to keep  $\overline{\text{index}}_i(v)$  instead of  $\text{index}_i(v)$ . First, it directly gives the indexes of the corresponding Merkle leaves (let recall the leaves indexes have an offset of  $2^n$  when the Merkle tree has  $2^n$  leaves) up to a shift operation. It also enables to easily identify the set  $L_i$  to which an evaluation point belongs to (thanks to the leading bit of  $\overline{\text{index}}_i(v)$ ).

The elements in the FRI queue must respect the following order:

- All the present elements of the same layer must be contiguous in the queue and must have their indices in increasing order.
- An element belonging to the  $i$ th layer must be before an element belonging to the  $j$ th layer if  $i < j$ .

## 2.4 STARK verifier

We specify hereafter the format and constraints underlying the implementation of the STARK verifier (besides the FRI context, FRI queue, and Merkle queue, already specified above). The STARK verifier is configured by a set of parameters which must verify some constraints (see Section 2.4.1). It takes as input a proof and relies on a context, for which formats are respectively specified in Section 2.4.2 and Section 2.4.3 hereafter.

### 2.4.1 Parameters

Besides the statement which must be proved (*i.e.* the “public input”) and the proof itself, the STARK verifier received a list of parameters which must verify some constraints.

Parameter	Constraints
The number of queries	-
The logarithm of the blowup factor $\beta$	$1 \leq \log_2(\beta) \leq 16$
The number of security bits for the proof of work	$\text{minProofOfWorkBits} \leq \dots \leq 50$ $\dots < \text{numSecurityBits}$
The bound on the log-degree of $p_{K-1}$	$\log_2(\deg p_{K-1}) \leq 10$
The number of FRI layers	$1 < K \leq 10$
The number of steps for each layer	$\text{fri\_step\_list}[0] = 0$ $0 < \text{fri\_step\_list}[i] \leq 4$ for $i \geq 1$

### 2.4.2 Proof format

As input of the STARK verifier, the user must give the proof which will convince the verifier about the correctness of the statement. The proof is a sequence of bytes. The type of the proof in `StarkVerifier.verifyProof` is

```
uint256[] memory proof ,
```

but it is manipulated as an array of bytes in the code. Specifically, the proof is the sequence of fields which are listed in the table below. All those fields are composed of one or several 256-bit (32-byte) elements, except one: the nonce of the proof of work, which is a single 64-bit (8-byte) value. As a result, the fields succeeding the nonce are not aligned on 32 bytes. The STARK verifier parses the proof assuming it contains the elements described in the following table.

Field	Description
Trace Commitment	Merkle root where leaves are evaluations of some trace column polynomials in the LDE (see Section 2.2.2).
Trace Commitment “Interactive”	<i>Only if the proof deals with Randomized Air with Preprocessing.</i> Merkle root where leaves are evaluations of other trace column polynomials in the LDE.
OODS Commitment	Merkle roots where leaves are evaluations of the composition polynomial columns in the LDE.
OODS Values	The $M_1 + M_2$ values $\{y_\ell\}_\ell \cup \{\hat{y}_i\}_i$ .
FRI Commitments	Merkle roots where leaves are evaluations of the layer polynomials (except the last one).
Last Layer Polynomial	All the coefficients of the <i>scaled</i> last layer polynomial $p'_{K-1}$ , starting from the coefficient of the constant monomial.
Nonce of Proof of Work	The nonce used to prove the work (a.k.a. grinding). <i>Only 64 bits.</i>
Trace Query Responses	The queried rows of the trace.
-	Authentication paths in the Merkle tree of the Trace Commitment to decommit the trace query responses.
Trace Query Responses 2	<i>Only if the proof deals with Randomized Air with Preprocessing.</i> The queried rows of the trace.
-	Authentication paths in the Merkle tree of the Trace Commitment (Interactive) to decommit the trace query responses.
Composition Query Responses	The queried rows of the composition polynomial columns.
-	Authentication paths in the Merkle tree of the OODS Commitment to decommit the composition query responses.
<i>Repeat the two last rows for each FRI layer (but the last one), i.e. for layer <math>i \in \{1, \dots, K-1\}</math>.</i>	
Required Evaluation Values for the layer $i$	The missing required evaluations of $p_{i-1}$ to compute the evaluations of $p_i$ on the queries $Q_i$ .
-	Authentication paths in the Merkle tree with the $i$ th FRI Commitment as root to valid the Evaluation Values of the current layer (i.e. layer $i$ ).

All the elements of  $\mathbb{F}_p$  (and their commitments) in the proof are in the *Montgomery form*, i.e. are multiplied by a coefficient  $R$  which verifies

$$R \equiv 2^{256} \pmod{p}.$$

### 2.4.3 Verifier context

The state of the verifier, called *verifier context*, is stored in a contiguous chunk of memory. This state is an array of elements of 256 bits, and contains the fields described in the below table. Each field is stored in a contiguous part of the state, the corresponding offsets are given by the constants. The column “Fixed” in the table indicates if the corresponding field contains a value which is set only once during the proof checking.

Constants	Fixed	Description
MM_BLOW_UP_FACTOR	✓	The blowup factor $\beta$
MM_LOG_EVAL_DOMAIN_SIZE	✓	The logarithm of the size of the Low Degree Extension: $\log_2(\beta \cdot N)$
MM_EVAL_DOMAIN_SIZE	✓	The size of the Low Degree Extension: $\beta \cdot N$
MM_EVAL_DOMAIN_GENERATOR	✓	The generator $g_{\text{LDE}}$ of the Low Degree Extension.
MM_PROOF_OF_WORK_BITS	✓	The required number of leading zeros in the proof of work.
MM_TRACE_COMMITMENT	✓	The Merkle root for commitments of the execution trace.
MM_OODS_COMMITMENT	✓	The Merkle root for commitments of the composition polynomial trace.
MM_N_UNIQUE_QUERIES	✗	The number of queries for the first layer ( <i>i.e.</i> size of the set $\mathcal{Q}_0$ ).
MM_CHANNEL	✗	Pointer to the verifier channel (further pointing to the proof and the PRNG).
MM_MERKLE_QUEUE	✗	The Merkle queue, which contains the values to be checked in the next decommitment verification.
MM_FRI_QUEUE	✗	The FRI queue, which contains the evaluation values of the next layer polynomial.
MM_FRI_QUERIES_DELIMITER	✓	End of the FRI queue. Used to detect the end of the FRI queue.
MM_FRI_CTX	✓/✗	The FRI context (see Section 2.3.2).
MM_FRI_STEPS_PTR	✓	Pointer to the list <code>fri_step_list[]</code> .
MM_FRI_EVAL_POINTS	✓	<i>Scaled</i> coefficients $\zeta'_0, \dots, \zeta'_{K-2}$ used to half the current <i>scaled</i> layer polynomial to build the polynomial of the next layer.
MM_FRI_COMMITMENTS	✓	The Merkle roots for commitments of the layer polynomials.
MM_FRI_LAST_LAYER_DEG_BOUND	✓	The number of monomials for the scaled last layer polynomial $p'_{K-1}$ ( <i>i.e.</i> the degree of this polynomial, added to 1).
MM_FRI_LAST_LAYER_PTR	✓	A pointer to the coefficients for the scaled last layer polynomial $p'_{K-1}$ .
MM_TRACE_LENGTH	✓	The trace length $N$ .
MM_TRACE_GENERATOR	✓	The generator $g$ of the trace evaluation domain.
MM_OODS_POINT	✓	The OODS point $z$ .
MM_OODS_EVAL_POINTS	✓	The OODS Evaluation Points, <i>i.e.</i> the queries for the first layer polynomial.
MM_TRACE_QUERY_RESPONSES	✓	The trace values for the queried rows.
MM_COMPOSITION_QUERY_RESPONSES	✓	The values of the composition polynomial traces for the queried rows.
MM_CONTEXT_SIZE	✓	The size of the verifier context.

## 2.5 External dependencies

### 2.5.1 Memory mapping contract

The memory mapping contract `MemoryMap` contains only declarations of constants which are related to the context of the STARK verifier. This context is stored in a contiguous



chunk of memory, and the constants are the offsets of the different fields. Users are free to organized the context as they want, but the offsets listed in Section 2.4.3 must be defined by the memory mapping contract.

### 2.5.2 OODS contract

Let us recall that the DEEP composition polynomial is in the form

$$p_0(x) = \sum_{\ell=0}^{M_1-1} \gamma_{\ell} \cdot \frac{f_{j_{\ell}}(x) - y_{\ell}}{x - zg^{s_{\ell}}} + \sum_{i=0}^{M_2-1} \gamma_{M_1+i} \cdot \frac{h_i(x) - \hat{y}_i}{x - z^{M_2}}.$$

Given the OODS point  $z$  (sampled by the verifier), the corresponding OODS values  $y_{\ell}$  and  $\hat{y}_i$  (given by the prover), the OODS coefficients  $\gamma_i$  (sampled by the verifier) and the evaluation of  $f_i$  and  $h_i$  for the queries, the OODS contract must return the evaluations for the queries of the DEEP composition polynomial and the inverses of all the evaluation points.

In practice, the OODS contract is autogenerated with hardcoded DEEP composition polynomial. This contract computes evaluations of  $p_0$  efficiently with a strategy of batching for the inverses (the denominators of rational functions and the inverses of the evaluation points).

No implementation of the OODS contract has been reviewed in the present audit. However, we stress that the OODS contract must comply to some API to be compatible with the audited STARK verifier. In particular, the OODS contract must implement a function which takes the verifier context as input and which returns the initial FRI queue (in which the  $p_0$  evaluations corresponding to the initial FRI queries have been filled under the format specified in Section 2.3.3). Moreover, with the current implementation of the STARK verifier, the evaluations of  $f_i$  and  $h_i$  in input of the OODS contract must be in *Montgomery form* while the point  $z$ , the values  $\{y_{\ell}\}_{\ell} \cap \{\hat{y}_i\}_i$  and the coefficients  $\gamma_i$  must be in *standard form*. Then the OODS contract must return evaluation values of the DEEP Composition Polynomial in the *standard form*.

### 2.5.3 Derived STARK verifier contract

The STARK verifier is an abstract contract. This means that to run an instance of the STARK verifier, one must implements a derived contract and the latter must implements the following functions:

- **getNColumnsInTrace**: Returns the number  $W$  of trace columns.
- **getNColumnsInComposition**: Returns the number  $M_2$  of composition polynomial trace columns.
- **getNCoefficients**: Returns the number of coefficients  $\{\alpha_j\}_j \cup \{\beta_j\}_j$  to build the composition polynomial from the AIR polynomial constraints.
- **getNOodsValues**: Returns the number  $M_1 + M_2$  of OODS values  $\{y_{\ell}\}_{\ell} \cup \{\hat{y}_i\}_i$ .

- **getNOodsCoefficients:** Returns the number  $M_1 + M_2$  of OODS coefficients  $\{\gamma_i\}_i$ .
- **getMmCoefficients:** Returns the memory offset where the coefficients  $\{\alpha_j\}_j \cup \{\beta_j\}_j$  must be stored.
- **getMmOodsValues:** Returns the memory offset where the OODS values must be stored.
- **getMmOodsCoefficients:** Returns the memory offset where the OODS coefficients must be stored.

In case *Randomized AIR with Preprocessing* [7] is used, the following functions must be overridden since the execution trace is computed in two rounds (instead of one):

- **getNInteractionElements:** Returns the number of  $\mathbb{F}_p$  elements that the IOP verifier must sample between both rounds to commit the execution trace.
- **getMmInteractionElements:** Returns the memory offset where the above elements must be stored.
- **getNColumnsInTrace0:** Returns the number of committed trace columns in the first round.
- **getNColumnsInTrace1:** Returns the number of committed trace columns in the second round (after sampling the interaction elements).

One must further implements these functions:

- **airSpecificInit:** Creates the verifier context, of size `MM_CONTEXT_SIZE`, and initializes all the context fields which are specific to the proved statement. Must return the logarithm (in base two) of the trace length  $N$ .
- **getPublicInputHash:** Returns the hash of the public input (*i.e.* the statement which must be proved). It is used to initialize the PRNG of the IOP verifier.
- **oodsConsistencyCheck:** Checks that the OODS values are consistent, *i.e.* that the mask  $\{y_\ell\}_\ell$  for the OODS point given by the prover is consistent with the evaluations of the composition polynomial trace  $h_0, \dots, h_{M_2-1}$ , and raises an error otherwise. To proceed, the function uses the coefficients  $\{\alpha_j\}_j \cup \{\beta_j\}_j$  and the interaction elements (if any).

## 3 Review and specific observations

### 3.1 Arithmetic primitives

#### 3.1.1 Contract PrimeFieldElement0

The contract `PrimeFieldElement0` defines arithmetic functions on the field  $\mathbb{F}_p$ , namely the addition (`fadd`), subtraction (`fsub`), multiplication (`fmul`), exponentiation (`pow`) and inversion (`inverse`). The field prime characteristic  $p$ , declared as the constant `K_MODULUS`, is defined as

$$p = 2^{251} + 17 \cdot 2^{192} + 1.$$

As required,  $p$  is a prime and the order  $p - 1$  of the multiplicative group  $\mathbb{F}_p^\times$  is a multiple of a large power of two:

$$p - 1 = 2^{192} \cdot 5 \cdot 7 \cdot 98714381 \cdot 166848103$$

(allowing large trace length  $N$ ). The implementation of the STARK verifier sometimes requires a generator of the multiplicative group  $\mathbb{F}_p^\times$ . This generator, which is declared here as the constant `GENERATOR_VAL`, is defined as  $g = 3$ .

The contract `PrimeFieldElement0` further implements functions for the conversion to/from Montgomery form: `toMontgomeryInt`, `fromMontgomery`, `fromMontgomeryBytes`. The Montgomery factor  $R$  and its inverse  $R^{-1} \bmod p$  (used by those conversion functions) are also declared as the constants `K_MONTGOMERY_R` and `K_MONTGOMERY_R_INV`. As required, the following relations are verified:

$$\begin{cases} R \equiv 2^{256} \bmod p \\ R \cdot R^{-1} \equiv 1 \bmod p \end{cases}.$$

*General observations:* Observations 21 (●), 22 (●) and 23 (●) are applicable to this contract.

#### ● Observation 1: Montgomery arithmetic

The choice of using Montgomery arithmetic is not documented. Our understanding is that Montgomery arithmetic is used in the proof computation, presumably for performances. It results that some values from the proof are in Montgomery form and must be converted to standard form in the verification process. It is not clear why those conversions are not done in the proof computation; this would lighten the verification process executed on the EVM. Also the special sparse form of the prime  $p$  might allow efficient modular arithmetic without relying on the Montgomery form.

**Recommendation:** We recommend to document the choice of Montgomery. We further suggest that

- converting values to the standard form might be done on the proof side to lighten the EVM computation;

- using the special form of  $p$  might allow efficient arithmetic without relying on the Montgomery form.

---

**Status: Unresolved (✗)**

This recommendation has not been implemented since it would require changing the implementation of the prover.

---

### 3.1.2 Contract HornerEvaluator

The contract `HornerEvaluator` implements the evaluation of a polynomial

$$p(x) := \sum_{i=0}^{\text{nCoefs}-1} \text{coefsStart}[i] \cdot x^i$$

for a given point  $x := \text{point}$  via the function of prototype

```
function hornerEval( uint256 coefsStart, uint256 point, uint256 nCoefs )
    internal pure returns (uint256) .
```

This function is used to compute the evaluations of the polynomial  $p_{K-1}$  in the last FRI layer for the queries  $Q_{K-1}$ .

● **Observation 2: Check of unclear purpose**

The purpose of the check “`nCoefs < 4096`” is not clear here. It does not seem to be a constraint from the implementation of the function `hornerEval`.

**Recommendation:** If this check is due to a constraint on the proof design, we would recommend to move it in the function

```
StarkVerifier.initVerifierParams
```

which is dedicated to this kind of checks. But since this function already includes a stronger constraint, which is

```
require(logFriLastLayerDegBound <= 10, "..."),
```

it seems this check could simply be removed.

---

**Status: Partially resolved (∼)**

A comment has been added but the purpose of the test remains unclear to us.

---

*General observations:* Observations 24 (■) and 25 (●) are applicable to this contract.

## 3.2 Verifier channel

### 3.2.1 Contract Prng

This contract implements the pseudo-random number generator (PRNG) of the IOP verifier, which aims to be secure when the underlying hash function is modeled as a *random oracle*. The PRNG state is composed of a seed (called `digest` in the implementation since this seed is always a hash function output) and a counter. When the verifier needs randomness, she calls the function `getRandomBytes`, which computes the hash digest of the pair (seed, counter), increments the counter and returns the computed hash (32 bytes) as randomness. The function `initPrng` initializes the PRNG state with an input seed (and set the counter to 0), and the function `mixSeedWithBytes` re-seeds the PRNG (by mixing the previous seed with some input data). Finally, the functions `getPrngDigest`, `loadPrng` and `storePrng` are getters and setter for the PRNG state.

#### ■ Observation 3: Naming

The variable names `prngPtr` and `statePtr` are both used to refer to the same concept.

**Recommendation:** We recommend to use a single name. `statePtr` (or `prngStatePtr`) is probably clearer.

#### Status: Resolved (✓)

All these variables are now named as `prngPtr`.

#### ● Observation 4: Seed refreshing

The way the PRNG state is mixed with fresh bytes in the `mixSeedWithByte` function might be weak. When the seed (a.k.a. *digest*) is refreshed using fresh bytes taking a value  $x$ , the new seed value is derived as

$$\text{newSeed} := \text{Hash}(\text{seed} \parallel x) .$$

On the other hand, the function which generates random bytes from the seed (before refreshing) and a counter value  $ctr$  is defined as

$$\text{randomBytes} := \text{Hash}(\text{seed} \parallel ctr) ,$$

where the hash function is similar in both cases.

We observe that if the value  $x$  of the fresh bytes collides with a value  $ctr$  previously taken by the counter, then `newSeed` will equal a previously generated `randomBytes` output of the PRNG. If such collision happens, it might break the security proof of the STARK protocol.

In practice, the function `mixSeedWithByte` is inlined by the function `readBytes` (from the `VerifierChannel` contract) which might refresh the seed with two types of fresh bytes read from the proof:

- Merkle roots of {trace, OODS, FRI} commitments (function `readHash`),
- OODS values (function `readFieldElement`).

An adversary could deliberately set one of these value to a previous counter value. While it would be hard to generate a valid Merkle root taking the value of a previous counter (and an invalid Merkle root would be detected by the Merkle verifier), the impact of such cheating is less clear for the OODS values.

More generally, this PRNG design does not comply to the security proof of the IOP paper [6] in which two different random oracles  $\rho_1$  and  $\rho_2$  are used for randomness generation and reseeding.

**Recommendation:** We recommend to use two different hash functions for randomness generation and reseeding. For example:

- `newSeed := Hash(0 || seed || x)` for reseeding,
- `randomBytes := Hash(1 || seed || ctr)` for randomness generation.

---

**Status: Resolved (✓)**

The reseeding strategy now consists in computing the new seed as

$$\text{newSeed} := \text{Hash}((\text{seed} + 1) \parallel x),$$

while the randomness generation has not changed.

*General observations:* Observations 22 (●) and 23 (●) are applicable to this contract.

### 3.2.2 Contract `VerifierChannel`

The *Verifier Channel* is a structure which contains

- the pointer to a position in the proof (which must have the format defined in Section 2.4.2);
- the PRNG which provides the randomness for the IOP verifier.

The function `initChannel` initializes the channel, and the function `getPrngPtr` returns the pointer to the PRNG state. Two functions are then dedicated to the sampling of random elements:

- `sendFieldElements` samples `nElements` elements of  $\mathbb{F}_p$  using the PRNG of the channel and stores them in the memory at address `targetPtr`. A conversion from Montgomery form to standard form is applied to each sampled value.
- `sendRandomQueries` samples `count` random elements in  $\{0, \dots, \text{mask}\}$ , sorts them and removes redundancy. The resulting list is stored in the memory at address `queriesOutPtr`.

The contract further implements three functions to read elements from the proof:

- `readBytes` reads a 256-bit value from the proof and increases the proof pointer. If the input `mix` is set at `true`, the PRNG is reseeded using the read value.
- `readHash` is similar to `readBytes` (it actually calls `readBytes` with the same input).
- `readFieldElement` proceeds as `readBytes`, but further applies a conversion from Montgomery form to standard form to the output.

Finally, the contract implements a check function for the grinding process. This function, named `verifyProofOfWork`, reads a 64-bit nonce from the proof and check that the prover computed a correct proof of work on the current seed of the PRNG. Specifically, this function checks the nonce satisfies that

$$\text{Hash}(\text{Hash}(0x0123456789abcded||\text{prngSeed}||\text{workBits})||\text{nonce})$$

has the `workBits` most significant bits to zero. In practice, this verification is run once before sampling the queries.

#### ● Observation 5: Sampling rejection

To sample an element  $x$  in  $\{0, \dots, p-1\}$  from a random 256-bit value  $y$ , the function `sendFieldElements` considers the 252 least significant bits  $\hat{y}$  of  $y$ . Then, if  $\hat{y} < p$ , the function defines  $x$  as  $\hat{y}$ , otherwise it samples a new random 256-bit value  $y$  and repeats. In average, the function will sample two 256-bit values to get a random element of  $\mathbb{F}_p$  (this is because of the sparseness of the most significant bits of  $p$ ).

**Recommendation:** We suggest to proceed as follows to obtain a lower rejection rate. From a uniform random 256-bit value  $y$ : if  $y < 31 \cdot p$ , define  $x$  as

$$x := y \bmod p ,$$

otherwise sample a new random 256-bit value  $y$  and repeat. This methodology reduces the average number of 256-bit samplings to 1.03 (instead of 2.00), which reduces almost by half the number of calls to `keccak256`.

#### Status: Resolved (✓)

The sampling rejection strategy has been revised as suggested.

#### ● Observation 6: Bias in the output distribution of `sendRandomQueries`

In the function `sendRandomQueries`, the variable `curr` is not explicitly initialized. Therefore, according to the Solidity documentation,<sup>a</sup> `curr` is implicitly initialized to zero. For the first sampled query, the program does not enter in the loop `while (ptr > queriesOutPtr)`. Then, for the evaluation of the condition `queryIdx != curr`, the variable `curr` is still zero. As a result, if the first sampled query index is zero, then it is rejected. This implies a small bias in the output distribution: sampling 0

for the first sampled query should occur with probability  $1/(\beta \cdot N)$  (since in practice  $\text{mask} = \beta \cdot N - 1$ ), but it occurs with probability 0 instead (and hence the other values are sampled with probability  $1/(\beta \cdot N - 1)$ ).

**Recommendation:** We recommend to explicitly initialize the variable `curr` with a value outside the interval  $\{0, \dots, \text{mask}\}$ .

**Status: Resolved (✓)**

The variable `curr` is now initialized with `uint256(-1)` which is outside the interval  $\{0, \dots, \text{mask}\}$  thanks to the inequality  $\text{mask} < 2^{64}$ .

<sup>a</sup><https://docs.soliditylang.org/en/v0.8.10/control-structures.html#default-value>

### ● Observation 7: Undefined behavior

If the input `mask` is equal or greater than  $2^{64}$ , the function `sendRandomQueries` has an undefined behavior. In practice, it implies that the evaluation domain  $L_0$  must have less than  $2^{64}$  elements, *i.e.* the relation  $\beta \cdot N \leq 2^{64}$  must be verified.

**Recommendation:** We recommend

- either to add an explicit requirement at the beginning of the function

```
require((mask > 64) == 0, "...");
```

and to further add a check on  $\beta \cdot N$  in the function `initVerifierParams`,

- or to change the implementation of `sendRandomQueries` to remove this limitation.

**Status: Resolved (✓)**

There is now an explicit requirement (at the beginning of the function) that prevents the undefined behavior. No checks have been added in the function `initVerifierParams`.

*General observations:* Observations 21 (●), 22 (●) and 26 (●) are applicable to this contract.

## 3.3 Merkle trees

### 3.3.1 Contract `IMerkleVerifier`

The contract `IMerkleVerifier` acts as an interface (although it does not use the keyword `interface`). It defines the following upper bound

```
uint256 internal constant MAX_N_MERKLE_VERIFIER_QUERIES = 128;
```



for the number of decommitted values in a call to the Merkle verifier. It also declares the following virtual function for the verification of decommitted values:

```
function verifyMerkle(  
    uint256 channelPtr,  
    uint256 queuePtr,  
    bytes32 root,  
    uint256 n  
    ) internal view virtual returns (bytes32 hash);
```

`channelPtr` is a pointer to the authentication paths in the Merkle tree (via the proof). `queuePtr` is a Merkle queue (see Section 2.2.1) containing the `n` leaves to be verified. The Merkle root is given by `root`. If the decommitment is invalid, the function shall raise an error, otherwise it shall return the Merkle root.

### 3.3.2 Contract MerkleVerifier

The contract `MerkleVerifier` is a derived contract of the `IMerkleVerifier` abstract contract. Its implementation of the function `verifyMerkle` (defined in Section 2.2.1) follows this iterative algorithm:

1. While the queue head is not the Merkle root (*i.e.* while the index of the node at the queue head is not 1),
  - 1.1. Pop a node at the head of the queue.
  - 1.2. If the next node in the queue is the sibling node, then pop it, otherwise get the sibling node from the verifier channel (*i.e.* from the proof). Note: Since the queue is sorted by increasing node index (see Section 2.2.1), if the sibling node is in the queue, it must be the next node.
  - 1.3. From the two sibling nodes, compute the label of the parent node and push it (together with corresponding index) in the queue.
2. Compare the computed root with the input root, and raise an error in case of mismatch.

As explained in the comment of `verifyMerkle`, the content of the input Merkle queue is destroyed during the process.

#### ● Observation 8: slotSize

The variable `slotSize` is used to store the constant `0x40` in the function `verifyMerkle`. Then the function sometimes uses this variable and sometimes hardcodes the constant `0x40` when the slot size is required.

**Recommendation:** Declare a constant for the slot size instead of a local variable and use this constant instead of hardcoding `0x40`.

**Status: Resolved (✓)**

Different constants (`MERKLE_SLOT_SIZE_IN_BYTES`, `COMMITMENT_SIZE_IN_BYTES`, ...) are now used to represent this value.

**● Observation 9: Hardcoded security parameter**

The hash function used for the Merkle commitments is implicitly defined as KECCAK-256 truncated to 160 bits (the most significant bits of the output), which implicitly sets the collision security of Merkle trees to 80 bits. The truncation mask is hardcoded in the function `getHashMask` and does not appear as the other constants of the verifier. It is not clear why a function declaration is needed here instead of a simple constant.

**Recommendation:** Treat the truncation mask as the other verifier constants.

**Status: Resolved (✓)**

The function `getHashMask` has been replaced by the constant `COMMITMENT_MASK`.

### 3.3.3 Contract `MerkleStatementContract`

The contract `MerkleVerifier` implements the function `verifyMerkle` as an internal function which assumes that its inputs are in the right format. In contrast, the contract `MerkleStatementVerifier` provides a public instance of `verifyMerkle`. The latter first applies a batch of format checks:

- the height of the tree (input `height`) must be lower than 200;
- the length of the Merkle queue (as an array of 256-bit values) must be even and at most twice `MAX_N_MERKLE_VERIFIER_QUERIES` (constant defined in `IMerkleVerifier`);
- the node in the queue must be of indices in  $\{2^h, \dots, 2^{h+1} - 1\}$  where  $h$  is here the height of the Merkle tree (input `height`), *i.e.* the nodes in the input queue must be leaves of the tree;
- the node indices in the queue must be increasing.

After completing those checks (and in case of success) the function then calls the `verifyMerkle` function from `MerkleVerifier` to check the decommitment validity. If the Merkle queue has a valid format and if the decommitment is also valid, then the function `registerFact` from the base contract `FactRegistry` is called to keep track of the fact that the input Merkle queue is a valid decommitment for the input root. `registerFact` stores the valid fact as a hash digest, which is built as

$$\text{ValidFact} := \text{Hash}(\text{MerkleQueue} || \text{MerkleRoot}) .$$

### ● Observation 10: Magic number

The maximum height of the Merkle tree is hardcoded as the magic number 200.

**Recommendation:** We recommend to avoid such a magic number by defining a constant and to document the choice of 200.

### Status: Partially resolved (~)

This remains as a magic number. A comment has been added which explains that this upper bound is somewhat arbitrary.

### 3.3.4 Contract MerkleStatementVerifier

The contract `MerkleVerifier` is a derived contract of the `IMerkleVerifier` abstract contract. It has a state variable which is the address of an instance of the contract `MerkleStatementContract` (initialized by the constructor). Its implementation of the function `verifyMerkle` (defined in Section 2.2.1) computes the hash of the corresponding “Merkle statement” from the input Merkle queue and the input Merkle root:

$$\text{Hash}(\text{MerkleQueue} || \text{MerkleRoot}) ,$$

and checks whether this hash is registered as valid fact in the `MerkleStatementContract` instance pointed by the stored address. If the statement is not registered, then an error is raised. Otherwise the function outputs the Merkle root.

## 3.4 FRI protocol

### 3.4.1 Contract FriLayer

The contract `FriLayer` implements the processing of one FRI layer, which aims at moving from the layer polynomial  $p_i$  to the layer polynomial  $p_{i+1}$  (as defined in Section 2.1). Specifically, from the set of queries  $\mathcal{Q}_i$  and corresponding evaluations of  $p_i$ , it

- computes the leaves to be decommitted from the Merkle tree of  $p_i$ ;
- computes the next set  $\mathcal{Q}_{i+1}$  of queries:

$$\mathcal{Q}_{i+1} := \left\{ v^{2^{\ell_{i+1}}}, v \in \mathcal{Q}_i \right\} ;$$

- computes the evaluations of  $p_{i+1}$  on  $\mathcal{Q}_{i+1}$ .

For  $v_{i+1} \in \mathcal{Q}_{i+1}$ , the evaluation  $p_{i+1}(v_{i+1})$  can be computed from the evaluations of  $p_i$  on the coset  $v_i \cdot \langle \xi \rangle \subseteq L_i$ , where  $v_i \in \mathcal{Q}_i$  s.t.  $v_{i+1} = v_i^{2^{\ell_{i+1}}}$  and where  $\xi$  is a generator of the subgroup of  $\mathbb{F}_p^\times$  of size  $2^{\ell_{i+1}}$ . Some of these evaluations are obtained from the previous layer (namely the evaluations in the points  $v_i \cdot \langle \xi \rangle \cap \mathcal{Q}_i$ ) while the others are taken from the proof.

The main function of this contract is `computeNextLayer`. To be functional, it requires that the second and third fields of verifier context (see Section 2.3.2) have been previously initialized by the function `initFriGroups`. When `computeNextLayer` is called, the queries of  $Q_i$  and the evaluations of  $p_i$  on these queries are stored in the FRI queue (see Section 2.3.3) pointed by the input `friQueuePtr`. Since these evaluations are not sufficient to compute the evaluations of  $p_{i+1}$  on  $Q_{i+1}$ , the remaining evaluations are read from the proof through the verifier channel pointed by the input `channelPtr`. The function `computeNextLayer` makes repeated calls to the functions `gatherCosetInputs` and `doFriSteps` which are described hereafter.

The function `gatherCosetInputs` gathers all the evaluations of  $p_i$  on  $v_i \cdot \langle \xi \rangle$  for the query  $v_i$  at the head of the FRI queue. Let us recall that the corresponding element in the FRI queue is the triplet

$$\left( \overline{\text{index}_i(v_i)}, p_i(v_i), \frac{1}{v'_i} \right)$$

where  $v'_i = (c_{\text{LDE}}^{2^{s_i}})^{-1} \cdot v_i$  is the scaled value of  $v_i$  (see Section 2.1). Let us denote  $c$  the element of  $v_i \cdot \langle \xi \rangle$  with the lowest index. The index of  $c$  is called the *coset index* (variable `cosetIdx` in the code). The coset index is computed by setting the  $\ell_{i+1}$  right-most bits of the index of  $v_i$  to 0. Then all the indexes of the elements of  $v_i \cdot \langle \xi \rangle$  are listed as `cosetIdx`, `cosetIdx+1`, ..., `cosetIdx+2 $\ell_{i+1}$ -1`. For each index, the function checks whether the corresponding element is in the FRI queue (thanks to the structure of the queue the indexes are in increasing order). In case of match, the element is popped from the FRI queue and the corresponding evaluation (second item of the triplet) is added to the set of gathered evaluations. In case of mismatch, the evaluation corresponding to the current index is read from the proof. All the gathered evaluations are stored in the FRI context (see Section 2.3.2). The function `gatherCosetInputs` further returns

$$\text{cosetIdx} = \overline{\text{index}_i(c)} \quad \text{and} \quad \frac{1}{c'} = \frac{1}{v'_i} \cdot \xi^j$$

where  $c' = (c_{\text{LDE}}^{2^{s_i}})^{-1} \cdot c$  is the scaled value of  $c$  and where  $j = \text{bit-reverse}_{\log |L_i|}(\overline{\text{index}_i(v_i)} - \text{cosetIdx})$ . Those returned values will be used to compute the FRI queue triplet corresponding to the query  $v_{i+1}$ .

Once `gatherCosetInputs` done, the function `doFriSteps` is called to compute  $p_{i+1}(v_{i+1})$  from the gathered evaluations and the returned value  $1/c'$  (input `cosetOffset_` in the code). In practice, the implementation evaluates the scaled layer polynomial  $p'_{i+1}$  on  $v'_{i+1}$  using the scaled inverse  $1/c'$  which satisfies  $(1/c')^{2^{\ell_{i+1}}} = 1/v'_{i+1}$  (see Section 2.3.1 for explanations about the scaling). This computation is done by calling the function `doXFriSteps` with  $X \in \{2, 3, 4\}$  depending on the number of FRI steps in the current layer  $\ell_{i+1} = \text{fri\_step\_list}[i+1]$ . The function `doFriSteps` then computes the leave of the Merkle tree corresponding to the gathered evaluations of  $p_i$  (see Section 2.2.3), namely

$$p'_i(c) \parallel p'_i(c \cdot \xi^{j_1}) \parallel p'_i(c \cdot \xi^{j_2}) \parallel \dots$$

where the exponent  $j_1, j_2, \dots$ , are in increasing bit-reverse order (according to the definition to the FRI group in Section 2.3.2). This leave is stored in the Merkle queue (to be

verified as a valid decommitment later). Finally the function `doFriSteps` add the triplet

$$\left( \overline{\text{index}}_{i+1}(v_{i+1}), p_{i+1}(v_{i+1}), \frac{1}{v'_{i+1}} \right) \quad \text{with} \quad \overline{\text{index}}_{i+1}(v_{i+1}) := \frac{\overline{\text{index}}_{i+1}(c)}{2^{\ell_{i+1}}}$$

to the FRI queue as an element to be processed by the next FRI layer.

The function `computeNextLayer` repeats the calls to `gatherCosetInputs` and `doFriSteps` for each query  $v_i \in \mathcal{Q}_i$ . When all the queries from  $\mathcal{Q}_i$  have been popped from the FRI queue, the latter has been filled with all the queries from  $\mathcal{Q}_{i+1}$ . The function `computeNextLayer` then stops and returns the size of the new queue (*i.e.* the number of queries in  $\mathcal{Q}_{i+1}$ ).

*Methodology.* As exposed in the introduction of this report, our audit consists in a careful review of the source code. However, the implementation of `do2FriSteps`, `do3FriSteps` and `do4FriSteps` is too complex to validate their correctness through a simple code review. Therefore, we wrote a SageMath script that parses the implementation of these functions, evaluates them in a symbolic way, and checks the obtained outputs. The followed approach is detailed in Appendix A. All the tests of the script have passed, we therefore validate the correctness of these functions.

#### ■ Observation 11: Comments and variable naming

The comments and naming of variables are confusing in this contract. For instance:

- Our understanding is that the label `friEvalPoint` is used for the (scaled) sampled value  $\zeta'_i$  which is confusing since this is not an evaluation point.
- The labels `cosetOffset_` and `xInv` are both used for the values  $1/c'$  (which does not exist in the protocol description, *e.g.* [10]). The terminology `X` is further used with `friEvalPointDivByX`.
- There is an ambiguity on the definition of the “coset offset”. We can assume it corresponds to the value `c` in the following comment (lines 7–19):

```
The main component of FRI is the FRI step which
takes the i-th layer evaluations on a coset c*<g>
and produces a single evaluation in layer i+1.
```

This definition matches what we denote  $c$  in the above description.

The notation `c` is reused in the comment of `gatherCosetInputs` (lines 669-675):

```
// Get the algebraic coset offset:
// I.e. given c*g^(-k) compute c, where
//      g is the generator of the coset group.
//      k is bitReverse(offsetWithinCoset, log2(cosetSize)).
```

However, this `c` is in reality the *inverse* of the `c` from the first comment.

**Recommendation:** We recommend to document the multiple-step FRI protocol and the scaling trick with a proper definition of the different variables occurring in the computation. The comments in the code and a consistent naming convention could then be based on such a description.

**Status: Unresolved (X)**

The labels `friEvalPoint`, `cosetOffset_` and `xInv` are still used. Moreover, the two above comments stay unchanged.

#### ■ Observation 12: Incorrect function description

The description of the function `doFriSteps` (lines 795–805) states:

```
The input is read either from the queue or from
the proof depending on data availability. Since
the function reads from the queue it returns an
updated head pointer.
```

This comment is not relevant for `doFriSteps` (it might be for `gatherCosetInputs`). Moreover, earlier in the same description, the third output is described as

```
3. The root of a Merkle tree for the input layer.
```

but this should be a Merkle leaf (from the layer polynomial Merkle tree).

**Recommendation:** We recommend to correct the description of this function.

**Status: Unresolved (X)**

The description of the function `doFriSteps` has not changed.

#### ● Observation 13: Unused function

As also reported in Observation 22, the function

```
nextLayerElementFromTwoPreviousLayerElements
```

is never called. Our guess is that this function is meant to ease the understanding of the functions `do2FriSteps`, `do3FriSteps` and `do4FriSteps`. However, the inputs of this function are not the same as the inputs of the latter functions.

**Recommendation:** We recommend to avoid unused functions. If the above function is useful, we would suggest to clarify its purpose and to homogenize its name and arguments with the other functions.

**Status: Resolved (✓)**

The function `nextLayerElementFromTwoPreviousLayerElements` has been removed.

*General observations:* Observations 21 (●), 22 (●), 23 (●), 24 (■) and 27 (■) are applicable to this contract.

### 3.4.2 Contract Fri

The contract `Fri` processes and verifies all the FRI layers of the protocol. The main function `friVerifyLayers` is called when the FRI queue has been initialized with the set  $\mathcal{Q}_0$  of queries for  $p_0$ . The FRI queue is accessed through the field `MM_FRI_QUEUE` of the verifier context (input `ctx` of the function). First, the function calls to `FriLayer.initFriGroups` to populate the constant data in the FRI context (see Section 2.3.2). Then, it processes each FRI layer (but the last one) thanks to `FriLayer.computeNextLayer` and decommits the evaluations of the current layer polynomial thanks to `MerkleVerifier.verifyMerkle`. At the end, the FRI queue holds the evaluations of  $p_{K-1}$  on  $\mathcal{Q}_{K-1}$ . The function `verifyLastLayer` is then called to recompute these evaluations using the coefficients of the scaled layer polynomial  $p'_{K-1}$  (available in the field `MM_FRI_LAST_LAYER_PTR` of the verifier context) as well as the queries  $v' \in \mathcal{Q}'_{K-1}$  recovered by inverting the  $1/v'$  items of the elements of the FRI queue.

As explained in the comments of the contract, all the evaluations of the polynomials  $p_0, \dots, p_{K-1}$  are in the Montgomery form when calling `FriLayer.computeNextLayer` and `Fri.verifyLastLayer`. However, the computation is performed as if those evaluations were in the standard form. Both computation are indeed equivalent thanks to the following properties:

- two added values are always in the same form (both in the standard form, or both in the Montgomery form);
- when a value  $x \cdot R$  in Montgomery form is multiplied by a value  $y$ , the latter is always in the standard form. Then, the result is in the Montgomery form

$$(x \cdot R) \cdot y = (x \cdot y) \cdot R .$$

*General observations:* Observations 23 (●), 24 (●) and 27 (■) are applicable to this contract.

### 3.4.3 Contract FriStatementContract

This contract implements a public function `verifyFRI` which, given

- a FRI queue containing evaluations of  $p_i$  on a query set  $\mathcal{Q}_i$  for a layer  $i + 1$ ;
- the scaled sampled value  $\zeta'_i$  (input `evaluationPoint`);
- the number  $\ell_{i+1}$  of steps from  $p_i$  to  $p_{i+1}$  (input `friStepSize`);

- the Merkle root for the commitment of  $p_i$  on  $L_i$ ;
- a proof sequence,

process the layer  $(i+1)$  and checks that the revealed evaluations of  $p_i$  (from the FRI queue and from the proof) are consistent with the Merkle root. If the computation of the layer succeeds and if the decommitment of the evaluations of  $p_i$  is valid, then the **FactRegistry** contract is called to save the fact that, given the initial and the final states of the FRI queue, there exists a valid computation which links the both states and which is consistent with the given Merkle root.

The function proceeds as follows:

- it checks that the FRI queue has the right format (checks the length, checks that indices are increasing and in the right range, checks that the elements of  $\mathbb{F}_p$  are encoded with a value of  $\{0, \dots, p-1\}$ );
- it checks that  $\zeta'_i$  is an element of  $\{0, \dots, p-1\}$ ;
- it allocates the memory for the verifier channel, the Merkle queue and the FRI context;
- it initializes the FRI context by calling `FriLayer.initFriGroups`;
- it computes the next layer by calling `FriLayer.computeNextLayer`;
- it checks the decommitment of the revealed evaluations of  $p_i$  by calling `verifyMerkle`;
- in case of successful verification, it computes the hash

$$\text{Hash}(\zeta'_i \parallel \ell_{i+1} \parallel \text{Hash}(\text{FriQueue}_i) \parallel \text{Hash}(\text{FriQueue}_{i+1}) \parallel \text{MerkleRoot})$$

and stores it as a valid fact in the **FactRegistry** contract. Here  $\text{FriQueue}_i$  denotes the content of the queue before layer processing and  $\text{FriQueue}_{i+1}$  is the content of the queue after layer processing.

*General observations:* Observation 24 (■) is applicable to this contract.

#### 3.4.4 Contract `FriStatementVerifier`

The contract **FriStatementVerifier** verifies all the FRI layers, one by one, using the contract **FriStatementContract** which stores as valid facts the processed and verified transitions between two consecutive states of the FRI queue. It has a state variable which is the address of an instance of the contract **FriStatementContract** (initialized by the constructor).

The function `friVerifyLayers` starts from a FRI queue which contains the initial queries  $\mathcal{Q}_0$  and the corresponding evaluations of the DEEP composition polynomial  $p_0$  (computed by the OODS contract). It proceeds as follows:



- It converts the  $p_0$  evaluations (computed by the OODS contract) to the Montgomery form;
- It computes the hash digest  $h_0 := \text{Hash}(\text{FriQueue}_0)$ , where  $\text{FriQueue}_0$  denotes the initial content of the FRI queue;
- For each FRI layer (but the last one), *i.e.* for  $i$  in  $\{1, \dots, K-2\}$ ,
  - It reads the hash digest  $h_i := \text{Hash}(\text{FriQueue}_i)$  from the verifier channel;
  - It computes the hash

$$\text{Hash}(\zeta'_{i-1} \parallel \ell_i \parallel h_{i-1} \parallel h_i \parallel \text{MerkleRoot}_{i-1})$$

where  $\text{MerkleRoot}_{i-1}$  is the commitment of the  $p_{i-1}$  evaluations (obtained from the verifier context);

- It checks if this hash is registered in the `FriStatementContract` instance as a valid fact (an error is raised otherwise).
- Using the function `computerLastLayerHash`, it computes the final state of the FRI queue, denoted  $\text{FriQueue}_{K-1}$ . This is done by first computing the query set  $\mathcal{Q}_{K-1}$  from the query set  $\mathcal{Q}_0$  by

$$\mathcal{Q}_{K-1} = \{v^{2^{s_{K-1}}}, v \in \mathcal{Q}_0\},$$

then evaluating  $p_{K-1}$  for each queries in  $\mathcal{Q}_{K-1}$  by calling `hornerEval`. From this final states, it computes the hash  $h_{K-1} := \text{Hash}(\text{FriQueue}_{K-1})$ .

- Finally, it computes the hash

$$\text{Hash}(\zeta'_{K-2} \parallel \ell_{K-1} \parallel h_{K-2} \parallel h_{K-1} \parallel \text{MerkleRoot}_{K-2}),$$

and checks if this hash is registered in the `FriStatementContract` instance as a valid fact (an error is raised otherwise).

#### ■ Observation 14: Naming

The label `numLayers` is used for the total number of FRI steps, *i.e.* the sum of the steps of all the layers, which we denote  $s_{K-1}$  in this report (and which is strictly greater than the number of FRI layers). This is confusing.

**Recommendation:** We recommend to change this name, *e.g.* for `totalNumSteps`.

#### Status: Resolved (✓)

The label has been renamed into `sumOfStepSizes`.

### ● Observation 15: Costly exponentiations

The piece of code from line 52 to line 56 makes two calls to `fpow` in order to compute  $v^{2^{s_K-1}}$  from  $1/v$  (raising to the  $2^{s_K-1}$  first and then to the group order minus 1). These costly exponentiations could be avoided.

**Recommendation:** An other way would be to use the index of  $v^{2^{s_K-1}}$  (which can be easily derived from the index of  $v$ ) and to raise a generator of  $L'_{K-1}$  (see Section 2.1) to the exponent corresponding to the index. This would require to pre-compute a generator of  $L'_{K-1}$  once, and then would result in a much smaller exponentiation inside the loop.

### Status: Resolved (✓)

The recommendation is not applicable since it would require reversing the bits of the index of  $v^{2^{s_K-1}}$ .

*General observations:* Observations 23 (●) and 24 (■) are applicable to this contract.

## 3.5 STARK verifier

### 3.5.1 Contract StarkVerifier

The contract `StarkVerifier` is the main contract of the STARK verifier which is derived from the other contracts and performs the global STARK verification of an input proof of computational integrity. The entry point of this contract is the function `verifyProof` which takes three inputs:

- the proof parameters `proofParams` (see Section 2.4.1);
- the proof `proof` (formatted as described Section in 2.4.2);
- the statement `publicInput` which must be verified thanks to the proof.

The proof verification proceeds as follows:

- It calls the function `initVerifierParams`. This function checks if the proof parameters are valid (the parameter checkings are described in Section 2.4.2), and initializes all the fields of the verifier context related to the statement and the proof parameters. It also checks that the trace length  $N$  is consistent with `fri_step_list[.]` and with the degree of the last layer polynomial  $p_{K-1}$  and that the soundness error

$$nQueries \cdot \log_2 \beta + proofOfWorkBits$$

achieves the target security level `numSecurityBits`.

- It initializes the verifier channel with the proof.

- It reads the trace commitments from the proof (and samples the corresponding challenges).
- It samples the OODS point  $z$ , reads the OODS values  $\{y_\ell\}_\ell \cup \{\hat{y}_i\}_i$  and calls the function `oodsConsistencyCheck`. As described in Section 2.5.3, this function (implemented by the user) must check that the mask  $\{y_\ell\}_\ell$  for the point  $z$  is consistent with all the  $\hat{y}_i := h_i(z^{M_2})$ .
- After sampling the OODS coefficients  $\{\gamma_0, \dots, \gamma_{M_1+M_2-1}\}$  which defines the first layer polynomial  $p_0$  (*i.e.* the DEEP composition polynomial), it reads from proof the commitments for all the layer polynomials  $p_0, \dots, p_{K-2}$  except the last one.
- It calls the function `readLastFriLayer`. This function reads the coefficients of the polynomial  $p_{K-1}$ , checks if they are in  $\{0, \dots, p-1\}$  and stores a pointer to them in the verifier context (field `MM_FRI_LAST_LAYER_PTR`).
- It checks if the proof of work is consistent with the current PRNG state (by calling `VerifierChannel.verifyProofOfWork`).
- It generates all the (initial) queries  $Q_0$  for the FRI protocol. And then, it calls the function `computeFirstFriLayer`, which
  - correctly formats the queries in  $Q_0$  in the FRI queue thanks to  
`adjustQueryIndicesAndPrepareEvalPoints;`
  - reads  $f_0(v), \dots, f_W(v)$  and  $h_0(v), \dots, h_{M_2-1}(v)$  for every  $v \in Q_0$  from the proof and checks these values are consistent with the trace commitments (see Section 2.2.2) thanks to `readQueryResponsesAndDecommit`.
  - calls the OODS contract (see section 2.5.2) to compute  $p_0(v)$  from the  $\{f_i(v)\}$  and  $\{h_i(v)\}$  for every  $v \in Q_0$  (these values are stored in the FRI queue).
- Finally, it launches the processing and verification of the FRI layers thanks to the function `Fri.friVerifyLayers`.

#### ● Observation 16: Magic numbers

Many constant values are hardcoded in the checks of `initVerifierParams` (while some checks make use of properly defined constants).

**Recommendation:** We recommend to avoid such magic numbers by defining constants.

#### Status: Partially resolved (~)

Those remain as a magic numbers. Some comments have been added which explain that some bounds are somewhat arbitrary.

**● Observation 17: Check for the number of FRI steps**

The auxiliary function `validateFriParams` checks that the number of steps of a layer is in the interval  $\{1, \dots, 4\}$ . The interval maximum 4 is hardcoded instead using the constant `FRI_MAX_FRI_STEP`. The interval minimum is 1, but this case is not supported by the function `FriLayer.doFriSteps`.

**Recommendation:** We recommend to define a constant for the minimum number of FRI steps, which is 2, and to use this constant as well as the defined constant `FRI_MAX_FRI_STEP` in the check.

**Status: Resolved (✓)**

The interval minimum is now 2 (hardcoded value) and the interval maximum is now set using the constant `FRI_MAX_STEP_SIZE`.

**■ Observation 18: Naming**

The proof parameter defined in the constant

`PROOF_PARAMS_FRI_LAST_LAYER_DEG_BOUND_OFFSET`

is not the upper bound of the degree of  $p_K$  but of its logarithm.

**Recommendation:** We recommend to rename this constant

`PROOF_PARAMS_FRI_LAST_LAYER_LOG_DEG_BOUND_OFFSET`

or some other name which makes appear the LOG keyword.

**Status: Resolved (✓)**

The constant has been renamed as suggested.

**● Observation 19: Costly exponentiation**

In the function `adjustQueryIndicesAndPrepareEvalPoints`, the following expression is used to compute a right shift:

`res := div(res, exp(2, sub(127, numberOfBits)))`

Thus it costs 25 units of gas [11] for a simple shift operation.

**Recommendation:** If the targeted EVM version is at least Constantinople<sup>a</sup>, we suggest to use the opcode `shr` which only costs 3 units of gas.

**Status: Resolved (✓)**

The expression has been replaced by

```
res := shr(sub(127, numberOfBits), res).
```

<sup>a</sup><https://docs.soliditylang.org/en/v0.8.10/using-the-compiler.html#target-options>

● **Observation 20: Montgomery vs. standard form for OODS contract**

As explained in Section 2.5.2, the OODS contract takes some inputs in standard form ( $z$ ,  $\{y_\ell\}_\ell \cap \{\hat{y}_i\}_i$  and  $\{\gamma_i\}_i$ ) and some inputs in Montgomery form (the  $f_i$  and  $h_i$  evaluations). Then it converts the Montgomery-form inputs to standard form and produces outputs in standard form.

**Recommendation:** We recommend to homogenize the form of the inputs and outputs of the OODS contract, namely to convert the  $f_i$  and  $h_i$  evaluations before calling the OODS contract.

**Status: Unresolved (X)**

No modification has been made.

*General observations:* Observations 21 (●), 22 (●), 24 (■), 26 (●) and 27 (■) are applicable to this contract.

## 4 General observations

### ● Observation 21: Harcoded constants

In the source code, many constant values are hardcoded even though the corresponding constants are declared. This mainly occurs in inline assembly. For example, the constant `PrimeFieldElement0.K_MODULUS` defines the prime  $p$ , but this constant is not used by the functions `fmul`, `fadd`, `fromMontgomery`, which directly hardcode the corresponding value `0x8000000000000011000[...]0001`. The list of the constants which are hardcoded in some parts of the implementation are:

- `PrimeFieldElement0.K_MODULUS`,
- `PrimeFieldElement0.K_MODULUS_MASK`,
- `PrimeFieldElement0.K_MONTGOMERY_R`,
- `PrimeFieldElement0.K_MONTGOMERY_R_INV`,
- `FriLayer.FRI_MAX_FRI_STEP`.

**Recommendation:** Solidity supports constants in inline assembly

- since version 0.5.11 (2019-08-12): “Inline Assembly: Support direct constants of value type in inline assembly.” [3], see also [1].
- since version 0.5.14 (2019-12-09): “Inline Assembly: Support constants that reference other constants.” [3], see also [2].

Since the pragma for Solidity version is configured as

```
pragma solidity ^0.6.11;
```

we recommend to avoid the hardcoding of constant values (in inline assembly) but to rely on declared constants.

**Status: Resolved (✓)**

All these hardcoded values have been replaced by the corresponding constants.

### ● Observation 22: Unused functions and constants

The following functions and constants are never used:

- `PrimeFieldElement0.GEN1024_VAL`
- `PrimeFieldElement0.fromMontgomeryBytes`

- `PrimeFieldElement0.toMontgomeryInt`
- `Prng.mixSeedWithBytes`
- `Prng.getPrngDigest`
- `VerifierChannel.CHANNEL_STATE_SIZE`
- `StarkVerifier.hashRow`
- `FriLayer.nextLayerElementFromTwoPreviousLayerElements`

**Recommendation:** If the above constants and functions aim to be called in external contracts, we recommend to add a comment to clarify it. Otherwise, we recommend to remove them.

**Status: Partially resolved (~)**

All these functions and constants have been removed, except `fromMontgomeryBytes` and `toMontgomeryInt`.

#### ● Observation 23: Auxiliary functions defined as internal

The following functions are auxiliary (*i.e.* used to avoid redundancy in the corresponding contract and/or to improve the code readability):

- `PrimeFieldElement0.expmod`,
- `Prng.getRandomBytesInner`,
- `FriLayer.doXFriSteps` with  $X \in \{2, 3, 4\}$ ,
- `FriLayer.gatherCosetInputs`,
- `FriLayer.doFriSteps`,
- `Fri.verifyLastLayer`.
- `FriStatementVerifier.computerLastLayerHash`.

**Recommendation:** Since these functions do not aim to be used in other contracts, we suggest to define them as private functions.

**Status: Unresolved (X)**

All these functions are still defined as internal.

**■ Observation 24: Pointer variables**

Pointer variables are often indicated by the suffix `Ptr` but not always. For instance, the following variables are pointers:

- `coefsStart` in `HornerEvaluator`, `Fri` and `FriStatementVerifier`
- `friCtx` in `FriLayer`, `Fri` and `FriStatementContract`
- `friQueue` in many places
- `friQueueTail` in `FriLayer`
- `friQueueHead` in `FriLayer`
- `friQueueEnd` in `StarkVerifier` and `FriLayer`
- `dataToHash` in `FriStatementContract`

**Recommendation:** Use suffix `Ptr` for all pointer variables.

**Status: Unresolved (X)**

All these pointer names have remained unchanged.

**● Observation 25: Homogenize the call to constants of base contracts**

To refer to a specific constant in the source code, it is sometimes written

`BaseContract.ConstantName`

(e.g. line 21 of `HornerEvaluator.sol`) and at other locations, it is just written

`ConstantName`

(e.g. line 157 of `StarkVerifier.sol.ref`).

**Recommendation:** We recommend to always use the same notation (without the contract name if there is no ambiguity).

**Status: Unresolved (X)**

The recommendation has not been fixed for the sake of readability.

**● Observation 26: Redundant code**

In multiple places in the code, some functions are re-implemented.

- In the contract `VerifierChannel`,



- `sendFieldElements` reimplements `getRandomBytes`.
- `readBytes` reimplements `mixSeedWithBytes`.
- In the contract `StarkVerifier`, the function `adjustQueryIndicesAndPrepareEvalPoints` reimplements
  - `bitReverse` (from `FriLayer`), and
  - `expmod` (from `PrimeFieldElement0`).

**Recommendation:** We recommend as much as possible to call the implemented functions. It makes the code easier to maintain and less prone to bugs.

**Status: Unresolved (X)**

The recommendation has not been applied since it is impossible to call global functions in assembly. However, the use of assembly does not seem always justified. For example, it is not clear why the implementation of `VerifierChannel.sendFieldElements` is entirely written in assembly. For this function, the assembly language seems useful only when storing the sampled value at `targetPtr`.

■ **Observation 27: Ambiguity in the meaning of “step” in the code**

The notion of “step” is used in different variable names for two different notions:

- A step of the FRI protocol (which divides the polynomial degree by 2) where steps are grouped by 2, 3, or 4 in one layer. This is *e.g.* the terminology suggested by the functions `doXFriSteps` and more generally in the `FriLayer` contract. This is also the terminology we use in this report, as introduced in Section 2.1 and according to the ethSTARK documentation [10].
- In the functions
  - `FriStatementVerifier.friVerifyLayers` (variable `nFriStepsLessOne`)
  - `StarkVerifier.initVerifierParams` (variable `nFriSteps`)

the usage of “step” refers to the notion of layer.

**Recommendation:** We recommend to clearly define the notion of “step” and to fix the variable names accordingly.

**Status: Partially resolved (~)**

Some variables have been renamed (`friStep` into `friStepSizes`, `nFriStepsLessOne` into `nFriInnerLayers`, ...). The notion of “step” then seems to refer to the transition between two layers. There is only one step between two layers, but this step can have

several sizes (2, 3 or 4). However, some names remain inconsistent with this choice (doXFriSteps for  $X \in \{2, 3, 4\}$ , doFriSteps).

## References

- [1] Constant variables in inline assembly. Solidity Issues on GitHub, 2018. <https://github.com/ethereum/solidity/issues/3776>.
- [2] Support referencing another constants in inline assembly. Solidity Pull Requests on GitHub, 2019. <https://github.com/ethereum/solidity/pull/7874>.
- [3] Solidity changelog. Solidity Repository on GitHub, 2021. <https://github.com/ethereum/solidity/blob/develop/Changelog.md>.
- [4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPICs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. <https://ia.cr/2018/046>.
- [6] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. Cryptology ePrint Archive, Report 2016/116, 2016. <https://ia.cr/2016/116>.
- [7] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. <https://ia.cr/2021/1063>.
- [8] Kineret Segal and Gideon Kaempfer. Starkdex deep dive: the stark core engine. Medium, 2019. <https://medium.com/starkware/starkdex-deep-dive-the-stark-core-engine-497942d0f0ab>.
- [9] StarkWare. Stark math. Medium, 2019. <https://medium.com/starkware/tagged/stark-math>.
- [10] StarkWare. ethstark documentation. Cryptology ePrint Archive, Report 2021/582, 2021. <https://ia.cr/2021/582>.
- [11] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger (rev. 2021-12-02), 2021. <https://ethereum.github.io/yellowpaper/paper.pdf>.

## A Formal checking of doXFriSteps

As explained in Section 3.4.1, the functions **doXFriSteps** with  $\mathbf{X} \in \{2, 3, 4\}$  compute an evaluation  $p_{i+1}(v_{i+1})$  from the  $2^{\ell_{i+1}}$  evaluations  $p_i(c \cdot \xi^{j_0}), p_i(c \cdot \xi^{j_1}), \dots$  where

- $j_0, j_1, \dots$  are in increasing bit-reverse order, *i.e.*

$$j_r = \text{bit-reverse}_{\log |L_i|}(r) .$$

- $\xi$  is a generator of the subgroup of  $\mathbb{F}_p^\times$  of size  $2^{\ell_{i+1}}$ .

As written in Equation 3, the polynomial  $p_{i+1}$  is built as

$$p_{i+1}(x) = \sum_{j=0}^{2^{\ell_{i+1}}-1} \zeta_i^j \cdot p_i^{[j]}(x)$$

where the  $\{p_i^{[j]}\}_j$  are the polynomials defined such that

$$p_i(x) = \sum_{j=0}^{2^{\ell_{i+1}}-1} x^j \cdot p_i^{[j]}(x^{2^{\ell_{i+1}}}) .$$

We have the following relation

$$\begin{pmatrix} p_i(c \cdot \xi^{j_0}) \\ p_i(c \cdot \xi^{j_1}) \\ p_i(c \cdot \xi^{j_2}) \\ \dots \end{pmatrix} = \begin{pmatrix} 1 & c \cdot \xi^{j_0} & (c \cdot \xi^{j_0})^2 & \dots \\ 1 & c \cdot \xi^{j_1} & (c \cdot \xi^{j_1})^2 & \dots \\ 1 & c \cdot \xi^{j_2} & (c \cdot \xi^{j_2})^2 & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} p_i^{[0]}(v_{i+1}) \\ p_i^{[1]}(v_{i+1}) \\ p_i^{[2]}(v_{i+1}) \\ \dots \end{pmatrix}$$

since  $c^{2^{\ell_{i+1}}} = v_{i+1}$ . Thus,

$$\begin{pmatrix} p_i^{[0]}(v_{i+1}) \\ p_i^{[1]}(v_{i+1}) \\ p_i^{[2]}(v_{i+1}) \\ \dots \end{pmatrix} = \frac{1}{2^{\ell_{i+1}}} \begin{pmatrix} 1 & 1 & \dots \\ (c \cdot \xi^{j_0})^{-1} & (c \cdot \xi^{j_1})^{-1} & \dots \\ (c \cdot \xi^{j_0})^{-2} & (c \cdot \xi^{j_1})^{-2} & \dots \\ \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} p_i(c \cdot \xi^{j_0}) \\ p_i(c \cdot \xi^{j_1}) \\ p_i(c \cdot \xi^{j_2}) \\ \dots \end{pmatrix} .$$

So we have

$$2^{\ell_{i+1}} \cdot p_{i+1}(v_{i+1}) = \begin{pmatrix} 1 \\ \zeta_i/c \\ (\zeta_i/c)^2 \\ \dots \end{pmatrix}^T \begin{pmatrix} 1 & 1 & \dots \\ (\xi^{j_0})^{-1} & (\xi^{j_1})^{-1} & \dots \\ (\xi^{j_0})^{-2} & (\xi^{j_1})^{-2} & \dots \\ \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} p_i(c \cdot \xi^{j_0}) \\ p_i(c \cdot \xi^{j_1}) \\ p_i(c \cdot \xi^{j_2}) \\ \dots \end{pmatrix} .$$

The formal checking we applied to verify the correctness of the functions **do2FriSteps**, **do3FriSteps** and **do3FriSteps** compares the outputs of these functions with the above formula.