STARKWARE

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 STARK/Cairo verifier. We chose to work with CryptoExperts based on our good experience and interaction with them in the past.

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	-	-	-	-
Medium risk	-	-	-	-
Low risk	-	-	-	-
Coding Practices	10	-	-	10
Documentation	10	-	-	10
Total	20	-	-	-

Contents

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. As part of StarkWare's technology, the Cairo STARK-friendly CPU architecture allows developers to easily write STARK-provable programs for general computation.

In this context, CryptoExperts has conducted a first audit of StarkWare's STARK verifier (implemented as a Solidity smart contract) in late 2021 as well as a second audit of their Cairo verifier and associated shared proof service (SHARP), which are built on top of the STARK verifier (together with the Cairo code of the bootloader) in March and April 2022.

StarkWare was then willing to perform a follow-up audit of the Cairo implementation of their STARK/Cairo verifiers, which is used for recursive STARK proofs. Upon business agreement with StarkWare, CryptoExperts has conducted this third audit in October and November 2022. The primary goal of this audit was to ensure that the Cairo code correctly verifies the computational integrity proof of a Cairo program. The service consisted of a study of the STARK & Cairo documentation and a review of the code by two engineers, experts in cryptography. The present report contains the results of this audit.

We first identify the audited code (Section 1.1) and give a summary of the audit methodology and findings (Section 1.2). We then provide some complementary documentation for the audited code in Section 2. The results of the audit are presented in Section 3 which is structured according to the functionalities of the underlying source files: verifier channel (§ 3.1), commitments (§ 3.2), FRI protocol (§ 3.3), core STARK verifier (§ 3.4), public input (§ 3.5), layout constraints (§ 3.6), DEEP composition polynomial (§ 3.7), and Cairo verifier (§ 3.8).

1.1 Audited code

The audited source code corresponds to a STARK/Cairo verifier written in the Cairo programming language. This code aims to verify a STARK proof of the computational integrity of a Cairo program. Given a proof, a program bytecode, and a program output, it verifies that the prover knows a program input for which the considered program produces the considered output. Such a Cairo implementation of the STARK/Cairo verifier aims to be used for recursive proofs.

The audited code is composed of 41 Cairo source files in the following repos:

- `https://github.com/starkware-libs/cairo-lang/tree/master/src/starkware/cairo/stark_verifier`
- `https://github.com/starkware-libs/cairo-lang/tree/master/src/starkware/cairo/cairo_verifier`

excluding files named `autogenerated.cairo` and `periodic_columns.cairo` which are auto-generated. The version of the code to be reviewed corresponds to the commit `d61255f`, "Cairo v0.10.0.", from September 5, 2022. Those 41 source files contain about 4400 lines of code written in Cairo language.

1.2 Methodology and summary of findings

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

- to confirm that the implemented verification process is compliant with the specification of the STARK/Cairo verifier,
- to check the absence of flaws 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.

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.

General remark. We were provided with a clear set of documentation about the STARK protocol and Cairo [6, 4, 5, 3, 7] but sometimes missed detailed specifications for the audited code. 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 present report. Part of it is given in Section 2 while the rest appears in the preamble of each sub-section of Section 3.

Summary of findings on the Cairo implementation of STARK/Cairo verifier. Our findings are summarized in the table below. Our observations are only related to coding practices and documentation. We did not detect any flaw that could affect the soundness of the verifier. We therefore conclude that, up to the limitations inherent to human code review, the code is a sound implementation of the STARK/Cairo verifier.

Category	**Number of findings**
High risk	0
Medium risk	0
Low risk	0
Coding practices	10
Documentation	10
Total	**20**

2 Complementary documentation

This section provides some complementary documentation which we inferred from the audited code (and from our previous reports). It includes a reminder of useful notions and notations (§ 2.1), some details about the verifier architecture (§ 2.2) and the input format (§ 2.3.1), and a list of the external dependencies which are out of scope of this audit (§ 2.5).

2.1 Notations

We recall hereafter the notions and notations which are relevant to this report. For the STARK verifier, we use the notations and terminologies introduced in the ethSTARK documentation [6] and additional notations introduced in our previous report on the STARK verifier [1]. For the CPU verifier, we use the notations and terminologies introduced in the Cairo article [3] and additional notations introduced in our previous report on the Cairo verifier [2].

2.1.1 STARK verifier

- 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 L, where L is a power of two. In case *Randomized AIR with Preprocessing* [3] is used (which is always the case for Cairo proofs), we denote W' the number of columns after the sampling of *interaction elements*.
- The values in the trace cells are elements of the finite field $\mathbb{F}_p$ with p a prime. In practice, the prime p is defined as

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

- The *Trace Evaluation Domain* is defined as a multiplicative subgroup $\langle g \rangle$ of $\mathbb{F}_p^\times$ of size L.
- Each trace column is interpreted as the L point-wise evaluations of a polynomial of degree smaller than L over the trace evaluation domain. These polynomials are referred to as the *Trace Column polynomials* and are denoted by $f_0, \ldots, f_{W-1}$.
- 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

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

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

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

- 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 $\{\gamma_0, \ldots, \gamma_{M_1+M_2-1}\}$ are random coefficients sampled by the IOP verifier, called *OODS Coefficients*.
- To achieve a secure protocol, the polynomials $f_0, \ldots, f_{W-1}$ and $h_0, \ldots, h_{M_2-1}$ 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 $|L_0|/L$ (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 β. In practice, L_0 is a non-unit coset of the multiplicative subgroup $\langle g_{\text{LDE}} \rangle \subseteq \mathbb{F}_p^\times$ of size βL. That is L_0 is defined as

$$L_0 := c_{\text{LDE}} \cdot \langle g_{\text{LDE}} \rangle$$

  for some $c_{\text{LDE}} \neq 1$.
- The FRI protocol is composed of different *layers*. We denote K the number of FRI layers. Except for the last one, each layer performs several *FRI steps*. For the ith layer, the number of steps is denoted ℓ_i, which is stored in a list named `fri_step_sizes[]`. Namely,

$$\ell_i := \texttt{fri\_step\_sizes[}i\texttt{]} \ .$$

(This list is named `fri_step_list` in the Solidity implementation of the verifer [1].) We further denote by s_i the total number of steps from layers 1 to i, that is

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

- At the ith 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

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

  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 $i \geq 0$,

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

  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 $\{p_i\}_i$ are 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 $\mathsf{index}_i(x)$ defined such as

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

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

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

- 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 4 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

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

2.1.2 Cairo verifier

- The Cairo machine has three registers:
  - the *program counter*, denoted `pc`, which contains the memory address of the next Cairo instruction to be executed,
  - the *allocation pointer*, denoted `ap`, which (by convention) points to the first memory cell that has not been used by the program so far, and
  - the *frame pointer*, denoted `fp`, which (by convention) points to the beginning of the stack frame of the current function.
- The CPU has access to a *nondeterministic continuous read-only memory* $\mathtt{m}$.
- The Cairo machine relies on *builtins* which offer a way to efficiently perform some specific computation tasks. A builtin aims to perform a specific computation that involves several field elements. A builtin *instance* is a set of those field elements and the *size* of the instance is defined as the size of this set. Each builtin is assigned to a *memory segment* which is split into small chunks. Each chunk corresponds to a builtin instance, and the values stored on a chunk must satisfy the relation of the builtin.

- The execution of a program can be represented as a sequence of states of the three registers $(\mathtt{pc}_i, \mathtt{ap}_i, \mathtt{fp}_i)_i$ and a memory function $\mathtt{m} : \mathbb{F}_p \to \mathbb{F}_p$. We denote T the number of steps (or state transitions) in the program execution and

$$N := T + 1$$

the number of states. Each state of the Cairo machine is represented in the execution trace by a small set of successive rows called *component*. The trace length L (*i.e.* its total number of rows) satisfies the relation

$$L = \texttt{CPU\_COMPONENT\_HEIGHT} \cdot N$$

where `CPU_COMPONENT_HEIGHT` is the number of rows in a component.

- We call *layout* the organization of all the variables in the execution trace. A layout depends on which builtins are supported, but it also depends on the configuration of those builtins.

2.2 Architecture of the verifier

2.2.1 Overview of the verification process

The STARK proof system is a non-interactive version of an IOP protocol relying on the FRI protocol. The underlying interactive protocol is depicted in Figure 1.

The STARK proof corresponds to a non-interactive version of this protocol using the Fiat-Shamir heuristic. In this paradigm, each verifier-to-prover arrow corresponds to some public-coin challenges obtained through pseudo-random generation from a seed, where the latter seed is obtained by hashing the previously received elements from the prover. On the other hand, each prover-to-verifier arrow corresponds to a commitment or a response to a (pseudo-randomly generated) challenge. Along the verification process, the verifier reads the prover-to-verifier elements from the proof, updates a hash digest for the Fiat-Shamir heuristic (the *verifier channel*), and pseudo-randomly generates the verifier-to-prover elements from this hash seed. Different verification steps are then applied to the received elements (read from the proof) and the sent elements (pseudo-randomly generated).

2.2.2 Three verification levels

The audited verifier has been organized with the following three levels.

Core STARK verifier. This bottom level implements the core of the STARK verification process. It does not deal with the format of execution traces, the AIR constraints, and the public input (*i.e.* the statement to be verified). For this reason, using STARK core further requires to:

- functions to validate the trace format, commit the traces and decommit them,
- functions to validate and hash the public input,

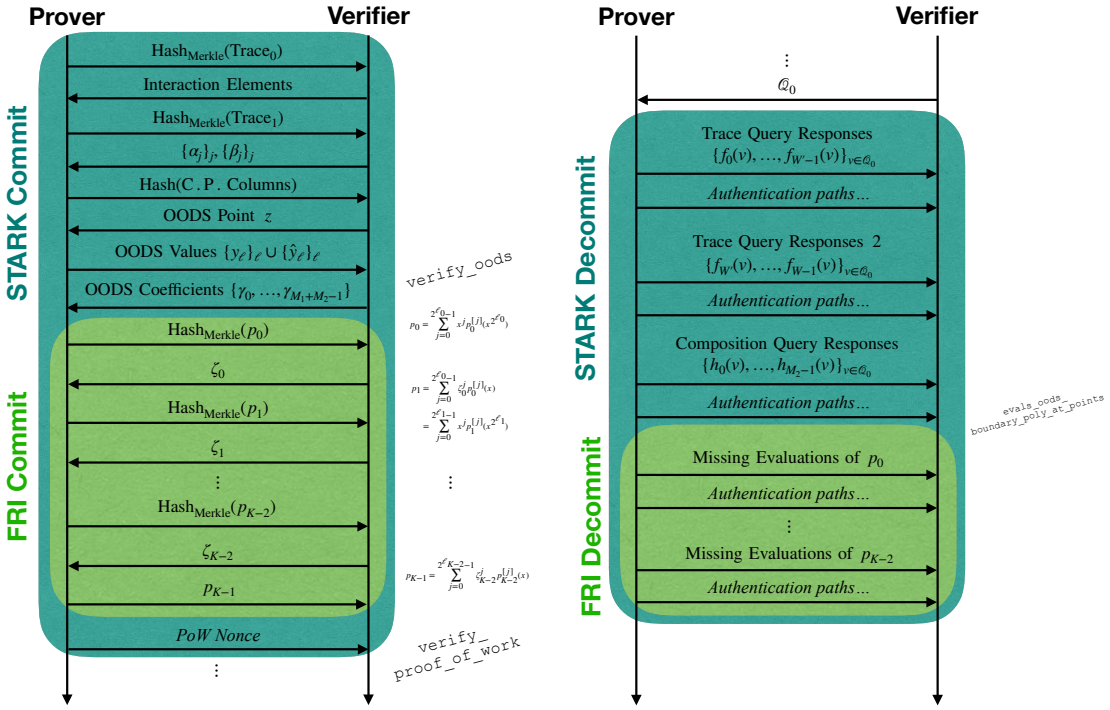

Figure 1: STARK interactive protcol

- functions to evaluate the composition polynomial and the DEEP composition polynomial (which depend on the AIR constraints).

From those functions, the STARK core can check that the proof is consistent with the statement. The source files of the STARK verifier are available in

`stark_verifier/core`

and its entry point is the function

`verify_stark_proof(air, proof, security_bits)`

of the file `stark_verifier/core/stark.cairo`.

CPU verifier (layout definition). This middle level specifies the above verifier by defining the format of the execution trace, the public input, and the layout (the set of AIR constraints corresponding to a specific Cairo machine). Specifically, it implements the functions required by the core STARK verifier. The public memory is represented by a *main page* and a set of *continuous pages*. The organization and the content of those

pages is not defined at this level. When calling the CPU verifier, one needs to previously fill those pages (for example, with the program bytecode). The source files of the CPU verifier are available in

`stark_verifier/air`

and its entry point is the function

`verify_proof(proof, security_bits)`

of the file `stark_verifier/air/layouts/xxxx/verify.cairo`, where `xxxx` is a CPU layout.

Cairo verifier (page definition). This top level defines the content of the pages of the public memory which shall be used in the CPU verifier. In the context of the Cairo implementation of the verifier, the full public memory (including the memory mapping about the output segment) is defined by the main page and there is no continuous page. The source files of the Cairo verifier are available in

`cairo_verifier`.

and its entry point is the function

`verify_cairo_proof(proof)`

of the file `cairo_verifier/layouts/xxxx/cairo_verifier.cairo`, where `xxxx` is a CPU layout.

Remark 1. *Continuous pages are used in the context of the SHARP verifier which verifies several Cairo programs, or* tasks*, loaded and executed through the* Cairo bootloader *[2]. The output of each task is stored in a separate continuous page, which enables independent validation of each task from the global proof. The purpose of the Cairo implementation of the verifier is not to verify several tasks at the same time but a single one, which is why the number of continuous pages is set to zero in the current implementation.*

2.2.3 Architecture of the code

The source code can be organized under several bundles of source files corresponding to different functionalities:

- the files which implement the *verifier channel* (*i.e.* the state of the IOP verifier in the Fiat-Shamir heuristic), dealing with proof reading and challenge sampling,
- the files which manage Merkle tree *commitments* (and decommitments),
- the files which implement the *FRI protocol*,
- the files which implement the *core STARK verifier*,

- the files which manage the *public input*,
- the files dealing with the *AIR constraints* corresponding to the *CPU layout*,
- the file which implements the *DEEP composition polynomial* (or OODS polynomial),
- the files which implement the *Cairo verifier* (as a specialization of the core STARK verifier).

This organization is further illustrated in Figure 2. The STARK verification level is covered by the *channel verifier*, the *commitments* (excluding the trace commitments), the *FRI protocol*, and the *core STARK verifier*. The CPU verification level is covered by the *trace commitments*, the *(CPU) public input*, the *layout constraints*, the *DEEP composition polynomial*, and part of the *Cairo verifier* (the files in `stark_verifier`). Finally, the rest of the *Cairo verifier* (the files in `cairo_verifier`) corresponds to the Cairo verification level.

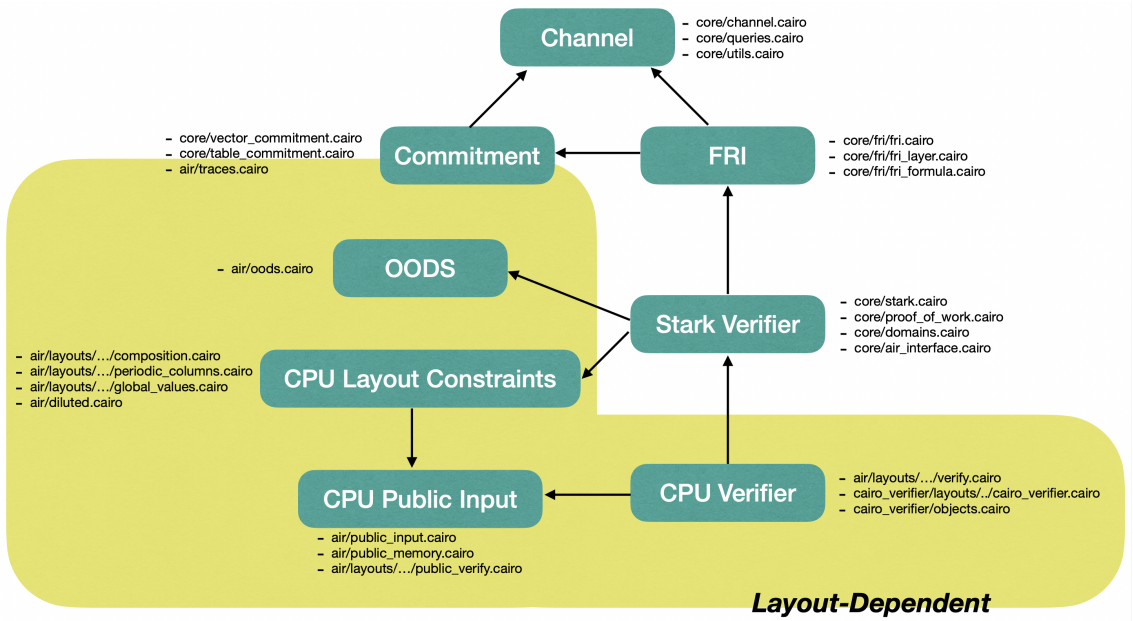

Figure 2: Architecture of the source code

Call graph. In Figure 3, we represent the call graph of the main routines in the source code.

2.3 Input format

We detail hereafter the format of

- the *public input* which stores the different parameters of the Cairo statement to be verified,

- the *proof* which is the input of the verifier. *NB: In the audited code, the public input is part of the proof.*

2.3.1 Public input

The format of the public input is defined by the structure `PublicInput` in the source file `stark_verifier/air/public_input.cairo`. The fields of this structure are described in Table 1.

Field		Description
`log_n_steps`		Log (in base 2) of the number of states N
`rc_min`		Min range-check value
`rc_max`		Max range-check value
`layout`		The layout identifier
`n_segments`		Number of segments
`segments[0].begin_addr`		First address of the segment 0
`segments[0].stop_ptr`		Final position of the segment-0 pointer
`segments[1].begin_addr`		First address of the segment 1
`segments[1].stop_ptr`		Final position of the segment-1 pointer
...		...
`padding_addr`		a_{pad}
`padding_value`		$\mathtt{m}(a_{\text{pad}})$
`main_page_len`		Size of the main page
`main_page`		Main page
	0	$\mathtt{m}^*(\mathtt{pc}_I) =$ `0x40780017fff7fff`
	1	$\mathtt{m}^*(\mathtt{pc}_I + 1) =$ Number of program builtins
	2	$\mathtt{m}^*(\mathtt{pc}_I + 2) =$ `0x1104800180018000`
	3	$\mathtt{m}^*(\mathtt{pc}_I + 3) =$ Absolute position of `main`
	4	$\mathtt{m}^*(\mathtt{pc}_I + 4) =$ `0x10780017fff7fff`
	5	$\mathtt{m}^*(\mathtt{pc}_I + 5) =$ `0x0`
	$6 \rightarrow \Delta - 2$	$\mathtt{m}^*(\mathtt{pc}_I + 6) \dots \mathtt{m}^*(\mathtt{fp}_I - 3) =$ Program Bytecode
	$\Delta - 1$	$\mathtt{m}^*(\mathtt{fp}_I - 2) = \mathtt{fp}_I$
	Δ	$\mathtt{m}^*(\mathtt{fp}_I - 1) = 0$
	$\Delta + 1 \rightarrow \Delta + n$	$\mathtt{m}^*(\mathtt{ap}_I) \dots \mathtt{m}^*(\mathtt{ap}_I + n - 1) = \mathtt{ptr}_I^{[1]} \dots \mathtt{ptr}_I^{[n]}$
	$\Delta + n + 1 \rightarrow \Delta + 2n$	$\mathtt{m}^*(\mathtt{ap}_F - n) \dots \mathtt{m}^*(\mathtt{ap}_F - 1) = \mathtt{ptr}_F^{[1]} \dots \mathtt{ptr}_F^{[n]}$
	$\Delta + 2n + 1 \rightarrow \Delta + 2n + n'$	$\mathtt{m}^*(\mathtt{ptr}_I^{[\text{output}]}) \dots \mathtt{m}^*(\mathtt{ptr}_F^{[\text{output}]} - 1) =$ Outputs
`n_continuous_pages`		Must be *zero*
`continuous_page_headers`		*Not used*

Table 1: Format of the public input. For each builtin i, $\mathtt{ptr}_I^{[i]}$ and $\mathtt{ptr}_F^{[i]}$ are the initial and final values of the corresponding builtin pointer. We define $\Delta := \mathtt{fp}_I - \mathtt{pc}_I - 1$.

Let us remark that the order of the segments in the public input is defined by the names-

pace `segments` in the file `stark_verifier/air/layouts/xxx/public_verify.cairo` for `xxx` being the corresponding CPU layout. Table 1 exhaustively lists the memory mapping for the main page as it should appear (in the same order) in the public input. Moreover, the number of continuous pages should be set to 0 (see Remark 1 below).

2.3.2 Proof format

The format of the proof is defined by the structure `StarkProof` in the source file `stark.cairo`:

```
struct StarkProof {
    config: StarkConfig*,
    public_input: PublicInput*,
    unsent_commitment: StarkUnsentCommitment*,
    witness: StarkWitness*,
}
```

It is made of four components:

- the STARK configuration (member `config` of `StarkProof`): structure containing some parameters for the commitments, the FRI protocol, and the proof of work;
- the public input (member `public_input` of `StarkProof`): as described above;
- the commitments sent by the IOP prover (member `unsent_commitment` of `StarkProof`): the trace commitments (before and after interaction), the composition polynomial trace commitment, the OODS values, the FRI commitments (including the coefficients of the last layer polynomial) and the nonce of the proof of work;
- the witness sent by the IOP prover in the decommitment phase (member `witness` of `StarkProof`): the decommitted values and their authentication paths of the Merkle trees to check decommitment.

Besides configuration data and the public input described above, Table 2 gives an overview of the different elements of the `unsent_commitment` and `witness` members of the `StarkProof` structure. These elements can be thought of as the prover-to-verifier transcript of the IOP protocol.

Montgomery representation. All the decommitted field elements in the proof are in *standard form* while their Merkle commitments are computed from their *Montgomery form* (*i.e.* scaled by a factor $R := 2^{256} \bmod p$).[1] Moreover, all the field elements sampled and hashed by the IOP verifier (in the Fiat-Shamir heuristic) are considered to be in the Montgomery form and are then converted to the standard form by the verifier.[2] This choice is presumably made because the prover performs all its computation in the Montgomery form (for the sake of efficiency).

[1] See the function `to_montgomery` of the file `table_commitment.cairo` which translates the proof values from standard to Montgomery form before decommitting them (*i.e.* before verifying their Merkle authentication paths).

[2] See `random_felts_to_prover`, `read_felt_from_prover`, `read_felt_vector_from_prover_inner` of the file `channel.cairo`.

Field	Description
Elements composing the `unsent_commitment` *member*	
"Original" Trace Commitment	Merkle root where leaves are evaluations in the LDE of the trace column polynomials before interaction.
"Interaction" Trace Commitment	Merkle root where leaves are evaluations in the LDE of other trace column polynomials after interaction.
OODS Commitment	Merkle roots where leaves are evaluations in the LDE of the composition polynomial columns.
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 *scaled* 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).
Elements composing the `witness` *member*	
"Original" Trace Query Responses	The queried rows of the trace columns before interaction.
"Original" Trace Query Witness	Authentication paths in the Merkle tree of the Trace Commitment to decommit the original trace query responses.
"Interaction" Trace Query Responses	The queried rows of the trace columns after interaction.
"Interaction" Trace Query Witness	Authentication paths in the Merkle tree of the "Interaction" Trace Commitment to decommit the interaction trace query responses.
Composition Query Responses	The queried rows of the composition polynomial columns.
Composition Query Witness	Authentication paths in the Merkle tree of the OODS Commitment to decommit the composition query responses.
Repeat the two last rows for each FRI layer (but the last one), i.e. for layer $i \in \{1, \ldots, K-1\}$.	
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 $\mathcal{Q}_i$.
Witness for Opened Evaluations	Authentication paths in the Merkle tree with the ith FRI Commitment as root to validate the Evaluation Values of the current layer.

Table 2: Prover-to-verifier elements of the STARK proof (besides configuration and public input).

Hash functions. The audited Cairo implementation of the STARK/Cairo verifier makes use of two hash functions: Blake2s and the Pedersen hash function. The former is implemented by an external Cairo library while the latter is a Cairo builtin. The Pedersen hash function is used to hash the main page of the public input, as well as for the hash of the program bytecode and the hash of the program output (which are both output by the Cairo verifier in case of a verification success). All the other hashes are based on Blake2s. The Merkle commitments use a truncated version of Blake2s, only keeping the 160 most significant bits of the hash digest, while the other hash calls (verifier channel, public input hash, proof of work) use the standard version of Blake2s.

2.4 Differences with the Solidity implementation

The audited Cairo implementation of the verifier has a few noticeable differences with the audited Solidity implementation of the verifier [1, 2]:

1. the form of the field elements in the proof: the Cairo implementation assumes ele-

ments in standard form while the Solidity implementation assumes element in Montgomery form,

2. hash functions: the Cairo implementation uses Blake2s (plus Pedersen) while the Solidity implementation uses Keccak,

3. continuous pages: the Cairo implementation fixes the number of continuous pages of the public memory to 0 while the Solidity implementation uses continuous pages for the SHARP verifier (see Remark 1).

These differences (in particular the second one) make the two versions of the verifier incompatible: a proof is necessarily specific to the verifier implementation.

2.5 External dependencies

The audited source code relies on several external Cairo libraries. The implementation of those libraries is out of the scope of the audit. We thus assume their correctness when reviewing the source code of the Cairo verifier. We list below all the used external functions and libraries:

- The functions of `common.math` are used in several places in the source code. They implement arithmetic operations and some assertion utilities.
- The function `pow` from `common.pow` is used in several places of the source code. It computes an exponentiation on $\mathbb{F}_p$.
- The hash function Blake2s (routines of `common.cairo_blake2s.blake2s`) is used in several places of the source code. It is involved in the verifier channel and the Merkle decommitments.
- The function `alloc` from `common.alloc` is used in several places of the source code. It returns a pointer to a free memory segment and deals with allocating the right size for the segment.
- The functions processing 256-bit unsigned integers (represented by a structure `Uint256` of two field values) are used in several places. These routines are defined in the file `common.uint256`.
- The function `usort` from `common.usort` is used in `queries.cairo`. It sorts an array of field elements and removes duplicates.
- The function `memcpy` from `common.memcpy` is used in `oods.cairo`. It copies one memory segment into another.
- The function `get_label_location` from `common.registers` is used in `verify.cairo`, in `public_verify.cairo`, in `cairo_verifier.cairo` and in `fri_layer.cairo`. It returns the memory address corresponding to a label.

- The data structure `HashBuiltin` from `common.hash` is used in `stark.cairo`, in `public_input.cairo`, in `cairo_verifier.cairo` and in `air_interface.cairo`.
- The data structure `BitwiseBuiltin` from `common.cairo_builtins` is used in several places.
- The function `bitwise_and` from `common.bitwise` is used in `utils.cairo`. It computes the bitwise AND between two field values (251-bit values).
- The interface for the Pedersen hash function from the file `common.hash_state` is used in `public_input.cairo` and in `cairo_verifier.cairo`.
- The function `get_fp_and_pc` from `common.registers` is used in `composition.cairo`.
- The data structure `StarkCurve` from `common.ec` is used in `composition.cairo`.

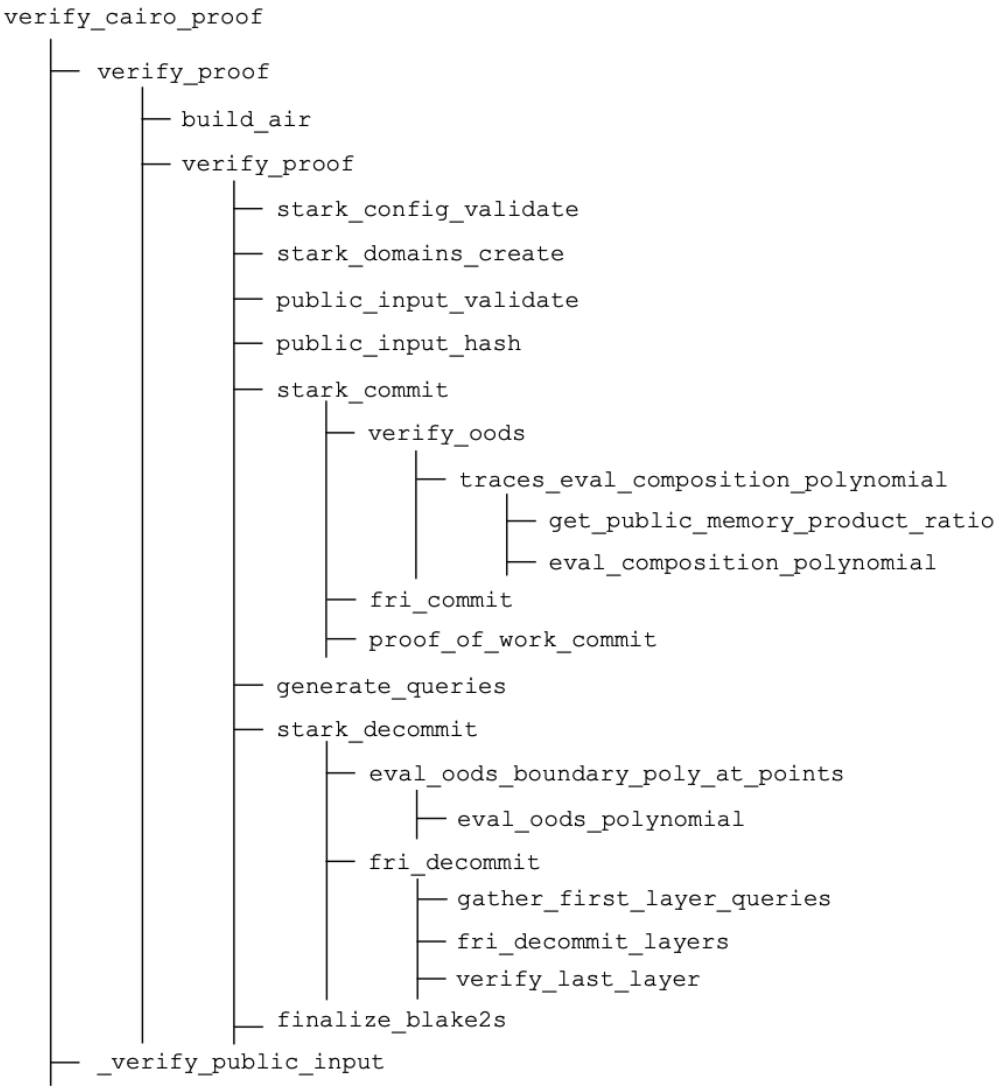

Figure 3: Call graph (main routines)

3 Review of the Cairo verifier

In what follows, we present our review of the source code. We organize this section by grouping the files by functionality as discussed in Section 2.2.3. For each functionality, we first provide some background documentation and then summarize our observations for each file.

3.1 Verifier channel

As illustrated in Figure 1, the IOP verifier receives parts of the proof from the prover and sends some public-coin challenges several times. In the current on-interactive setting, these public-coin challenges are obtained through pseudo-random generation from a seed using the *verifier channel*. The latter is represented by a state (h, c) where h is a hash digest and c is a counter. Whenever the verifier receives a value v (*i.e.* reads a value v from the proof), the channel (h, c) is updated as

$$\begin{cases} h \leftarrow \text{Hash}(h+1, v) \\ c \leftarrow 0 \end{cases}$$

Whenever the verifier needs to sample a value r, it set $r := \text{Hash}(h, c)$ and then increases the counter c. Figure 4 represents this process.

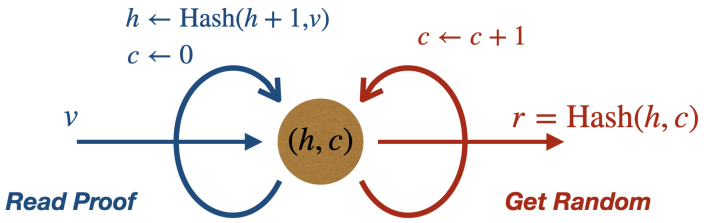

Figure 4: Mechanism of the verifier channel

The types of data that the verifier channel can read from the proof and the types of data that the verifier channel can sample are the following:

Reading	Sampling
unsigned 64-bit integer	unsigned 256-bit integer (`Uint256`)
Field element (`felt`)	Field elements
Field elements	Queries
Vector of field elements	
Truncated Hash (160 bits)	

Note that all the above types are derived from the field element type (`felt`) since all the intermediate values of a Cairo program are encoded as (bunches of) field elements.

The verifier channel is implemented in the file `channel.cairo`, except for the sampling of the queries which is implemented in `queries.cairo`.

3.1.1 File `channel.cairo`

Sent *vs.* unsent elements. Each field element read from the proof is under one of these two forms:

- *unsent* whenever the field element has not been read by the verifier channel – such an element is cast as a `ChannelUnsentFelt` instance;
- *sent* whenever the field element has been read by the verifier channel and can then be used by the verifier – such an element is cast as a `ChannelSentFelt` instance.

At the beginning of the verification process, when the proof is parsed, all the values from the proof are considered as *unsent*. Namely, they are arranged in structures whose base elements are `ChannelUnsentFelt` instances (the `Unsent` is further used for these structures). When the verifier reads some of these values, it involves one of the reading functions from `channel.cairo`. Those functions update the verifier channel and mark the read value as *sent* (*e.g.* it returns an `ChannelSentFelt` instance). This principle does not affect the functionality of the verifier and is presumably used for the sake of code security (by avoiding the risk of using a value from the proof which does not feed the verifier channel, hence breaking the Fiat-Shamir heuristic).

Initialization. Given an input digest, the function `channel_new` returns a new channel with this digest and counter set to zero.

Sampling. The sampling of an unsigned 256-bit integer by the verifier relies on the function `random_uint256_to_prover`. The latter hashes the channel state h and the counter c and returns the 256-bit hash digest after updating the counter of the channel. The implementation assumes that the counter stays below 2^{128} (which means that less than 2^{128} elements have been sampled).

To sample field elements, the function `random_felts_to_prover` calls the above function (at least one call per field element) and performs some rejection sampling. Specifically, it rejects whenever the sampled 256-bit value is greater than $31 \cdot p$ and keeps the value modulo p otherwise (the modulo is direct while going from an `Uint256` to a `felt`). As explained in Section 2.3, the resulting field element is considered to be in the *Montgomery form*. Thus, to return a value in *standard form*, the function further divides the sampled elements by the Montgomery factor.

Proof reading. The file provides several functions to read a value through the channel depending on its type. In contrast with the Solidity implementation of the verifier, the proof reading functions take as input the read value. These functions take *unsent* data as input, update the verifier channel (see Figure 4) and return the same data marked as *sent* (*i.e.* as `ChannelSentFelt` instances). In those reading functions, when $h + 1$ is computed,

only the low order part of h is incremented and an assertion is used to avoid missing the carry (*i.e.* the implementation requires that h always satisfies $h \not\equiv -1 \bmod 2^{128}$).

Three different reading functions with the same structure are defined depending on the type of input element:

- `read_truncated_hash_from_prover` for a 160-bit (truncated) hash value;
- `read_felt_from_prover` for a field element;

  NB: the field element is read in standard form but put in Montgomery form before being hashed to update de channel;
- `read_uint64_from_prover` for 64-bit integer;

  NB: the function further checks that the input value is in $[0, 2^{64})$.

Additionally, two functions are defined to read several field elements:

- `read_felts_from_prover` (with `read_felts_from_prover_inner`) calls successively `read_felt_from_prover` for each read elements,
- `read_felt_vector_from_prover` (with `read_felt_vector_from_prover_inner`) updates the channel by hashing all the field elements at once.

Observation 1: Inconsistent hash input sizes

In `read_uint64_from_prover`, the number of hashed bytes (`0x28`) matches with the size of the hashed content (*i.e.* 32-byte digest plus 8-byte integer). This is not the case for `read_truncated_hash_from_prover` which uses `64` as input hash size while the hashed content makes 52 bytes (32-byte digest plus 20-byte truncated digest).

Recommendation: We recommend homogenizing the implementation, for instance by setting the number of hashed bytes as `0x34` in the second case. This modification also impacts the prover implementation.

Observation 2: Implicit assumption

At line 102, the assertion

```
assert_nn(high);
```

aims to check that `high` belongs in the range $\{0, \ldots, 2^{128} - 1\}$. However, it implicitly assumes that the upper bound of the range-check builtin is 2^{128} (which is not strictly necessary for other calls to `assert_nn`). Even if this upper bound is currently defined to 2^{128} (since the constant `RC_N_PARTS` is defined to `8`), the soundness of using the above assertion only holds of it does not change. This looks risky and could provoke feature soundness breaches.

Recommendation: We recommend changing the way this check is performed in

order to rely on a weaker assumption. For example, one could use

$$\texttt{assert\_le(high, } 2^{128} - 1\texttt{);}$$

which will work as long as the upper bound of the range-check is *at least* 2^{128}. Let us remark that this check only ensures that `high` $< 2^{128}$, but the non-negativity is already ensured by `unsigned_div_rem`.

3.1.2 File `queries.cairo`

Using the same notations as in our audit report on the Solidity implementation of the STARK verifier [1], a FRI query is sampled under the form of a random index $\mathsf{index}_0(v)$ to which corresponds the field element

$$v := c_{\text{LDE}} \cdot g_{\text{LDE}}^e \in L_0$$

where the exponent e is defined such that

$$\mathsf{index}_0(v) := \mathsf{bit\text{-}reverse}_{\log |L_0|}(e) \ .$$

All these notions are further recalled in Section 3.3 (FRI protocol) below.

Query sampling. The function `generate_queries` first calls the auxiliary function `random_uint256_to_prover` to generate random samples (given a size and an upper bound) which are then sorted (with the removal of duplicates) through a call to `usort`. The queries are sampled in quadruplets by `sample_random_queries`. For each quadruplet, 256 bits are sampled using `random_uint256_to_prover`, which are then split into four 64-bit chunks. Each query is defined as the remainder of a chunk modulo the upper bound.

- **Observation 3: Missing sanity check**

The function `sample_random_queries` would produce biased samples if it was called with an upper bound argument (`query_upper_bound`) which is not a power of two.

Recommendation: Although the above case should not occur for the FRI protocol, we still recommend adding a sanity check to verify that the given upper bound is a power of two.

Query to points. The function `queries_to_points` take as input a list of *query indexes* and returns the list of corresponding field elements by computing

$$v := c_{\text{LDE}} \cdot g_{\text{LDE}}^e$$

which corresponds to $\mathsf{index}_0(v)$. The value c_{LDE} is defined as the field generator which is given by the constant `FIELD_GENERATOR` (defined to 3 in `utils.cairo`).

3.1.3 File `utils.cairo`

Field generator. The file defines the constant `FIELD_GENERATOR` to the value 3, which is a generator of the multiplicative group $\mathbb{F}_p^\times$. This constant is used to derive the generators of the trace evaluation domain and the evaluation domain of the trace LDE (see Section 2.1) and to define the c_{LDE} value used in the FRI protocol (see Section 3.3).

Bit reversing. Let us assume that we want to swap the n-bit chunks in an unsigned integer a. Namely given

$$a := \sum_{k\geq 0}(a_{2k} \cdot (2^n)^{2k} + a_{2k+1} \cdot (2^n)^{2k+1})$$

where $0 \leq a_i < 2^n$ for all i, we want to compute

$$\sum_{k\geq 0}(a_{2k+1} \cdot (2^n)^{2k} + a_{2k} \cdot (2^n)^{2k+1}).$$

To proceed, let us define $\texttt{mask} := \sum_{k\geq 0}(\texttt{set-bits} \cdot (2^n)^{2k})$ with $\texttt{set-bits} := 2^n - 1$, we get that

$$\begin{aligned} \texttt{masked} &:= a \ \&_{\text{bitwise}} \ \texttt{mask} \\ &= \sum_{k\geq 0}(a_{2k} \cdot (2^n)^{2k}) \\ a - \texttt{masked} &= \sum_{k\geq 0}(a_{2k+1} \cdot (2^n)^{2k+1}) \end{aligned}$$

When computing $2^{2n} \cdot \texttt{masked} + (a - \texttt{masked})$, we get

$$\sum_{k=0}(a_{2k} \cdot (2^n)^{2k+2} + a_{2k+1} \cdot (2^n)^{2k+1})$$

which corresponds to the desired value scaled by a factor 2^n.

The function `bit_reverse_u64` uses the above swap process to reverse the bits of a 64-bit value: it swaps the 1-bit chunks, then the 2-bit chunks, then 4-bit chunks, etc. In the end, the result is divided by 2^{63} to remove the factor introduced by the process.

3.2 Commitments

All the commitments in the STARK protocol are realized thanks to Merkle trees, a.k.a. hash trees. In such a binary tree every node is labeled with the hash digest of the two child nodes' labels. This structure enables to commit several inputs as leaves of the tree into a single hash commitment (the root of the tree). Then, it is possible to show the consistency of a revealed subset of inputs with the 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.

We start by giving an overview of the format of the different commitments involved in the STARK protocol. We stress that the source files `vector_commitment.cairo` and `table_commitment.cairo` (reviewed hereafter) are independent of these particular formats and could be used for other types of Merkle commitments.

Trace commitments. In the STARK protocol, Merkle trees are used to commit two types of traces: the *execution trace* $f_0, \ldots, f_{W-1}$ and the *composition polynomial trace* $h_0, \ldots, h_{M_2-1}$ which are both defined on the *low degree extension* L_0 of size $\beta \cdot L$ (see Section 2.1).

The leaves of the corresponding Merkle trees have the following formats.

- For the execution trace:

$$\mathsf{input}_j = f_0(x) \parallel f_1(x) \parallel \ldots \parallel f_{W-1}(x) \quad \text{where} \quad x \in L_0,\ \mathsf{index}_0(x) = j$$

- For the composition polynomial trace:

$$\mathsf{input}_j = h_0(x) \parallel h_1(x) \parallel \ldots \parallel h_{M_2-1}(x) \quad \text{where} \quad x \in L_0,\ \mathsf{index}_0(x) = j$$

Remark 2. *Whenever the AIR to be verified is a randomized AIR with preprocessing [3] (which is always the case for a Cairo program), 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 interaction step, and the second tree contains the remaining trace columns.*

Layer polynomial commitments. In the FRI protocol, a polynomial p_i is committed at the layer $i+1$ (except for the last layer). We denote L_i the evaluation domain for the commitment of p_i. To compute an evaluation of p_{i+1} for a point of L_{i+1}, one needs $2^{\ell_{i+1}}$ evaluations of p_i (see Section 3.3 for details). 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:

$$\mathsf{input}_j = 2^{s_i} \cdot p_i(x_0) \parallel 2^{s_i} \cdot p_i(x_1) \parallel \ldots \parallel 2^{s_i} \cdot p_i(x_{2^{\ell_{i+1}}-1})$$

where

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

Merkle tree structure. 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. Each node is an instance of a `VectorQuery` structure, which is composed of two members:

- `index` (field element): the index of the node. The nodes of depth i have indexes 2^i, ..., $2^{i+1}-1$. Thus the root has index "1" and the leave indexes are $\{2^n, \ldots, 2^{n+1}-1\}$. Moreover, given an internal node with index i, the child nodes have indexes $2i$ and $2i+1$.

- `value` (unsigned 256-bit integer): the label of the node in the Merkle tree. This is the hash digest of the child nodes for an internal node. For a leaf node, this is either the corresponding input value whenever it holds on 256 bits, or the hash digest of the input value otherwise.

The hash function used for the Merkle tree is a truncated version of Blake2s for which we keep only the 160 most significant bits of the digest.

3.2.1 File `vector_commitment.cairo`

The file deals with commitments and decommitments of vectors. A vector $(x_1, \ldots, x_{2^n})$ of length 2^n is committed as a Merkle tree with x_1, $\ldots$, x_{2^n} as leaf values. The coordinates of committed vectors are unsigned 256-bit integers.

Commitment. For the IOP verifier, the vector commitment simply consists in receiving a 160-bit Merkle root from the prover. The `vector_commit` function thus only calls `read_truncated_hash_from_prover`.

Decommitment. A vector decommitment consists in checking that some opened vector coordinates are consistent with a Merkle root using the underlying authentication paths. For this purpose, the `vector_commitment_decommit` function proceeds as follows:

- by calling `shift_queries`, it computes the indexes $\overline{\mathsf{index}}(v)$ of the opened Merkle leaves. This is done by shifting the indexes $\mathsf{index}(v)$ of the opened vector coordinates:

$$\overline{\mathsf{index}}(v) := \mathsf{index}(v) + 2^{\texttt{height}}$$

  where `height` is the height of the Merkle tree.

- by calling `verify_authentications`, it computes the expected Merkle root using the opened vector coordinates and the authentication paths.
  - To compute the expected Merkle root, the implementation uses a queue. At the beginning of the verification, the queue only contains the decommitted leaves (sorted according to their indexes). Along the verification, the queue is fed with the encountered nodes on the Merkle paths to the decommitted leaves.
  - A computation step consists to pop the Merkle node at the queue head, to compute the index and the label of the parent node (using the following node in the queue when it corresponds to the sibling node, using a value from the authentication paths otherwise) and to push the parent node in the queue. In case any inconsistency is encountered, the verification fails.
  - At the end, the queue contains a single node which shall be the Merkle root.

- it checks that the obtained root (`expected_commitment`) is well truncated to 160 bits and further checks that it matches the committed value read from the proof.

Observation 4: Inconsistent naming

The decommitment function for vectors is named `vector_commitment_decommit`, while the decommitment functions for tables and traces are named `table_decommit` and `traces_decommit`.

Recommendation: We recommend using consistent naming, for instance, change

`vector_commitment_decommit`

into `vector_decommit`.

Observation 5: Useless assertion

The assertion

```
assert_nn(expected_commitment.low / 2 ** 96);
```

seems to check that `expected_commitment` has only 160 bits (on the leading positions). However, the value `expected_commitment` is a value returned by `truncated_blake2s`, and thus always satisfies this assertion. The only exception is when the committed vector has a single coordinate, which should not occur.

Recommendation: If the desired behavior is to avoid the case of a committed vector with a single coordinate, then we recommend implementing a more explicit assertion. Otherwise, we recommend removing the assertion.

Observation 6: Confusing typing

The structure `VectorQuery` is used for both the input values of a Merkle tree as well as the internal nodes of a Merkle tree. For an input value, the member `index` is the index of a field element, $\mathsf{index}_i(v)$, and the member `value` is the hash digest of polynomial evaluations in v (and in further elements following v for a FRI layer polynomial commitment). For a Merkle node, the member `index` is the index of the node in the tree, and the member `value` is the label (hash digest) of the node. The role of the function `shift_queries` is to translate the former notion to the latter one since there is a mismatch between the two notions of indexes. Using the same structure for the two notions is confusing.

Recommendation: We recommend defining two different structures, one for Merkle inputs (before index shifting) and one for Merkle nodes (after index shifting). We would also recommend avoiding the terminology of "query" but prefer the terminologies of "input" or "leaf" and "node".

3.2.2 File `table_commitment.cairo`

The purpose of this file is to extend vector commitments into *table commitments*. Instead of (de)commitments of unsigned integers (the coordinates of the vector), this file deals with (de)commitments of rows of field elements. All the values of a row are committed together, meaning that the prover has to decommit entire rows. The idea is to hash each row into a digest (except when the rows have a unique element) and to use the vector commitment to commit the resulting vector of hash digests.

Commitment. For the IOP verifier, a table commitment is equivalent to a vector commitment. The function `table_commit` simply receives a 160-bit hash digest from the prover by calling `vector_commit`.

Decommitment. The function `table_decommit` proceeds as follows:

- it converts each input decommitted value to *Montgomery form* by calling `to_montgomery`,
- it computes the hash digest for each opened row by calling `generate_vector_queries` (except if the rows have a single element),
- it checks that the decommitment is valid by calling `vector_commitment_decommit`.

> **Observation 7: Code duplication**
>
> In `generate_vector_queries`, the code blocks at lines 135-141 and lines 153-159 are the same.
>
> **Recommendation:** For the sake of code readability, we recommend using an `else` statement and having a single `return` statement.

3.2.3 File `traces.cairo`

This file deals with the commitment and decommitment of the execution traces.

Commitment. The function `traces_commit` is used to read a trace commitment from the channel. First, it calls `table_commit` to read from the proof the table commitment of the "original" trace columns (namely the trace columns before interaction). Then it calls `random_felts_to_prover` to generate the interaction elements. Afterwards, it calls `table_commit` once again to read the table commitment of the "interaction" trace columns (namely the trace columns after interaction). Finally, it returns the resulting table commitments (of the original and interaction columns) packed in a `TracesCommitment` structure (together with the public input and the sampled interaction elements).

Decommitment. The function `traces_decommit` is used to verify a decommitment for the traces at some given query indices. It simply calls the `table_decommit` function on the table commitment of the original columns and the table commitment of the interaction columns (inputting the corresponding decommitments and witnesses).

> ● **Observation 8: File organization**
>
> The only purpose of the file `stark_verifier/air/config.cairo` is to define the trace configuration structure and the function to validate a trace configuration.
>
> **Recommendation:** We recommend merging `stark_verifier/air/config.cairo` into the file `stark_verifier/air/traces.cairo`.

3.3 FRI protocol

We recall hereafter some basics about the FRI protocol and the underlying notions and data structures. The FRI protocol has two distinct phases: a *commitment* phase and a *query* (or *decommitment*) phase. We outline those two phases hereafter:

- **Commitment phase**. For all the layers except the last one,
  - At the ith layer, the IOP prover starts with a polynomial p_i. It first commits this polynomial in a table commitment (see the previous section). The evaluation domain for the commitment of p_i is

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

which by definition of L_0 implies

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

with $s_i := \sum_{j=1}^{i} \ell_j$. For each value $x := c_{\text{LDE}}^{2^{s_i}} \cdot (g_{\text{LDE}}^{2^{s_i}})^e \in L_i$, we denote

$$\mathsf{index}_i(x) := \mathsf{bit\text{-}reverse}_{\log |L_i|}(e).$$

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:

$$\mathsf{input}_j = 2^{s_i} \cdot p_i(x_{j,0}) \,||\, 2^{s_i} \cdot p_i(x_{j,1}) \,||\, \ldots \,||\, 2^{s_i} \cdot p_i(x_{j,2^{\ell_{i+1}}-1})$$

where

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

Remark 3. *One can observe that the leaves' polynomial evaluations are scaled by a factor 2^{s_i}. This factor is not in the original definition of the FRI protocol but is introduced for the sake of efficiency of the verifier implementation. Indeed, the verifier evaluates so-called folding formulae to map queries from one layer to queries of the next layer. This evaluation introduces the 2^{s_i} factor. To avoid correcting it at the verifier level, this factor is then further applied to the committed values.*

– Once the layer polynomial p_i committed, the IOP prover gets a random challenge ζ_i from the verifier and computes the new polynomial p_{i+1} as

$$p_{i+1}(x) := \sum_{j=0}^{2^{\ell_{i+1}}-1} \zeta_i^j \cdot p_i^{[j]}(x) \tag{4}$$

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

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

To compute an evaluation of p_{i+1} for a point of L_{i+1}, one needs $2^{\ell_{i+1}}$ evaluations of p_i, it is why these evaluations are committed together.

At the last layer, the IOP prover simply sends $2^{s_{K-1}} \cdot p_{K-1}$ to the verifier.

- **Query/decommitment phase**. Once all the layer polynomials $\{p_i\}_i$ have been 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.

At the ith layer, the Cairo implementation of the verifier

– takes a set $\mathcal{Q}_i$ of queries where each query $v = c_{\text{LDE}}^{2^{s_i}} \cdot g_i^e \in \mathcal{Q}_i$ with scaled value $v' = g_i^e$ is represented as a triplet

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

stored via the structure `FriLayerQuery`, where

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

– builds the set $\mathcal{Q}_{i+1}$ of queries represented with the same format (but for $i+1$),

– checks that all the evaluations of p_i used to build $\mathcal{Q}_{i+1}$ are consistent with the commitment.

Scaled values. As explained in our previous audit report [1], the verifier implementation is based on *scaled* versions of several FRI notions (yet another scaling different from the 2^{s_i} factor addressed above). 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 as Equation (4) 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

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

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. This choice enables to avoid all the offsets $\left\{c_{\text{LDE}}^{2^{s_i}}\right\}_i$ 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 deal with.

This change of representation does not impact the commitments, 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 equivalent) and that the IOP prover sends the scaled version of the last layer polynomial p'_{K-1} (instead of p_{K-1}).

The FRI group. The FRI group is the subgroup of $\mathbb{F}_p^\times$ with 2^4 elements, sorted in a specific order. Let ξ be a generator of this group. Then the ith element of the group, denoted `friGroup[`i`]`, is defined as

$$\texttt{friGroup[}i\texttt{]} := \xi^e \quad \iff \quad i = \mathsf{bit\text{-}reverse}_4(e) \ .$$

We refer the reader to our previous audit report [1] for more details about the FRI group.

3.3.1 File `fri/fri_formula.cairo`

In what follows, we denote $\mathsf{rev}(\cdot)$ for $\mathsf{bit\text{-}reverse}_{|L_i|}(\cdot)$ to ease the notations.

This file implements the computation of $2^{s_{i+1}} \cdot p_{i+1}(v)$ for some $v \in \mathcal{Q}_{i+1}$ using evaluations of $2^{s_i} \cdot p_i$ where p_i and p_{i+1} satisfy Equation (4). Let us denote

$$\mathsf{Combine}_j(\{e_0, \ldots, e_{2^j-1}\}, \zeta, v)$$

the function which returns $2^{s_{i+1}} \cdot p_{i+1}(v^{2^{\ell_{i+1}}})$ whenever $\zeta = \zeta_{i-1}$ and $e_0, \ldots, e_{2^j-1}$ are the evaluations

$$2^{s_i} \cdot p_i(v \cdot \xi^{\mathsf{rev}(0)}), \ 2^{s_i} \cdot p_i(v \cdot \xi^{\mathsf{rev}(1)}), \ \ldots, \ 2^{s_i} \cdot p_i(v \cdot \xi^{\mathsf{rev}(2^j-1)})$$

for ξ a generator of the subgroup of $\mathbb{F}_p$ of size $2^{\ell_{i+1}}$. By [1, Appendix A], we know that

$$p_{i+1}(v^{2^{\ell_{i+1}}}) = \frac{1}{2^{\ell_{i+1}}} \cdot \sum_{q=0}^{2^{\ell_{i+1}}-1} \left(\frac{\zeta_i}{v}\right)^q \cdot \sum_{k=0}^{2^{\ell_{i+1}}-1} (\xi^{-\mathsf{rev}(k)})^q \cdot p_i(v \cdot \xi^{\mathsf{rev}(k)}),$$

thus we get

$$\mathsf{Combine}_j(\{e_0,\ldots,e_{2^j-1}\},\zeta,v) := \sum_{q=0}^{2^j-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^j-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_k \ .$$

When $j=1$, we get

$$\mathsf{Combine}_1(\{e_0,e_1\},\zeta,v) := (e_0+e_1)+\zeta\cdot v^{-1}\cdot(e_0-e_1)$$

since $\xi^{-\mathsf{rev}(0)}=1$ and $\xi^{-\mathsf{rev}(1)}=-1$. Moreover, Combine satisfies the recursive relation

$$\mathsf{Combine}_j(\{e_0,\ldots,e_{2^j-1}\},\zeta,v) = \mathsf{Combine}_1(\{e'_0,e'_1\},\zeta^{2^{j-1}},v^{2^{j-1}})$$

where

$$\begin{aligned} e'_0 &:= \mathsf{Combine}_{j-1}(\{e_0,\ldots,e_{2^{j-1}-1}\},\zeta,v)\\ e'_1 &:= \mathsf{Combine}_{j-1}(\{e_{2^{j-1}},\ldots,e_{2^j-1}\},\zeta,v\cdot\xi)\end{aligned}$$

We provide a proof that this recursive relation is correct in Appendix A.

The function `fri_formula2` correctly implements $\mathsf{Combine}_1$. Moreover, the functions `fri_formula4`, `fri_formula8` and `fri_formula16` correctly implement $\mathsf{Combine}_2$, $\mathsf{Combine}_3$ and $\mathsf{Combine}_4$ using the recursive relation. Finally, the constants `OMEGA_16`, `OMEGA_8`, `OMEGA_4` and `OMEGA_2` are well defined (respectively defined as a generator of the subgroup of $\mathbb{F}_p^\times$ of size 16, of size 8, of size 4 and of size 2).

3.3.2 File `fri/fri_layer.cairo`

FRI group. The function `get_fri_group` returns an address of a memory segment where the FRI group is stored.

Computing next layer. The function `compute_next_layer` takes as input

- a list `queries` of queries $\mathcal{Q}_i$ (represented as a list of `FriLayerQuery`), and
- a list `sibling_witness` of evaluations of the polynomial p_i.

The function computes the queries of the next layer as follows:

- it guesses the coset index `cosetIdx` (with a hint), which corresponds to the common prefix of the indexes of $v\cdot\langle\xi\rangle$, where ξ is the generator of the subgroup of $\mathbb{F}_p^\times$ of size 2^{ℓ_i}. Let us denote c the element of $v\cdot\langle\xi\rangle$ with the lowest index.
- by calling `compute_coset_elements`, it gathers all the evaluations

  $$2^{s_i}\cdot p'_i(c),\ 2^{s_i}\cdot p'_i(c\cdot\xi^{j_1}),\ 2^{s_i}\cdot p'_i(c\cdot\xi^{j_2}),\ldots$$

  where the exponents $j_1,j_2,\ldots,$ matches the set $\{0,\ldots,2^{\ell_i}-1\}$ in increasing bit-reverse order. Specifically:

  - if the query list is not empty and the following query has index $\texttt{cosetIdx} \cdot 2^{\ell_i} + j$, it gets the evaluation of p_i from the query list (and computes $\frac{1}{c}$ as $v \cdot \texttt{friGroup[}j\texttt{]}^{-1}$);
  - otherwise, it takes the evaluation from `sibling_witness` (this value will be checked later with the Merkle tree).

  The evaluations of p'_i are stored in `verify_y_values` and the coset index is stored in `verify_indices`.

- it checks that at least one query has been read from the query list. This notably implies that the index of v is in the range $\{\texttt{cosetIdx} \cdot 2^{\ell_i}, \ldots, (\texttt{cosetIdx}+1) \cdot 2^{\ell_i} - 1\}$ hence the guess on the coset index was necessarily correct.
- it computes $2^{s_{i+1}} \cdot p_{i+1}(v^{2^{\ell_i}})$ using the folding formulae from `fri/fri_formula.cairo`.
- it adds the triplet
$$\left(\texttt{costIdx},\ 2^{s_{i+1}} \cdot p_{i+1}(v^{2^{\ell_i}}),\ \frac{1}{(v')^{2^{\ell_i}}}\right)$$
to the FRI queries as an element to be processed by the next FRI layer.

At the end, the function `compute_next_layer` outputs

- `next_queries`: the list of the queries to be processed by the next FRI layer,
- `verify_indices`: the list of indexes of layer polynomial evaluations to be decommitted,
- `verify_y_values`: the list of layer polynomial evaluations to be decommitted.

Observation 9: Confusing naming

The variable `sibling_witness` contains the additional evaluations of the layer polynomials required to perform the decommit phase of the FRI protocol. However, the term "sibling witness" usually refers to the authentication paths of Merkle trees.

Recommendation: We recommend renaming the variable and avoiding the "sibling" terminology. For example, it could be renamed as `coset_witness`.

3.3.3 File `fri/fri.cairo`

FRI commitment. The function `fri_commit` corresponds to the commitment phase of the FRI protocol. It calls the recursive function `fri_commit_rounds`: for each layer i except the last one, this function reads the commitment of the layer polynomial p_i from the proof and samples the challenge ζ_i. Then the function `fri_commit` reads the coefficients of the polynomial p'_{K-1} from the proof. It returns all the commitment digests, the challenges $\{\zeta_i\}_i$ and the polynomial p'_{K-1} packed in a `FriCommitment` structure.

Observation 10: Confusing naming

The values in `eval_points` corresponds the challenges $\{\zeta_i\}_i$ which are not evaluation points.

Recommendation: We recommend changing the name of this variable.

FRI decommitment. The function `fri_decommit` proceeds as follow:

- the function `gather_first_layer_queries` formats the initial FRI queries $\mathcal{Q}_0$ using the evaluations of the DEEP composition polynomial p_0. For each $v \in \mathcal{Q}_0$ with scaled value $v' := \frac{v}{c_{\text{LDE}}}$, it builds the triplet

$$\left(\mathsf{index}_0(v),\ p_0(v),\ \frac{1}{v'}\right)$$

  represented by the `FriLayerQuery` structure.

- then, the function `fri_decommit_layers` runs the decommitment of the FRI layers: for each layer i,
  - it computes the evaluations of p_{i+1} on the queries $\mathcal{Q}_{i+1} := \mathcal{Q}_i^{2^{\ell_i}}$ using evaluations of p_i, thanks to the function `compute_next_layer` of `fri_layer.cairo`.
  - it checks that the evaluations of p_i used by the `compute_next_layer` function (which are returned in `verify_y_values` with corresponding coset indices in `verify_indices`) are consistent with the commitment of p_i, thanks to the function `table_decommit`.

- finally, the function `verify_last_layer` evaluates the last layer polynomial p_{K-1} on all the queries $\mathcal{Q}_{K-1}$ and checks that the evaluations are consistent with the output of the FRI protocol.

Observation 11: Unused variable

The member `n_leaves` of the structure `FriLayerWitness` is never used.

Recommendation: We recommend verifying that the number of read values in `sibling_witness` is exactly `n_leaves`.

3.3.4 File `fri/config.cairo`

FRI configuration. The function `fri_config_validate` performs a batch of checks on the FRI configuration. It performs some sanity checks:

$$0 \leq \deg p_{K-1} \leq 15,$$
$$2 \leq K \leq 15.$$

It also checks that $\ell_0 = 0$, and by calling the function `fri_layers_config_validate`, it checks that

$$1 \leq \ell_i \leq 4,$$

for all $1 \leq i < K$. At the same time, it computes

$$s_{K-1} := \sum_{i=1}^{K-1} \ell_i$$

and it returns the expected degree of the polynomial p_0 (*i.e.* the first layer polynomial)

$$\log_2 \deg p_0 := \log_2 p_{K-1} + s_{K-1}$$

after checking that $\log_2 \deg p_0 + \log_2 \beta = \log_2 |L_0|$.

Observation 12: Confusing naming

The variable `log_n_cosets` corresponds to the logarithm of the blowup factor β.

Recommendation: We recommend using the notion of *blowup factor* instead of speaking about the number of cosets.

3.4 Core STARK verifier

The source files reviewed in this section correspond to the first verification level described in Section 2.2.2. They implement the core of the STARK verification process without dealing with the format of execution traces, the AIR constraints, and the public input. The STARK core depends on several external functions. Those functions must be made accessible to the STARK core through an `AirInstance` structure (from `air_interface.cairo`). The entry point of the STARK verifier is the function

```
verify_stark_proof(air, proof, security_bits)
```

of the file `stark_verifier/core/stark.cairo`.

3.4.1 File `domains.cairo`

This file defines the structure `StarkDomains`. The latter gathers all the information about the trace domain (each trace column is interpreted as the evaluations of a polynomial over this domain) and the evaluation domain L_0. The members of this structure are

- `eval_domain_size` which corresponds to $|L_0| := \beta \cdot L$,
- `log_eval_domain_size` which corresponds to $\log_2 |L_0|$,
- `eval_generator` which corresponds to $g^{\frac{p-1}{|L_0|}}$,

- `trace_domain_size` which corresponds to $L := \texttt{CPU\_COMPONENT\_HEIGHT} \cdot N$,
- `log_trace_domain_size` which corresponds to $\log_2 L$,
- `trace_generator` which corresponds to $g^{\frac{p-1}{L}}$,

where g is a generator of $\mathbb{F}_p^\times$.

This structure is initialized by the function `stark_domains_create` (from the file `stark_verifier/core/config.cairo`) which is called by `verify_stark_proof`. This initialization ensures that the values in the structure are self-consistent, while a subsequent call to `public_input_validate` (from `public_verify.cairo`) makes sure that the structure is consistent with the public input.

3.4.2 File `air_interface.cairo`

This file defines the structure `AirInstance` which gathers all the functions necessary to the STARK core for running the proof verification process. Those functions are

- `public_input_hash`: which takes as input a pointer to the public input and must return the hash digest of the public input.
- `public_input_validate`: which performs a batch of checks to validate the configuration of the public input. At the same time, it must check that the given STARK domains are consistent with the public input.
- `traces_config_validate`: which performs a batch of checks to validate the format of the traces.
- `traces_commit`: which updates the verifier channel by reading trace commitments from the proof.
- `traces_decommit`: which checks that the revealed values of the execution trace are consistent with the commitment.
- `traces_eval_composition_polynomial`: which evaluates the composition polynomial h in the OODS point z using the mask $\{y_\ell\}_\ell$.
- `eval_oods_boundary_poly_at_points`: which evaluates the DEEP composition polynomial p_0 for the queries in $\mathcal{Q}_0$.

Moreover, the `AirInstance` structure includes some constants related to the AIR constraints:

- `n_constraints` is the number of constraints, which we denote k,
- `constraint_degree` is the constraint degree, which we denote M_2,
- `mask_size` is the size of the mask, which we denote M_1.

The `air_interface.cairo` file further provides wrapper functions for the routines from `AirInstance`. For example, the function `public_input_hash` takes an `AirInstance` structure as input and performs an absolute jump to the same function from the structure (thus forwarding its input arguments).

The `air_interface.cairo` file also defines void structures `PublicInput` and `Traces*` since the core STARK verifier manipulates some instances of these structures (without accessing their members). It further defines the structure `OodsEvaluationInfo` used by the function `eval_oods_boundary_poly_at_points`.

Let us remark that the definition of the traces is not part of the core STARK verifier. In practice, there are two possible trace formats depending on whether the STARK proof relies on classical AIR constraints, or on Randomized AIR with Preprocessing [3]. A different choice is made in the Solidity implementation of the verifier for which the two formats are supported in the core STARK verifier (which takes a parameter flag to select the format). Another option would be to only support the second format which is necessary for Cairo programs (because of the way the memory is handled). The current choice offers more flexibility for possible future evolutions of the format of traces.

3.4.3 File `proof_of_work.cairo`

A proof of work is used to impose a computational effort on the IOP prover after the STARK commitment phase and before receiving the FRI queries. The proof of work consists in solving a partial hash preimage puzzle which depends on the channel state. The IOP verifier just receives a nonce (*i.e.* the solution of the puzzle) and checks its validity.

Proof of work. The proof-of-work puzzle consists in finding a 64-bit nonce such that the hash digest resulting of the hash of the nonce which, when hashed together with the channel state, results in a digest with `n_bits` most significant bits to zero. Figure 5 gives a graphical description of the puzzle.

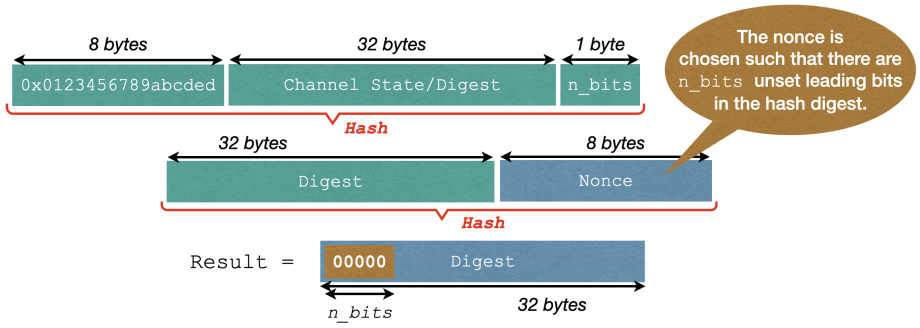

Figure 5: Proof of work

The function `proof_of_work_commit` gets the nonce from the proof transcript and checks it with `verify_proof_of_work`. Given the nonce and the channel state, the latter function verifies that the nonce constitutes a valid proof of work by computing the hash digest as depicted in Figure 5 and by checking that its `n_bits` leading bits are unset. Let us remark that the channel state on which the proof of work depends is the state *before reading the nonce*.

> **Observation 13: Terminology**
> The name `proof_of_work_commit` does not seem fully appropriate. Indeed this function does not properly "commit" a proof of work but it verifies by calling the function `verify_proof_of_work`.
> **Recommendation:** Avoid the "commit" terminology for this function.

Proof-of-work configuration. The function `proof_of_work_config_validate` performs the following sanity check on the proof-of-work configuration:

$$1 \leq \kappa_{pow} \leq 50$$

where κ_{pow} is the computational cost (in bits) of the proof of work.

3.4.4 File `config.cairo` (`stark_verifier/core`)

In what follows, we denote κ_{pow} the computational cost (in bits) of the proof of work and κ the security (in bits) of the STARK proof.

STARK configuration. The function `stark_config_validate` performs several sanity checks on the proof configuration:

$$\begin{aligned} 1 \leq \log_2 N &\leq 64 \\ 0 \leq \kappa & \\ 1 \leq \log_2 \beta &\leq 16 \\ \kappa_{pow} &\leq \kappa \\ 1 \leq \#\mathcal{Q}_0 &\leq 48. \end{aligned}$$

It also checks that the configuration achieves the desired security level, that is:

$$\#\mathcal{Q}_0 \cdot \log_2 \beta + \kappa_{pow} \geq \kappa.$$

It then calls the validation functions of different proof components:

- `proof_of_work_config_validate` validates the configuration of the proof of work by checking
$$1 \leq \kappa_{pow} \leq 50.$$

- `traces_config_validate` validates the configuration of the traces by checking

$$1 \leq W' \leq 128,$$
$$1 \leq W - W' \leq 128.$$

- `fri_config_validate` validates the configuration of the FRI protocol by checking

$$0 \leq \deg p_{K-1} \leq 15,$$
$$2 \leq K \leq 15,$$
$$\ell_0 = 0,$$
$$\forall 1 \leq i < K, 1 \leq \ell_i \leq 4.$$

Initializing domains. The function `stark_domains_create` initializes a `StarkDomains` object from the STARK configuration:

- it computes the size of the trace domain and computes a generator of the trace domain,
- it computes the size of the evaluation domain $|L_0| = \beta L$ and computes the generator g_{LDE}, such that $L_0 = c_{\text{LDE}} \cdot \langle g_{\text{LDE}} \rangle$.

The two generators are computed by raising the "field generator" g to the appropriate power.

Observation 14: Confusing expression

At line 85, the generator for the evaluation domain $g_{\text{LDE}} := g^{\frac{p-1}{|L_0|}}$ is computed as

```
pow(FIELD_GENERATOR, (-1) / eval_domain_size),
```

which might be confused with $g^{\frac{-1}{|L_0|}}$. Same remark for the generator of the trace domain at line 87.

Recommendation: We recommend defining a constant `PRIME` for p and to use it in those formulae, or at least to add some comments next to those lines to explain the computation.

3.4.5 File `stark.cairo`

Structure definition. The file `stark.cairo` defines several data structures. The structure `StarkProof` gathers the different elements of the proofs (as detailed in Section 2.3.1):

- the proof configuration (`StarkConfig`),
- the public input (`PublicInput`),

- the commitments of the proof transcript (`StarkUnsentCommitment`), and
- the witnesses revealed in the STARK decommitment phase (`StarkWitness`).

The structure `StarkCommitment` represents all the data from the STARK commitment phase which will be used in the STARK decommitment phase. Finally, additional structures (whose names start by `InteractionValuesAfter`) are defined to wrap some challenges sampled by the verifier.

STARK commitment. The function `stark_commit` processes the successive steps of the STARK commitment phase as described in Figure 1 (Section 2.2):

- it reads the trace commitment from the proof using `traces_commit`,
- it samples the constraint coefficients $\{\alpha_j\}_j, \{\beta_j\}_j$ using `random_felts_to_prover`,
- it reads the commitment of the composition polynomial h using `table_commit`,
- it samples the OODS point z using `random_felts_to_prover`,
- it reads the OODS values $\{y_\ell\}_\ell \cup \{\hat{y}_\ell\}_\ell$ using `read_felts_from_prover`,
- by calling `verify_oods`,
  - it evaluates the composition polynomial h on the OODS point z with the mask $\{y_\ell\}_\ell$ using `traces_eval_composition_polynomial`,
  - it evaluates again the composition polynomial h on the OODS point z as
$$h(z) = \hat{y}_0 + z \cdot \hat{y}_1,$$
  - it checks that the two above computations give the same result.
- it samples the OODS coefficients $\{\gamma_i\}_i$ using `random_felts_to_prover`,
- it runs the FRI commitment phase by calling `fri_commit`,
- it verifies that the prover correctly performed the proof of work,
- it finally returns all the commitments (trace, composition, FRI), the OODS point, the OODS coefficients, and the OODS values packed in a structure `StarkCommitment` (let us remark that the constraint coefficients are not useful anymore).

STARK decommitment. The function `stark_decommit` processes the successive steps of the STARK decommitment phase as described in Figure 1 (Section 2.2):

- it checks that `witness.traces_decommitment` are valid opened values
$$\{f_0(v), \ldots, f_{W-1}(v)\}_{v \in \mathcal{Q}_0}$$
using `traces_decommit`,

- it checks that `witness.composition_decommitment` are valid opened values

$$\{h_0(v), \ldots, h_{M_2-1}(v)\}_{v \in \mathcal{Q}_0}$$

  using `table_decommit`,
- it computes the field elements in $\mathcal{Q}_0$ from their indexes using `queries_to_points`,
- it evaluates the DEEP composition polynomial p_0 on the queries:

$$\{p_0(v), v \in \mathcal{Q}_0\}$$

  using `eval_oods_boundary_poly_at_points`,
- and finally, it runs the FRI decommitment phase by calling `fri_decommit`.

STARK verifier. The function `verify_stark_proof` is the entry point of the core STARK verifier. It takes as input an `AirInstance` instance, a `StarkProof` instance, and the security level (in bits). It proceeds as follows:

- it validates the proof configuration by calling `stark_config_validate` and computes the STARK domains by calling `stark_domains_create`,
- it validates the public input (and checks its consistency with the STARK domains) by calling `public_input_validate`,
- it allocates a memory segment dedicated for Blake2s instances: this memory segment will be given as an implicit parameter to functions (in the same way as a builtin memory segment),
- it hashes the public input using `public_input_hash`,
- it initializes the verifier channel with the hash digest of the public input by calling `channel_new`,
- it runs the STARK commitment phase by calling `stark_commit`,
- it samples the queries $\mathcal{Q}_0$ by calling `generate_queries`,
- it runs the STARK decommitment phase by calling `stark_decommit`,
- it finally checks that all the instances in the Blake2s memory segment are valid.

Observation 15: Confusing naming

The naming used in the implementation is inconsistent with the notions from the STARK/Cairo documentation.

- The constraint coefficients $\{\alpha_j\}_j$ and $\{\beta_j\}_j$ are named (or referred to as):
  - `interaction values after traces` in `stark.cairo`
  - `traces coefficients` in `stark.cairo`

  - `constraint_coefficients` in `composition.cairo`

- The OODS coefficients $\gamma_0, \ldots, \gamma_{M_1+M_2-1}$ are named (or referred to as):
  - `interaction values after OODS` in `stark.cairo`
  - `oods_coefficients` in `stark.cairo`
  - `constraint_coefficients` in `oods.cairo` and in `OodsEvaluationInfo`

This confusing naming harms the code readability and increases the risk of implementation errors.

Recommendation: We recommend making the naming consistent, typically by using the terminology defined in Section 2.1 (which was gathered from different StarkWare resources [5, 6, 3]).

■ **Observation 16: Useless wrapping**

The structures

`InteractionValuesAfterTraces`, `InteractionValuesAfterComposition` and `InteractionValuesAfterOods`

are wrappers for the constraint coefficients, the OODS point and the OODS coefficients respectively. However, the purpose of those wrappers is not clear, and using them decreases the readability of the source code (by increasing the naming confusion reported above).

Recommendation: We recommend removing these structures from the source code, or at least, to document their purpose and revising their naming.

3.5 Public input

The statement to be verified by the CPU verifier (the second verification level, see Section 2.2.2) is represented by a structure named *the public input*. This structure is summarized in Table 3. Let us stress that the public input of the Cairo verifier described in Table 1 (Section 2.3) is a specific case of Table 3.

The CPU verifier aims to check that the proof has been built from a valid execution trace of a given Cairo program. Such an execution trace involves a memory mapping. Some values of this memory mapping are publicly known, *i.e.* are part of the statement to be verified. This public part of the memory is called the *public memory* which is denoted $\mathfrak{m}^*$ in the Cairo documentation. The verifier must check that the execution trace committed by the prover is consistent with $\mathfrak{m}^*$.

In the present implementation of the CPU verifier, the public memory is divided into several pages. Each page contains a part $\mathfrak{m}_i^*$ of the public memory. All together, the pages

Field	Description
`log_n_steps`	Log (in base 2) of the number N of states
`rc_min`	Min range-check value
`rc_max`	Max range-check value
`layout`	The layout identifier
`n_segments`	Number of segments
`segments[0].begin_addr`	First address of the segment 0
`segments[0].stop_ptr`	Final position of the segment-0 pointer
`segments[1].begin_addr`	First address of the segment 1
`segments[1].stop_ptr`	Final position of the segment-1 pointer
...	...
`padding_addr`	The padding address a_{pad}
`padding_value`	The padding value $\mathfrak{m}(a_{\text{pad}})$
`main_page_len`	Size of the main page
`main_page[0]`	A memory mapping (an address with its value)
`main_page[1]`	Another memory mapping
...	...
`n_continuous_pages`	Number of continuous pages
For each continuous page i:	
`start_address`	The first address of $\mathfrak{m}_i^*$
`size`	Size of $\mathfrak{m}_i^*$
`hash`	Hash of $\mathfrak{m}_i^*$
`prod`	$\prod_{a \in A_i^*}(z - (a + \alpha \cdot \mathfrak{m}_i^*(a)))$

Table 3: Structure of the public input.

form a partition of the public memory:

$$\mathfrak{m}^* = \bigcup_i \mathfrak{m}_i^*.$$

The first page (named the *main page*) is a *regular* one, meaning that this page corresponds to a list of pairs address-value (represented by the structure `AddrValue`). The other pages are *continuous* ones, *i.e.* all the addresses in the same page are adjacent. A continuous page is encoded by its starting address and a list of consecutive values.

3.5.1 File `public_memory.cairo`

The function `get_page_product` computes the cumulative product for a regular page. Given the (partial) public memory $\mathfrak{m}_0^* : A^* \to \mathbb{F}_p$ and interaction elements (z, α), it

outputs

$$\prod_{a \in A^*} (z - (a + \alpha \cdot \mathfrak{m}_0^*(a))) \ .$$

In practice, the above function is used to compute the cumulative product of the main page.

A header of a continuous page $\mathfrak{m}_i^*$ (represented by the structure `ContinuousPageHeader`) contains the first address a_0 of the page, its size $n := |A_i^*|$, its hash digest

$$h_{\mathfrak{m}_i^*} := \text{Hash}\left(\mathfrak{m}^*(a_0), \ldots, \mathfrak{m}^*(a_0 + n - 1)\right)$$

and its cumulative product

$$\mathtt{prod}_i = \prod_{a \in A_i^*} (z - (a + \alpha \cdot \mathfrak{m}^*(a))) \quad \in \mathbb{F}_p,$$

where $A_i^* := \{a_0, a_0 + 1, \ldots, a_0 + n - 1\}$ is the domain of $\mathfrak{m}_i^*$. Using such headers, the function `get_continuous_pages_product` computes the global cumulative product for all the continuous pages:

$$\prod_i \mathsf{prod}_i$$

where $\mathsf{prod}_1, \mathsf{prod}_2, \ldots$ are the individual cumulative products of the continuous pages.

Let us stress that the CPU verifier does not check that the cumulative products of the continuous pages are consistent with their hash digests. It should be verified by the code which calls this implementation of the CPU verifier. In the audited source code, this call is performed by `verify_cairo_proof` of the file

`cairo_verifier/layouts/xxxx/cairo_verifier.cairo`.

This version of the Cairo verifier considers a public memory that is only made of the main page (see explanation in Section 2.2.2). Therefore, the audited verifier does not care about the consistency between the cumulative products and the hash digest of the continuous pages.

3.5.2 File `public_input.cairo`

This file implements the functions related to the public input. It defines the structure `PublicInput` as depicted in Table 3. It further provides two functions.

Hashing the public input. The function `public_input_hash` computes the hash digest of the public input to be used as the initial state for the verifier channel. To proceed, it first computes the *Pedersen* hash digest of the main page as

$$\texttt{main\_page\_hash} := \text{Hash}(a_0, \mathfrak{m}_0^*(a_0), a_1, \mathfrak{m}_0^*(a_1), \ldots)$$

where $\{a_0, a_1, \ldots\} = A_0^*$ is the domain of $\mathfrak{m}_0^*$, and then it computes the *Blake2s* hash digest of the public input. The data is hashed in the order depicted Table 4. A recursive function `add_continuous_page_headers` is used to loop over the continuous pages.

log_n_steps
rc_min
rc_max
layout
segments[0].begin_addr
segments[0].stop_ptr
segments[1].begin_addr
segments[1].stop_ptr
⋮
padding_addr
padding_value
n_continuous_pages + 1
main_page_len
main_page_hash
pages[0].start_address
pages[0].size
pages[0].hash
pages[1].start_address
pages[1].size
pages[1].hash
⋮

Table 4: Hashing order for the public input

Let us remark that the number of segments is fixed for a specific layout, which is why this number is not hashed. Moreover, the cumulative products of the public memory pages do not belong to the statement to prove since they depend on interaction elements which are sampled during the proof generation, which is why they are not hashed either.

Cumulative product ratio. Some cells of the execution trace are dedicated to the public memory. The number of such cells is defined as

$$S := \frac{L}{\texttt{PUBLIC\_MEMORY\_STEP}}$$

where `PUBLIC_MEMORY_STEP` is a parameter of the CPU layout. Given S and two interaction elements (z, α), the function `get_public_memory_product_ratio` computes the quotient

$$\frac{z^S}{(z - (a_{\text{pad}} + \alpha \cdot \mathfrak{m}^*(a_{\text{pad}})))^{S-|A^*|} \cdot \prod_{a \in A^*}(z - (a + \alpha \cdot \mathfrak{m}^*(a)))}$$

which will be used for the boundary constraint checking that the public memory is valid (see [3, 2] for details). The product

$$\prod_{a \in A^*}(z - (a + \alpha \cdot \mathfrak{m}^*(a)))$$

is computed by `get_public_memory_product` and is obtained by simply multiplying the cumulative product of the main page (returned by `get_page_product`) with the cumulative product of the other pages (returned by `get_continuous_pages_product`).

3.5.3 File `public_verify.cairo`

Segment list. This file defines the list of the segments which are in the public input for the considered layout. This list is implemented via the namespace `segments`. In practice, the segments `PROGRAM` and `EXECUTION` will be always present since they correspond to the program segment (holding the bytecode of the proven Cairo program) and the execution segment (holding the intermediate variables of the execution). These segments are respectively defined over the domains $[\texttt{pc}_I, \texttt{pc}_F]$ and $[\texttt{ap}_I, \texttt{ap}_F]$ with $\texttt{pc}_I$, $\texttt{pc}_F$ the initial and final states of the program counter, and $\texttt{ap}_I$, $\texttt{ap}_F$ the initial and final states of the allocation pointer. The other segments correspond to builtin segments.

Builtin list. The function `get_layout_builtins` outputs the list of the builtins present in the considered layout. Each builtin is represented by a short string literal with the builtin name. This list should be consistent with the namespace `segment` and is further terminated by a zero cell.

> **Observation 17: Magic number**
>
> In the function `get_layout_builtins`, the number of builtins is hardcoded.
>
> **Recommendation:** We recommend to defining the number of builtins as
>
> `segments.N_SEGMENTS - 2.`
>
> with a comment explaining that the value 2 corresponds to the program and execution segments.

Public input validation. The function `public_input_validate` aims to validate the public input. It takes as input a `PublicInput` instance and a `StarkDomains` instance and performs the following sanity checks:

- it checks that the number N of Cairo states is upper bounded as:

$$\log_2 N \leq \texttt{MAX\_LOG\_N\_STEPS} ,$$

  where `MAX_LOG_N_STEPS` is equal to 50 in the audited files;

- it checks that the range-check configuration is valid:

$$0 \leq \texttt{rc\_min} \leq \texttt{rc\_max} \leq \texttt{MAX\_RANGE\_CHECK} := 2^{16} - 1$$

- it checks that the layout identifier in the public input corresponds to the identifier of the considered layout;

- it checks that
$$N \times \texttt{CPU\_COMPONENT\_HEIGHT} = L$$
where for this check N is obtained from the `PublicInput` instance while L is obtained from the `StarkDomains` instance. This single check ensures that the STARK domains are consistent with the public input since we already know that the STARK domains are self-consistent thanks to the function `stark_domains_create`.

- for each builtin, it checks that the number of used instances is less than the maximum number of instances:
$$0 \leq \underbrace{\frac{\texttt{ptr}_F - \texttt{ptr}_I}{\texttt{INSTANCE\_SIZE}}}_{\text{Nb used instances}} \leq \underbrace{\frac{N}{\texttt{BUILTIN\_RATIO}}}_{\text{Max nb instances}},$$
where $\texttt{ptr}_I$ and $\texttt{ptr}_F$ are the initial and final states of the builtin pointer. While the builtin ratio is a layout parameter, the instance size is a constant depending on the builtin. Table 5 lists all the builtins used in the audited files and their corresponding instance sizes. Let us remark that the above verification assumes that the size of the builtin segment is a multiple of the instance size, which shall always be the case for a correctly implemented Cairo program (note that this correct behavior can be verified from the statement definition). If this is not the case, the above test shall fail anyway (since the left-hand ratio shall then be large on $\mathbb{F}_p$).

Existing builtin	Instance Size
Output	1
Pedersen	3
Range-Check	1
ECDSA	2
Bitwise	5
Keccak	16
Pedersen	3
EC Op	7

Table 5: Instance sizes for the builtins

■ **Observation 18: Magic numbers**

The instance sizes of the builtins are hardcoded.

Recommendation: We recommend to defining constants for these sizes.

3.6 Layout constraints

A CPU layout involves a set of AIR constraints. In the STARK proof system, those constraints come into play when computing the composition polynomial

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

where $\{C_j\}_j$ are the AIR constraints on the trace (of degrees $\{D_j\}_j$) and D the smallest power of two which is greater than all the constraint degrees.

In the verification process, the constraints $\{C_j\}_j$ are only involved when the verifier evaluates h on the OODS point z after receiving the mask $\{y_\ell\}_\ell$.

3.6.1 File `diluted.cairo`

This file provides the function `get_diluted_prod` which computes the diluted component product used for the layout constraints of the bitwise builtin. Given $(n, \texttt{spacing}, z, \alpha)$, this function computes the term r_{2^n} of the sequence

$$\begin{cases} r_1 = 1 \\ \forall j, r_{j+1} = r_j \cdot (1 + z \cdot u_j) + \alpha \cdot u_j^2 \end{cases}$$

where

$$\forall j := 2^i \cdot (2k+1),\ u_j = u_{2^i} = u_{2^{i-1}} + 2^{i \cdot \texttt{spacing}} - 2^{(i-1)\cdot \texttt{spacing}+1}$$

As detailed in Appendix B, the function `get_diluted_prod` correctly implements the above formula (which was inferred from the comment in the code). Note that the soundness of the AIR constraints is out of scope of the present audit so we did not further check the soundness of the above formula.

3.6.2 File `global_values.cairo`

This file defines the structure `GlobalValues` which accumulates all the required information to evaluate the composition polynomial for the considered layout with the autogenerated function `eval_composition_polynomial`. It also defines a wrapper `InteractionElements` for the interaction elements.

3.6.3 File `composition.cairo`

To evaluate the composition polynomial h in the OODS point z using Equation (1), one needs the *mask* $\{y_\ell\}_\ell$ which is a set of values of the form $f_j(zg^s)$ which are involved in the expressions of the $\{C_j(z)\}_j$.

From the OODS point z, the mask values $\{y_\ell\}_\ell$ and some constraints-related coefficients, the function `traces_eval_composition_polynomial` computes $h(z)$. To proceed, it needs to collect and format some information about the constraints:

- thanks to the function `get_public_memory_product_ratio`, it computes the ratio

$$\frac{z^S}{(z-(a_{\text{pad}}+\alpha\cdot\mathfrak{m}^*(a_{\text{pad}})))^{S-|A^*|}\cdot\prod_{a\in A^*}(z-(a+\alpha\cdot\mathfrak{m}^*(a)))}$$

  used for the boundary constraint checking that the public memory is valid.

- thanks to the function `get_diluted_prod`, it computes the diluted component product used for the constraints of the bitwise builtin (if the bitwise builtin is used in the considered layout).

- thanks to the autogenerated functions in `periodic_columns.cairo`, it evaluates periodic columns for the constraints of the builtins which are in the selected layout.

- it compiles all the necessary values to compute constraints in a structure named `GlobalValues` defined in `globalvalues.cairo`.

It then calls the autogenerated function `eval_composition_polynomial` (from the file `autogenerated.cairo`) with the above structure together with the mask values, the constraint coefficients, and the point z, and gets the desired value $h(z)$.

3.7 DEEP composition polynomial

Let us recall that the DEEP composition polynomial is the polynomial p_0 defined as

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

This polynomial is further called the *OODS polynomial*. Once the FRI layer polynomials $\{p_i\}_i$ have been committed, the IOP verifier sends a list $\mathcal{Q}_0$ of queries and the IOP prover must evaluate the DEEP composition polynomial (or OODS) polynomial p_0 on those queries.

3.7.1 File `oods.cairo`

Evaluating the DEEP composition polynomial. The function

`eval_oods_boundary_poly_at_points`

aims to evaluate the DEEP composition polynomial p_0. This function takes as inputs

- `points`: the list of evaluation points in $\mathcal{Q}_0$,
- `decommitment`: the evaluations of f_i in these points:

$$\{(f_1(v),\dots,f_{W-1}(v)),\ v\in\mathcal{Q}_0\}\,,$$

- `composition_decommitment`: the evaluations of h_i in these points:

$$\{(h_1(v), \ldots, h_{M_2-1}(v)),\ v \in \mathcal{Q}_0\}\,,$$

- `eval_infos`: a structure with members
  - `eval_infos.oods_point`: the OODS point z,
  - `eval_infos.oods_values`: the OODS values $\{y_\ell\}_\ell \cup \{\hat{y}_\ell\}_\ell$,
  - `eval_infos.constraint_coefficients`: the OODS coefficients $\{\gamma_i\}_i$.

The function then returns the evaluations of p_0 for the input queries:

$$\{p_0(v), v \in \mathcal{Q}_0\}\,.$$

In practice, using `eval_oods_boundary_poly_at_points_inner`, the function loops on the queries and calls the autogenerated evaluation function `eval_oods_polynomial` (from the file `autogenerated.cairo`) for each query. More precisely, given a query $v \in \mathcal{Q}_0$, the OODS point z, the OODS values $\{y_\ell\}_\ell \cup \{\hat{y}_\ell\}_\ell$, the OODS coefficients $\{\gamma_i\}_i$ and the evaluations $(f_0(v), \ldots, f_{W-1}(v))$ and $(h_0(v), \ldots, h_{M_2-1}(v))$, the autogenerated function `eval_oods_polynomial` returns $p_0(v)$.

Observation 19: Inconsistent choice of function parameters

In `eval_oods_boundary_poly_at_points_inner`, the number `n_original_columns` of trace columns before sampling interaction elements is a function parameter, while the number `n_interaction_columns` is not.

Recommendation: We recommend homogenizing the implementation by removing `n_original_columns` from the function parameters.

3.8 Cairo verifier

3.8.1 File `layout.cairo`

The core STARK verifier (Section 3.4) takes as input some layout-dependent functions which are called during the verification process. However, these functions sometimes need layout-dependent data besides the inputs provided by the core STARK verifier. This additional data is stored in a structure `Layout` defined in the file `layout.cairo`.

The virtual functions of `AirInstance` take as first argument a pointer to an `AirInstance` object. To grant them access to the additional data, the idea is to store a `Layout` object right after the `AirInstance` object in the memory. Namely, if `air` denotes the pointer to the `AirInstance` object, the `Layout` object shall be stored at address

$$\texttt{air} + \texttt{AirInstance.SIZE}\,.$$

The file defines the structure `AirWithLayout` as a wrapper containing an `AirInstance` object followed by a `Layout` object.

The members of the `Layout` structure are the following:

- `eval_oods_polynomial`: the autogenerated function to evaluate the DEEP composition polynomial,
- `n_original_columns`: the number of columns in the execution trace before sampling interaction elements,
- `n_interaction_columns`: the number of columns in the execution trace added after sampling interaction elements,
- `n_interaction_elements`: the number of interaction elements.

In the same way as for `AirInstance`, the file implements a function `eval_oods_polynomial` which performs an absolute jump to the function with the same name in the input `Layout` structure.

3.8.2 File `verify.cairo`

This file implements the *CPU verifier*, *i.e.* the second verification level abstracted in Section 2.2.2. The entry point of this level is the function `verify_proof`. It takes as input a proof (stored in a `StarkProof` object) and calls the function `build_air`. The latter gathers the layout-dependent information required by the core STARK verifier (trace format, AIR constraints, etc.) in a `AirInstance` structure as well as the additional data required by the `AirInstance` functions in a structure `Layout`, both wrapped in a `AirWithLayout` object. Then, the function `verify_proof` simply runs the STARK verification process by calling `verify_stark_proof`.

3.8.3 File `cairo_verifier.cairo`

This file implements the Cairo verifier, *i.e.* the third verification level abstracted in Section 2.2.2. This level defines the pages of the public memory on top of the CPU verifier.

Program builtin list. The function `get_program_builtins` outputs the list of the builtins used in the considered Cairo program. Each builtin is represented by a short string literal with the builtin name, as for the function `get_layout_builtins` of `public_verify.cairo`. This list is always terminated by a zero cell. The list of layout builtins must be a subset of the program builtins, ordered in the same way.

Public input. As explained in Section 2 (and Remark 1), the audited verifier implementation fixes the number of continuous pages of the public memory $\mathfrak{m}^*$ to 0. This means that $\mathfrak{m}^*$ is entirely defined with the main page. For this reason, the output builtin segment is included to the main page, as opposed to the audited Solidity implementation of the verifier for which it was part of the continuous pages [1].

The function `_verify_public_input` performs some sanity checks and verifies that the public input satisfies the format described in Table 1. Specifically, this function:

- checks that the given public input is made of a single page (the main page),

- checks that the main page starts with the program bytecode, with first instructions:

  $$\texttt{ap += n\_args; call main; jmp rel 0.}$$

  where `n_args` is the number of program builtins,

- checks that the cells following the program bytecode are set to
  - $\mathfrak{m}^*(\texttt{fp}_I - 2) = \texttt{fp}_I$
  - $\mathfrak{m}^*(\texttt{fp}_I) = 0$

- checks that
  - $\mathfrak{m}^*(\texttt{fp}_I) \ldots \mathfrak{m}^*(\texttt{fp}_I + \texttt{n\_args} - 1)$ matches the initial state of the program builtin pointers,
  - $\mathfrak{m}^*(\texttt{ap}_F - \texttt{n\_args}) \ldots \mathfrak{m}^*(\texttt{ap}_F - 1)$ matches the final state of the builtin pointers.

  This is done by calling the auxiliary function `verify_stack`. If a program builtin is not included in the list of layout builtins, then the start and end pointers for this builtin are set to zero.

- checks that the initial/final program counter and allocation pointer satisfy:
  - $\texttt{pc}_I = \texttt{INITIAL\_PC} := 1$,
  - $0 \leq \texttt{ap}_I < \texttt{MAX\_ADDRESS} + 1$,
  - $0 \leq \texttt{ap}_F < \texttt{MAX\_ADDRESS} + 1$.

We note that the implementation of `_verify_public_input` implicitly assumes that

$$\texttt{segments.PROGRAM} = 0 \quad \text{and} \quad \texttt{segments.EXECUTION} = 1$$

since it assumes that the builtin segment starts from the index 2 (see lines 112 and 118 of the source code). Table 6 represents the format of the memory which is ensured by these different checks.

In addition to checking the memory, the function `_verify_public_input` further computes

- the Pedersen hash digest of the program bytecode (with the booting instructions calling the `main` function),
- the Pedersen hash digest of the values in the memory segment of the output builtin.

Those two digests are returned by the function (if all the sanity checks pass).

Observation 20: Unexpected behaviour

If the function `verify_stack` encounters a layout builtin that is not in the list of the program builtins, all the following memory values will be set to zero without raising an error.

Recommendation: We recommend raising an error when there exists a layout

builtin that is not in the list of the program builtins.

Main entry point. This file further contains the `main` function of the Cairo verifier. The function:

- loads the proof (with the public input) in a `StarkProof` object thanks to a hint parsing the program input,
- calls the function `verify_cairo_proof` which:
  - runs the CPU verifier using `verify_proof` with a security level of

    $$\texttt{SECURITY\_BITS} := 80 \text{ bits},$$

  - verifies that the public input is valid and complies with Cairo's public memory format using `_verify_public_input`.
- outputs (*i.e.* writes in the output memory segment):
  - the hash digest of the bytecode of the proven Cairo program,
  - the hash digest of the output memory segment of the proven Cairo program

  both packed in a `CairoVerifierOutput` object.

$\texttt{pc}_I := 1$	`0x40780017fff7fff`
	Number program builtins
	`0x1104800180018000`
	(no check)
	`0x10780017fff7fff`
	`0x0`
	rest of the bytecode
$\texttt{fp}_I - 2$	$\texttt{fp}_I$
	`0x0`
$\texttt{ap}_I = \texttt{fp}_I$	Start pointer for builtin 1
	Start pointer for builtin 2
	⋮
$\texttt{fp}_I + (n-1)$	Start pointer for builtin n
	(no check)
$\texttt{ap}_F - n$	End pointer for builtin 1
	End pointer for builtin 2
	⋮
$\texttt{ap}_F - 1$	End pointer for builtin n

Table 6: Format of the memory as ensured by the function `_verify_public_input`, with $n = \texttt{n\_args}$ the number of program builtins.

References

[1] CryptoExperts (Thibauld Feneuil and Matthieu Rivain). Code Review of StarkWare's EVM STARK Verifier, December 2021.

[2] CryptoExperts (Thibauld Feneuil and Matthieu Rivain). Code Review of the Cairo & SHARP Verifiers, July 2022.

[3] 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.

[4] 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.

[5] StarkWare. Stark math. Medium, 2019. https://medium.com/starkware/tagged/stark-math.

[6] StarkWare. ethstark documentation. Cryptology ePrint Archive, Report 2021/582, 2021. https://ia.cr/2021/582.

[7] StarkWare. StarkNet and Cairo Documentation, 2022. https://www.cairo-lang.org/docs/.

A FRI folding formula

As explained in Section 3.3.1, the term

$$\mathsf{Combine}_j(\{e_0,\ldots,e_{2^j-1}\},\zeta,v) := \sum_{q=0}^{2^j-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^j-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_k$$

follows a recursive relation

$$\mathsf{Combine}_j(\{e_0,\ldots,e_{2^j-1}\},\zeta,v) = \mathsf{Combine}_1(\{e'_0,e'_1\},\zeta^{2^{j-1}},v^{2^{j-1}})$$

where

$$\begin{aligned} e'_0 &:= \mathsf{Combine}_{j-1}(\{e_0,\ldots,e_{2^{j-1}-1}\},\zeta,v)\\ e'_1 &:= \mathsf{Combine}_{j-1}(\{e_{2^{j-1}},\ldots,e_{2^j-1}\},\zeta,v\cdot\xi). \end{aligned}$$

Here is below the proof of correctness of this recursive relation.

Initial case. When $j=1$, we directly get that

$$\mathsf{Combine}_1(\{e_0,e_1\},\zeta,v) := (e_0+e_1)+\zeta\cdot v^{-1}\cdot(e_0-e_1).$$

Induction. Let us take $j>1$. By induction assumption, we have

$$\begin{aligned} e'_0 &= \sum_{q=0}^{2^{j-1}-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^{j-1}-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_k\\ e'_1 &= \sum_{q=0}^{2^{j-1}-1}\left(\frac{\zeta}{v\cdot\xi}\right)^q \cdot \sum_{k=0}^{2^{j-1}-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_{2^{j-1}+k}\\ &= \sum_{q=0}^{2^{j-1}-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=2^{j-1}}^{2^j-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_k \end{aligned}$$

since $\xi^{\mathsf{rev}(k)}\cdot\xi = \xi^{\mathsf{rev}(k+2^{j-1})}$. We then get

$$\begin{aligned} e'_0+e'_1 &= \sum_{q=0}^{2^{j-1}-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^j-1}(\xi^{-\mathsf{rev}(k)})^q \cdot e_k\\ e'_0-e'_1 &= \sum_{q=0}^{2^{j-1}-1}\left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^j-1}(\xi^{-\mathsf{rev}(k)})^{2^{j-1}+q} \cdot e_k \end{aligned}$$

since $(\xi^{-\mathsf{rev}(k)})^{2^{j-1}+q} = (\xi^{-\mathsf{rev}(k)})^{2^{j-1}} \cdot (\xi^{-\mathsf{rev}(k)})^q = \begin{cases} (\xi^{-\mathsf{rev}(k)})^q & \text{if } k < 2^{j-1} \\ -(\xi^{-\mathsf{rev}(k)})^q & \text{if } k \geq 2^{j-1} \end{cases}$. Finally, we can deduce that

$$\begin{aligned} \mathsf{Combine}_2(\{e'_0, e'_1\}, \zeta^{2^{j-1}}, v^{2^{j-1}}) &= (e'_0 + e'_1) + \left(\frac{\zeta}{v}\right)^{2^{j-1}} \cdot (e'_0 - e_1) \\ &= \sum_{q=0}^{2^{j-1}-1} \left(\frac{\zeta}{v}\right)^q \cdot \sum_{k=0}^{2^{j-1}-1} (\xi^{-\mathsf{rev}(k)})^q \cdot e_k \\ &= \mathsf{Combine}_j(\{e_0, \ldots, e_{2^j-1}\}, \zeta, v). \end{aligned}$$

B Diluted component product

The file `diluted.cairo` provides a function `get_diluted_prod` which computes the diluted component product used for the layout constraints of the bitwise builtin. More precisely, given $(n, \texttt{spacing}, z, \alpha)$, it computes the term r_{2^n} of the sequence

$$\begin{cases} r_1 = 1 \\ \forall j, r_{j+1} = r_j \cdot (1 + z \cdot u_j) + \alpha \cdot u_j^2 \end{cases}$$

where $u_1 = 1$ and

$$\begin{aligned} \forall j := 2^i \cdot (2k+1),\ u_j = u_{2^i} &= u_{2^{i-1}} + 2^{i \cdot \texttt{spacing}} - 2^{(i-1)\cdot\texttt{spacing}+1} \\ &= u_{2^{i-1}} + 2^{(i-1)\cdot\texttt{spacing}} \cdot (2^{\texttt{spacing}} - 2). \end{aligned}$$

We have for all j,

$$r_{2^j} = \underbrace{\prod_{k=1}^{2^j-1} (1 + z \cdot u_k)}_{p_j := \ldots} + \alpha \cdot \underbrace{\sum_{k=1}^{2^j-1} (u_k^2 \cdot \prod_{\ell=k+1}^{2^j-1} (1 + z \cdot u_\ell))}_{q_j := \ldots}.$$

For $k \in \{2^j + 1, \ldots, 2^{j+1} - 1\}$, we have $u_k = u_{k-2^j}$ (since u_i only depends on the number of trailing zeros in the binary representation of i), and so we have

$$\prod_{k=2^j+1}^{2^{j+1}-1} (1 + z \cdot u_k) = p_j.$$

Thus, we get

$$\begin{aligned}
p_{j+1} &= p_j \cdot (1 - z \cdot u_{2^j}) \cdot p_j \\
q_{j+1} &= \sum_{k=1}^{2^j-1} \left(u_k^2 \cdot \prod_{\ell=k+1}^{2^{j+1}-1} (1 + z \cdot u_\ell) \right) + u_{2^j}^2 \cdot p_j + \sum_{k=2^j+1}^{2^{j+1}-1} \left(u_k^2 \cdot \prod_{\ell=k+1}^{2^{j+1}-1} (1 + z \cdot u_\ell) \right) \\
&= q_j \cdot (1 + z \cdot u_{2^j}) \cdot p_j + u_{2^j}^2 \cdot p_j + \sum_{k=1}^{2^j-1} \left(u_k^2 \cdot \prod_{\ell=k+1}^{2^j-1} (1 + z \cdot u_\ell) \right) \\
&= q_j \cdot (1 + z \cdot u_{2^j}) \cdot p_j + u_{2^j}^2 \cdot p_j + q_j
\end{aligned}$$

with $(p_0, q_0) = (1, 0)$ and $(p_1, q_1) = (1 + z, 1)$.

Let us denote n the value of `n_bits` when calling `get_diluted_prod`. Then the auxiliary function `get_diluted_prod_inner` takes as input

- a counter c,
- the term $\texttt{diff\_x}_{n-c}$ of the sequence $\texttt{diff\_x}_i := 2^{(i-1)\cdot \texttt{spacing}} \cdot (2^{\texttt{spacing}} - 2)$,
- the term $\texttt{x}_{n-c} := u_{2^{n-c-1}}$,
- the term p_{n-c},
- the term q_{n-c},

computes

$$\begin{aligned}
\texttt{x}_{n-c+1} &= \texttt{x}_{n-c} + \texttt{diff_x}_{n-c} \\
\texttt{diff_x}_{n-c+1} &= \texttt{diff_x}_{n-c} \cdot \texttt{diff_multiplier} \\
p_{n-c+1} &= p_{n-c} \cdot (p_{n-c} + z \cdot \texttt{x}_{n-c+1} \cdot p_{n-c}) \\
q_{n-c+1} &= q_{n-c} \cdot (p_{n-c} + z \cdot \texttt{x}_{n-c+1} \cdot p_{n-c}) + \texttt{x}_{n-c+1} \cdot (\texttt{x}_{n-c+1} \cdot p_{n-c}) + q_{n-c}
\end{aligned}$$

and outputs

$$(p_n, q_n)$$

after doing a recursive call. Finally, the function `get_diluted_prod` outputs

$$p_n + \alpha \cdot q_n$$

which corresponds to the desired value.