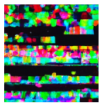


ZKSECURITY

Audit of Herodotus' Integrity - A Cairo Verifier compatible with Starknet written in Cairo 1

September 23rd, 2024

Introduction

On September 23rd, 2024, Herodotus requested an audit of their Integrity verifier, which is a Cairo verifier deployed on Starknet. The audit was conducted by three consultants over the course of four weeks. The main focus of the audit was the soundness of the protocol by looking in depth at the STARK and FRI protocol it uses and comparing it to the reference implementation from Starkware.

Scope

The scope of the audit included the following components:

- *.cairo files under `integrity/src`
- *.cairo files under `cairo-lang/src/starkware/cairo/cairo_verifier` and `cairo-lang/src/starkware/cairo/stark_verifier` were provided as reference implementation

Methodology

First, we explored in depth the Cairo, STARK, and FRI protocols used by the Integrity verifier.

Second, the Herodotus team used Starkware's existing Cairo verifier implementation as reference, so we checked for unintended divergences from that implementation. This step involved checking both the implementation logic and the language differences, since Starkware's verifier is written in Cairo 0 while Integrity is written in Cairo 1.

Third, we tracked the following added features from the reference implementation:

- split contracts
- split proofs
- support verifying Cairo 1 programs (reference implementation only verifies Cairo 0 programs)
- support stone prover v6 proofs

Divergences from Starkware's verifier

Splitting contracts

Starknet, like other blockchains, has various physical constraints in processing transactions (see https://docs.starknet.io/tools/limits-and-triggers/).

	Description	Mainnet
Max Cairo steps for transaction	The maximum number of computational steps, measured in Cairo steps, that a transaction can contain.	10,000,000
Max calldata length		4,000 felts
Max contract class size	The maximum size for a contract class within Starknet.	

4,089,446 bytes

To overcome these limitations, Integrity divided the verifier into multiple contracts. The autogenerated code responsible for polynomial evaluation in each layout is the largest part in terms of size. For example, in https://github.com/HerodotusDev/integrity/blob/main/src/air/layouts/small.cairo#L35, the implementation of `eval_oods_polynomial_inner()` has been separated into a different contract, `src/air/layouts/small/autogenerated.cairo`.

```
#[cfg(feature: 'monolith')]
use cairo_verifier::air::layouts::small::autogenerated::{
    eval_oods_polynomial_inner as eval_oods_polynomial_inner_,
};

#[cfg(feature: 'monolith')]
fn eval_oods_polynomial_inner(
    column_values: Span<felt252>,
    oods_values: Span<felt252>,
    constraint_coefficients: Span<felt252>,
    point: felt252,
    oods_point: felt252,
    trace_generator: felt252,
    contract_address: ContractAddress,
) -> felt252 {
    eval_oods_polynomial_inner_(
        column_values, oods_values, constraint_coefficients, point, oods_point, trace_generator,
    )
}
```

This structure appears in various layouts such as `recursive` and `starknet_with_keccak`.

Splitting proofs

When transmitting proofs through calldata, the size limit of calldata becomes an issue. Since the FRI witness accounts for a large portion of the proof size, the FRI witness is divided into multiple steps and verified through separate transactions. The code for splitting the proof is implemented in https://github.com/HerodotusDev/integrity-calldata-generator. The basic idea is that a user sequentially calls `verify_proof_initial`, `verify_proof_step`,

and `verify_proof_final` with divided proofs in separate transactions. In each transaction, information about the partial proof is stored in state variables.

```
// src/contracts/verifier.cairo

fn verify_proof_step(
    ref self: ContractState,
    job_id: JobId,
    state_constant: FriVerificationStateConstant,
    state_variable: FriVerificationStateVariable,
    witness: FriLayerWitness,
) -> (FriVerificationStateVariable, u32) {
    ...

    let (con, var) = StarkProofImpl::verify_step(
        state_constant, state_variable, witness, @settings
    );
    self.state_variable.entry(job_id).write(Option::Some(hash_variable(@var)));

    let layers_left = con.n_layers - var.iter;

    (var, layers_left)
}
```

Only after all partial proofs have been transmitted can the verification be performed.

Supporting verifying Cairo 1 programs

Cairo 0 programs have a simple public memory structure that allows the verifier to check the following:

1. The program contains a set of fixed instructions at the beginning.
2. The first and final instructions of the program are fixed.
3. The arguments and return values of the main function are builtin pointers that are specified in the public input.

Cairo 1 programs have a more complex public memory structure compared to Cairo 0 programs, making them more difficult to verify.

For example, Cairo 1 programs create an additional `entry code` function that is called before the `main` function in order to handle language particulars that are unique to Cairo 1 programs, such as dictionaries and gas builtin. It also handles builtin pointers differently compared to Cairo 0 programs.

Another example is the structure of the output segment of the public memory. Cairo 1 programs accept inputs to the main function, which are then serialized into the output segment of the public memory. Cairo 1 programs are allowed to panic, which means that a valid trace of the program can be created even though the program fails during

execution. This is handled by adding an extra value at the output segment of the public memory indicating whether the program has panicked or not.

Since Cairo 1 programs have a more complex public memory structure, the `integrity` verifier opts removes some of the checks that are performed on the public memory for Cairo 0 programs.

Supporting Stone Prover v6 proofs

The Stone Prover v6 (**https://github.com/starkware-libs/stone-prover/commit/1414a545e4fb38a85391289abe91dd4467d268e1**) has two major differences from v5:

1. When creating the initial seed for the Fiat-Shamir used in the STARK protocol, the prover hashes the public input of the proof. In v6, the `n_verifier_friendly_commitment_layers` value is added in addition to the values added in v5.

```
if *settings.stone_version == StoneVersion::Stone6 {
    hash_data.append(n_verifier_friendly_commitment_layers);
}
```

2. For the commitment hash used in the STARK protocol, v5 only supported 160-bit masks, while v6 also supports 248-bit masks.

```
let (hasher, hasher_bit_length) = if verifier_config.hasher == 'keccak_160_lsb' {
    ('keccak', HasherBitLength::Lsb160)
} else if verifier_config.hasher == 'keccak_248_lsb' {
    ('keccak', HasherBitLength::Lsb248)
} else if verifier_config.hasher == 'blake2s_160_lsb' {
    ('blake2s', HasherBitLength::Lsb248)
} else {
    assert(verifier_config.hasher == 'blake2s_248_lsb', 'Unsupported hasher variant');
    ('blake2s', HasherBitLength::Lsb248)
};
```

Recommendations

Based on the findings of the audit, we recommend the following steps to enhance the security and robustness of the Integrity verifier.

While the codebase was found to be generally well-designed and implemented in Cairo, there is significant room for improvement in the area of documentation. Specifically, clearer specification of the protocols used, such as FRI and other cryptographic primitives, would significantly enhance code readability, maintainability, and auditability. We mention this in more detail in finding **Lack Of Specification Makes It Difficult To Verify The Implementation**. As part of this work we started writing specifications for the STARK, FRI, Channel, and Hash-based Polynomial Commitment

schemes, which can be found at **https://zksecurity.github.io/RFCs/**. Included after this section are the specifications at the moment of this writing. Refer to the previous link for an up-to-date version. We strongly recommend that the Herodotus team continue to improve these specifications and consider performing another audit once they are in a better place (including adding a Cairo specification to address finding **Cairo Public Input Validation Might Not Be Sufficient**).

In addition we observed that the existing tests and integration tests cover only the most basic scenarios, leaving room for significant improvement in test coverage. A notable example is the splitted proof verification process, which involves verifying divided proofs in sequential stages. Although the documentation provides an explanation about how the integrity verifier splits proofs and a script for splitting and verifying proofs, this crucial component is not only separated into an independent repository but is also absent from the integration tests. More comprehensive tests should be included to verify proper functionality, including easier ways to generate and (de)serialize proofs in different ways to create different ways to verify them or to implement negative tests (e.g. a malformed proof should not be verifiable).

Finally, layout codes rely on calculations from auto-generated code, which is generated using a Python script `src/air/layouts/_generator/main.py` that converts the Cairo 0 code from the reference implementation into Cairo 1 code. Given the critical importance of this generator code, we strongly recommend implementing additional verification procedures for this component, and considering performing an audit of the generator code itself.

Starknet Channels for Fiat-Shamir Instantiation

For a more up-to-date specification check https://zksecurity.github.io/RFCs/.

Channels are an abstraction used to mimic the communication channel between the prover and the verifier in a non-interactive protocol. It is useful to ensure that all prover messages are correctly absorbed before being used by the verifier, and that all verifier challenges are correctly produced.

Overview

A channel is an object that mimics the communication channel between the prover and the verifier, and is used to abstract the **Fiat-Shamir transformation** used to make the protocol non-interactive.

The Fiat-Shamir transformation works on public-coin protocols, in which the messages of the verifier are pure random values. To work, the Fiat-Shamir transformation replaces the verifier messages with a hash function applied over the transcript up to that point.

A channel is initialized at the beginning of the protocol, and is instantiated with a hash function. It is implemented as a continuous hash that "absorbs" every prover message and which output can be used to produce the verifier's challenges.

Dependencies

A channel is instantiated with the following two dependencies:

- `hades_permutation(s1, s2, s3)`. The Hades permutation which permutes a given state of three field elements.
- `poseidon_hash_span(field_elements)`. The Poseidon sponge function which hashes a list of field elements.

Interface

A channel has two fields:

- A `digest`, which represents the current internal state.
- A `counter`, which helps produce different values when the channel is used repeatedly to sample verifier challenges.

The channel has the following interface:

Initialize. This initializes the channel in the following way:

- Set the `digest` to the given `digest`, which is the prologue/context to the protocol.
- Set the `counter` to 0.

Absorb a message from the prover.

- Resets the `counter` to 0.
- Set the `digest` to `POSEIDON_hash(digest + 1 || value)`.

Absorb multiple messages from the prover.

- Resets the `counter` to 0.
- Set the `digest` to `POSEIDON_hash(digest + 1 || values)`.

This function is not compatible with multiple call to the previous function.

Produce a verifier challenge.

- Produce a random value as `hades_permutation(digest, counter, 2)`.
- Increment the `counter`.

With the current design, two different protocols where one produces n challenges and another that produces m challenges will have the same "transcript" and thus will continue to produce the same challenges later on in the protocol. While there are no issues in this design in the context of Starknet, this might not always be secure when used in other protocols.

Starknet Merkle Tree Polynomial Commitments

For a more up-to-date specification check https://zksecurity.github.io/RFCs/.

Merkle tree polynomial commitments provide a standard on how to commit to a polynomial using a Merkle tree.

Overview

Commitments of polynomials are done using **Merkle trees**. The Merkle trees can be configured to hash some parameterized number of the lower layers using a circuit-friendly hash function (Poseidon).

Dependencies

- the verifier-friendly hash is `hades_permutation(s1, s2, 2)` always setting the last field element to 2
- the default hash is either keccak256 or blake2s

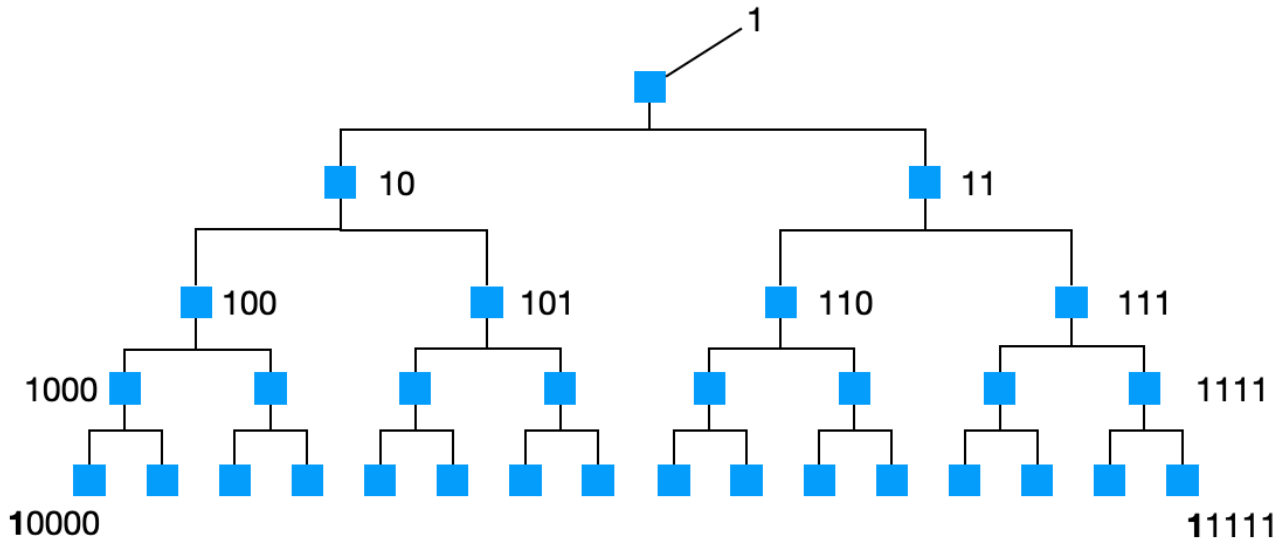
Constants

`MONTGOMERY_R = 3618502788666127798953978732740734578953660990361066340291730267701097005025`. The Montgomery form of $2^{256} \mod \text{STARK\_PRIME}$.

Vector Commitments

A vector commitment is simply a Merkle tree. It is configured with two fields:

- `height`: the height of the Merkle tree
- `n_verifier_friendly_commitment_layers`: the depth at which layers will start using the verifier-friendly hash.



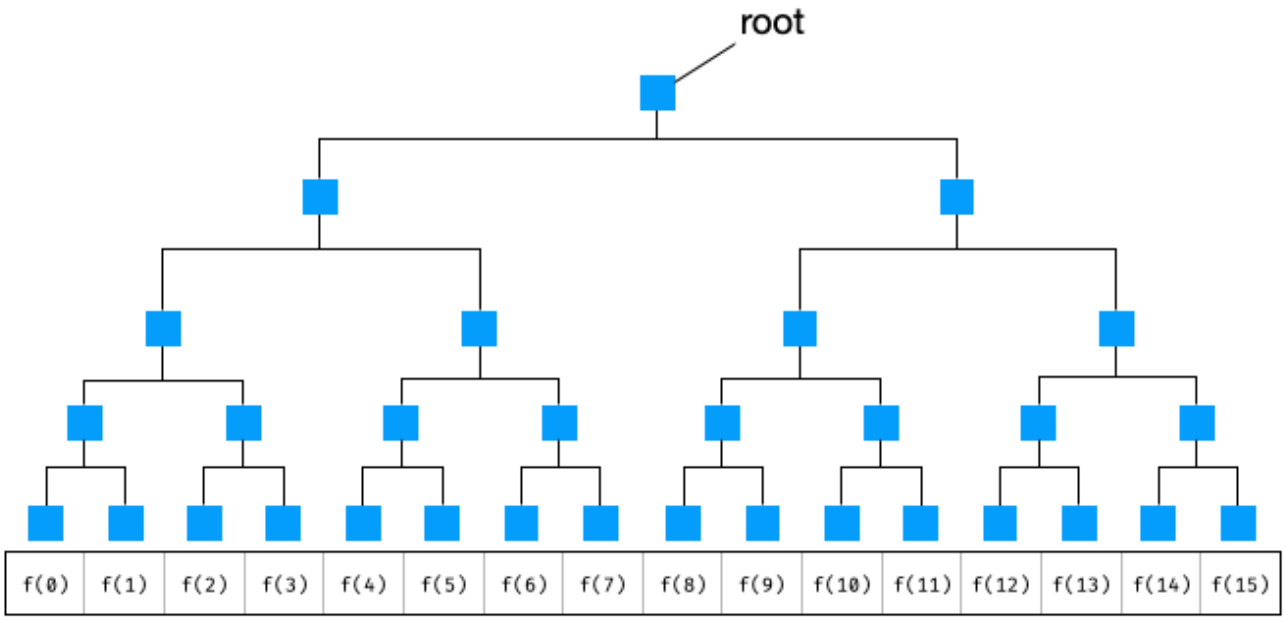


Table Commitments

A table commitment in this context is a vector commitment where leaves are hashes of multiple values. Or in other words, a leaf can be seen as a hash of a table of multiple columns and a single row.

It can be configured with two fields:

- `n_columns`: the number of columns in each leaf of the tree
- `vector`: the vector commitment configuration (see previous section).

A few examples:

- the trace polynomials in the **STARK verifier specification** are table commitments where each leaf is a hash of the evaluations of all the trace column polynomials at the same point
- the composition polynomial in the **STARK verifier specification** is a table commitment where each leaf is a hash of the evaluations of the composition polynomial columns at the same point
- the FRI layer commitments in the **FRI verifier specification** are table commitments where each leaf is a hash of the evaluations of the FRI layer columns at associated points (e.g. v and $-\mathrm{v}$)

Note that values are multiplied to the `MONTGOMERY_R` constant before being hashed as leaves in the tree.

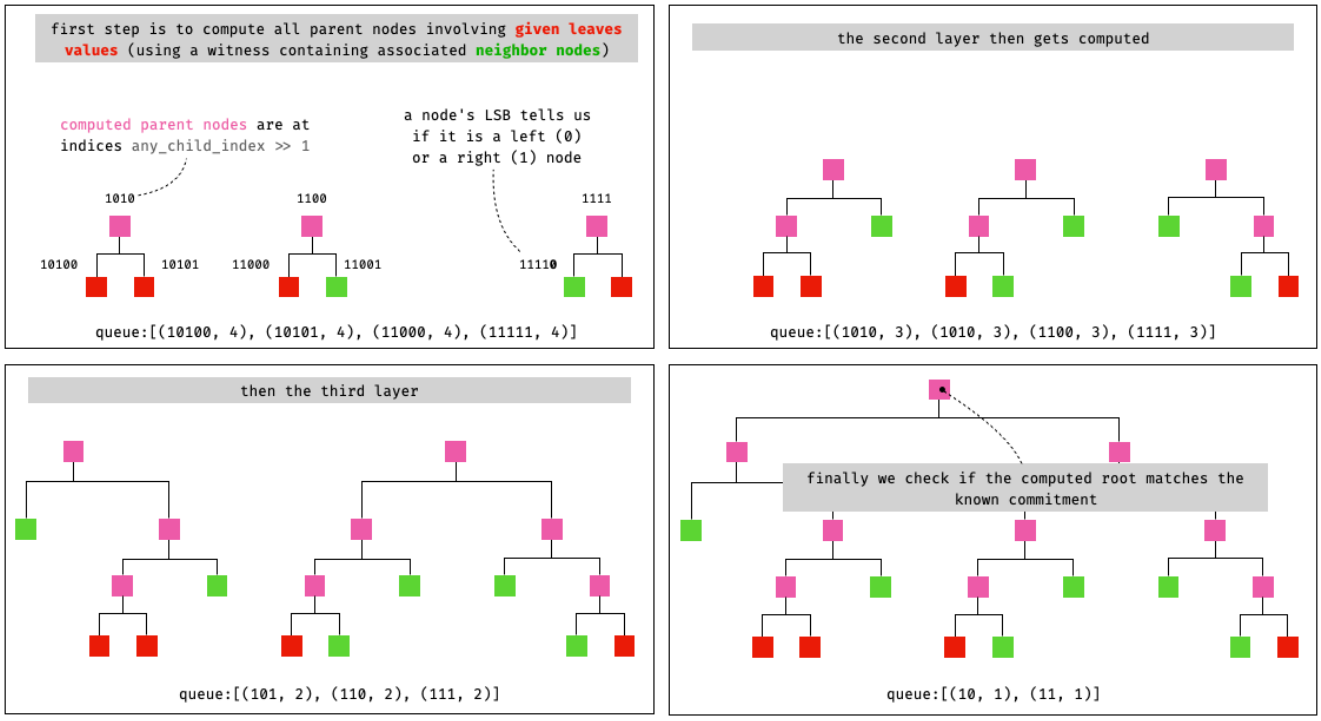
Index to Path Conversion

Random evaluation of the polynomial might produce an index in the range $[0, 2^{\mathrm{h}})$ with h the height of the tree. Due to the way the tree is indexed, we have to convert that index into a path. To do that, the index is added with the value 2^{h} to set its MSB.

For example, the index `0` becomes the path `10000` which correctly points to the first leaf in our example.

Vector Membership Proofs

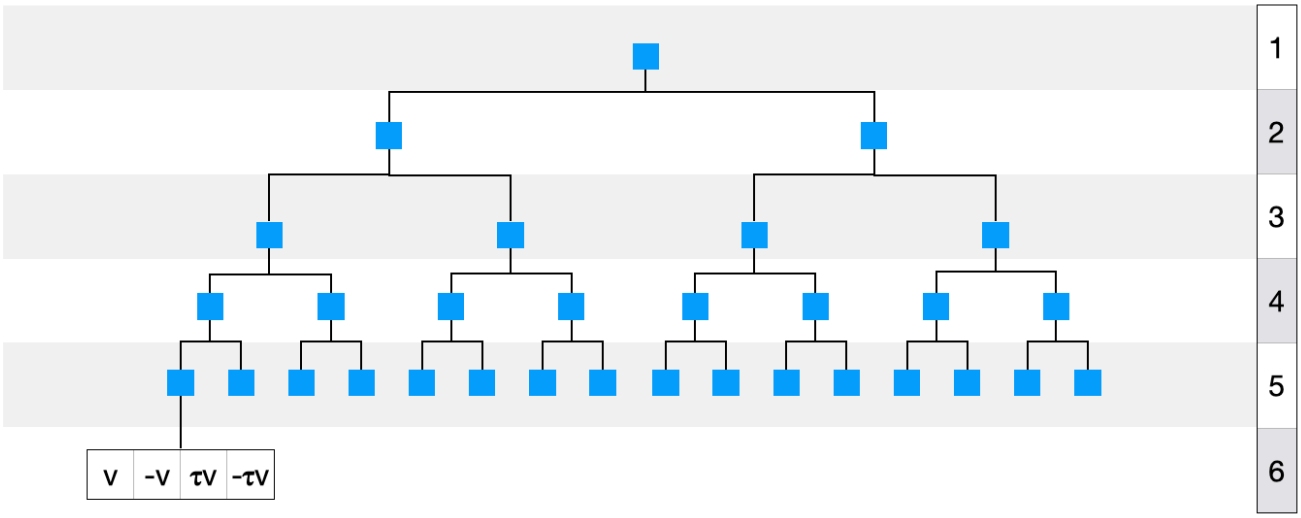
A vector decommitment/membership proof must provide a witness (the neighbor nodes missing to compute the root of the Merkle tree) ordered in a specific way. The following algorithm dictates in which order the nodes hash values provided in the proof are consumed:



Verifier-Friendly Layers

A `n_verifier_friendly_layers` variable can be passed which dictates at which layer the Merkle tree starts using a verifier-friendly hash.

In the following example, the height of the table commitment is 6 (and the height of the vector commitment is 5). As such, a `n_verifier_friendly_layers` of 6 would mean that only the table would use the verifier-friendly hash. A `n_verifier_friendly_layers` of 5 would mean that the last / bottom layer of the Merkle tree would also use the verifier-friendly hash. A `n_verifier_friendly_layers` of 1 would mean that all layers would use the verifier-friendly hash.



Starknet FRI Verifier

For a more up-to-date specification check https://zksecurity.github.io/RFCs/.

The **Fast Reed-Solomon Interactive Oracle Proofs of Proximity (FRI)** is a cryptographic protocol that allows a prover to prove to a verifier (in an interactive, or non-interactive fashion) that a hash-based commitment (e.g. a Merkle tree of evaluations) of a vector of values represent the evaluations of a polynomial of some known degree. (That is, the vector committed is not just a bunch of uncorrelated values.) The algorithm is often referred to as a "low degree" test, as the degree of the underlying polynomial is expected to be much lower than the degree of the field the polynomial is defined over. Furthermore, the algorithm can also be used to prove the evaluation of a committed polynomial, an application that is often called FRI-PCS. We discuss both algorithms in this document, as well as how to batch multiple instances of the two algorithms.

For more information about the original construction, see **Fast Reed-Solomon Interactive Oracle Proofs of Proximity**. This document is about the specific instantiation of FRI and FRI-PCS as used by the StarkNet protocol.

Specifically, it matches the **integrity verifier**, which is a **Cairo 1** implementation of a Cairo verifier. There might be important differences with the Cairo verifier implemented in C++ or Solidity.

Overview

We briefly give an overview of the FRI protocol, before specifying how it is used in the StarkNet protocol.

FRI

Note that the protocol implemented closely resembles the high-level explanations of the **ethSTARK paper**, as such we refer to it in places.

FRI is a protocol that works by successively reducing the degree of a polynomial, and where the last reduction is a constant polynomial of degree 0. Typically the protocol obtains the best runtime complexity when each reduction can halve the degree of its input polynomial. For this reason, FRI is typically described and instantiated on a polynomial of degree a power of 2.

If the reductions are "correct", and it takes n reductions to produce a constant polynomial in the "last layer", then it is a proof that the original polynomial at "layer 0" was of degree at most 2^n.

In order to ensure that the reductions are correct, two mechanisms are used:

1. First, an interactive protocol is performed with a verifier who helps randomize the halving of polynomials. In each round the prover commits to a "layer" polynomial.

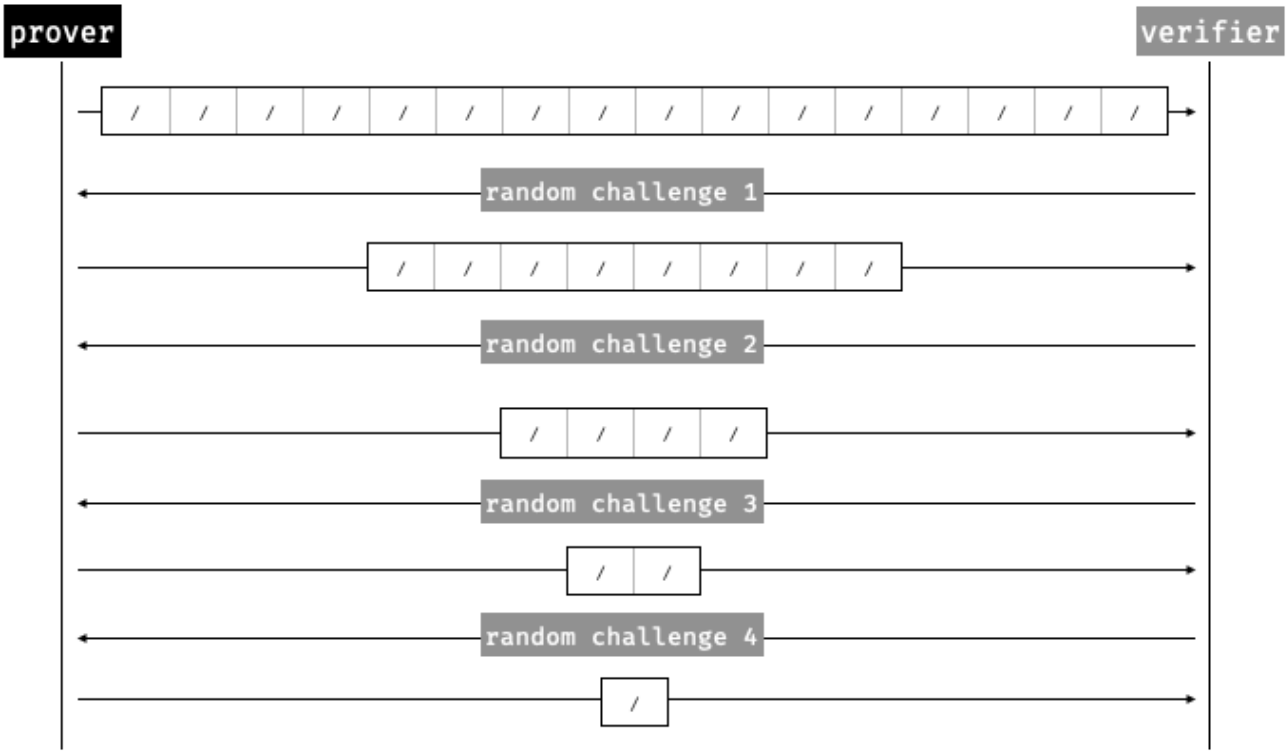
2. Second, as commitments are not algebraic objects (as FRI works with hash-based commitments), the verifier query them in multiple points to verify that an output polynomial is consistent with its input polynomial and a random challenge. (Intuitively, the more queries, the more secure the protocol.)

Setup

To illustrate how FRI works, one can use **sagemath** with the following setup:

```
# We use the starknet field (https://docs.starknet.io/architecture-and-concepts/cryptography/p-value/)
starknet_prime = 2^251 + 17*2^192 + 1
starknet_field = GF(starknet_prime)
polynomial_ring.<x> = PolynomialRing(starknet_field)

# find generator of the main group
gen = starknet_field.multiplicative_generator()
assert gen == 3
assert starknet_field(gen)^(starknet_prime-1) == 1 # 3^(order-1) = 1

# lagrange theorem gives us the orders of all the multiplicative subgroups
# which are the divisors of the main multiplicative group order (which, remember, is p - 1 as 0 is not part of it)
# p - 1 = 2^192 * 5 * 7 * 98714381 * 166848103
multiplicative_subgroup_order = starknet_field.order() - 1
assert list(factor(multiplicative_subgroup_order)) == [(2, 192), (5, 1), (7, 1), (98714381, 1), (166848103, 1)]

# find generator of subgroup of order 2^192
# the starknet field has high 2-adicity, which is useful for FRI and FFTs
# (https://www.cryptologie.net/article/559/whats-two-adicity)
gen2 = gen^( (starknet_prime-1) / (2^192) )
assert gen2^(2^192) == 1

# find generator of a subgroup of order 2^i for i <= 192
def find_gen2(i):
    assert i >= 0
    assert i <= 192
    return gen2^( 2^(192-i) )

assert find_gen2(0)^1 == 1
assert find_gen2(1)^2 == 1
assert find_gen2(2)^4 == 1
assert find_gen2(3)^8 == 1
```

Reduction



A reduction in the FRI protocol is obtained by interpreting an input polynomial p as a polynomial of degree $2n$ and splitting it into two polynomials g and h of degree n such that $p(x) = g(x^2) + xh(x^2)$.

Then, with the help of a verifier's random challenge ζ, we can produce a random linear combination of these polynomials to obtain a new polynomial $g(x) + \zeta h(x)$ of degree n:

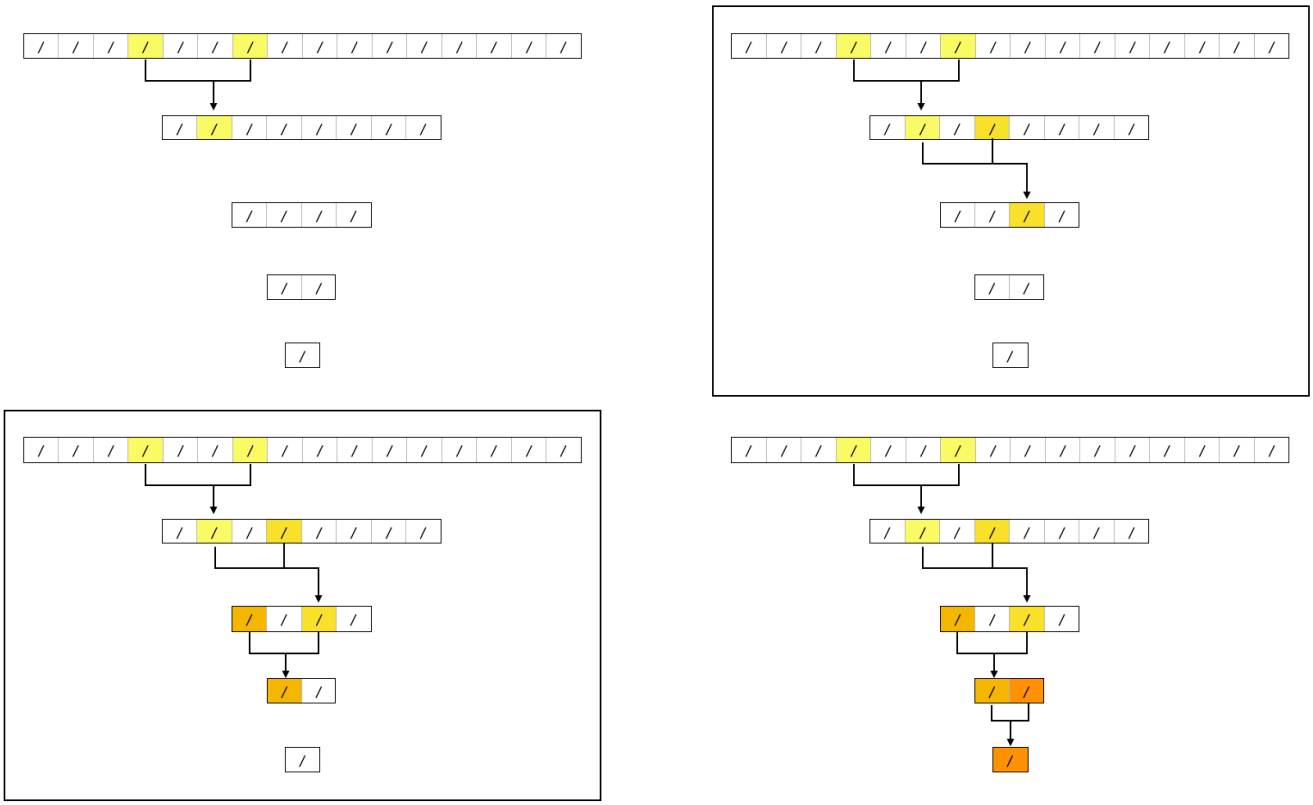
```
def split_poly(p, remove_square=True):
    assert (p.degree()+1) % 2 == 0
    g = (p + p(-x))/2 # <---------- nice trick!
    h = (p - p(-x))//(2 * x) # <--- nice trick!
    # at this point g and h are still around the same degree of p
    # we need to replace x^2 by x for FRI to continue (as we want to halve the degrees!)
    if remove_square:
        g = g.parent(g.list()[::2]) # <-- (using python's `[start:stop:step]` syntax)
        h = h.parent(h.list()[::2])
        assert g.degree() == h.degree() == p.degree() // 2
        assert p(7) == g(7^2) + 7 * h(7^2)
    else:
        assert g.degree() == h.degree() == p.degree() - 1
        assert p(7) == g(7) + 7 * h(7)
    return g, h
```

When instantiating the FRI protocol, like in this specification, the verifier is removed using the **Fiat-Shamir** transformation in order to make the protocol non-interactive.

We can look at the following example to see how a polynomial of degree 7 is reduced to a polynomial of degree 0 in 3 rounds:

```
# p0(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + 6x^5 + 7x^6 + 8x^7
# degree 7 means that we'll get ceil(log2(7)) = 3 rounds (and 4 layers)
p0 = polynomial_ring([1, 2, 3, 4, 5, 6, 7, 8])

# round 1: moves from degree 7 to degree 3
h0, g0 = split_poly(p0)
assert h0.degree() == g0.degree() == 3
zeta0 = 3 # <-------------------- the verifier would pick a random zeta
p1 = h0 + zeta0 * g0 # <--------- the prover would send a commitment of p1
assert p0(zeta0) == p1(zeta0^2) # <- sanity check

# round 2: reduces degree 3 to degree 1
h1, g1 = split_poly(p1)
assert g1.degree() == h1.degree() == 1
zeta1 = 12 # <------------ the verifier would pick a random zeta
p2 = h1 + zeta1 * g1 # <-- the prover would send a commitment of p2
assert p1(zeta1) == p2(zeta1^2)
h2, g2 = split_poly(p2)
assert h2.degree() == g2.degree() == 0

# round 3: reduces degree 1 to degree 0
zeta2 = 3920 # <---------- the verifier would pick a random zeta
p3 = h2 + zeta2 * g2 # <-- the prover could send p3 in the clear
assert p2(zeta2) == p3
assert p3.degree() == 0
```

Queries



In the real FRI protocol, each layer's polynomial would be sent using a hash-based commitment (e.g. a Merkle tree of its evaluations over a large domain). As such, the verifier must ensure that each commitment consistently represents the proper reduction of the previous layer's polynomial. To do that, they "query" commitments of the different polynomials of the different layers at points/evaluations. Let's see how this works.

Given a polynomial $p_0(x) = g_0(x^2) + xh_0(x^2)$ and two of its evaluations at some points v and $-v$, we can see that the verifier can recover the two halves by computing:

- $g_0(v^2) = \frac{p_0(v)+p_0(-v)}{2}$
- $h_0(v^2) = \frac{p_0(v)-p_0(-v)}{2v}$

Then, the verifier can compute the next layer's evaluation at v^2 as:

$$p_1(v^2) = g_0(v^2) + \zeta_0 h_0(v^2)$$

We can see this in our previous example:

```
# first round/reduction
v = 392 # <-------------------------------------- fake sample a point
p0_v, p0_v_neg = p0(v), p0(-v) # <--------------- the 2 queries we need
g0_square = (p0_v + p0_v_neg)/2
h0_square = (p0_v - p0_v_neg)/(2 * v)
```

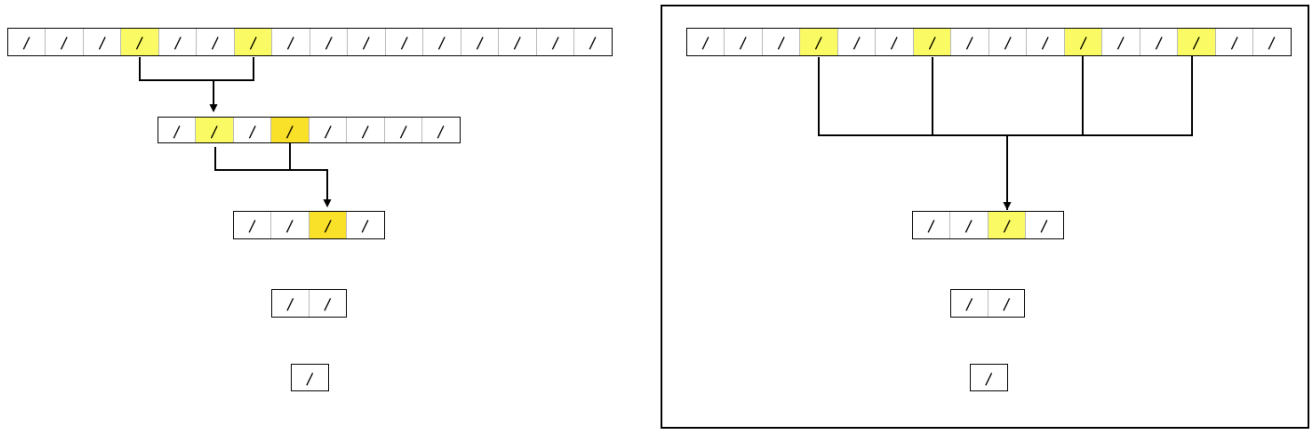
```
assert p0_v == g0_square + v * h0_square # <------ sanity check
```

In practice, to check that the evaluation on the next layer's polynomial is correct, the verifier would "query" the prover's commitment to the polynomial. These queries are different from the **FRI queries** (which enforce consistency between layers of reductions), they are **evaluation queries or commitment queries** and result in practice in the prover providing a Merkle membership proof (also called decommitment in this specification) to the committed polynomial.

As we already have an evaluation of v^2 of the next layer's polynomial p_1, we can simply query the evaluation of $p_1(-v^2)$ to continue the **FRI query** process on the next layer, and so on:

```
p1_v = p1(v^2) # <------------------------------- query on the next layer
assert g0_square + zeta0 * h0_square == p1_v # <--- the correctness check

# second round/reduction
p1_v_neg = p1(-v^2) # <-- the 1 query we need
g1_square = (p1_v + p1_v_neg)/2 # g1(v^4)
h1_square = (p1_v - p1_v_neg)/(2 * v^2) # h1(v^4)
assert p1_v == g1_square + v^2 * h1_square # p1(v^2) = g1(v^4) + v^2 * h1(v^4)
p2_v = p2(v^4) # <-- query the next layer
assert p2(v^4) == g1_square + zeta1 * h1_square # p2(v^4) = g1(v^4) + zeta1 * h1(v^4)

# third round/reduction
p2_v_neg = p2(-v^4) # <-- the 1 query we need
g2_square = (p2_v + p2_v_neg)/2 # g2(v^8)
h2_square = (p2_v - p2_v_neg)/(2 * v^4) # h2(v^8)
assert p2_v == g2_square + v^4 * h2_square # p2(v^4) = g2(v^8) + v^4 * h2(v^8)
assert p3 == g2_square + zeta2 * h2_square # we already received p3 at the end of the protocol
```

Skipping FRI layers



Section 3.11.1 "Skipping FRI Layers" of the ethSTARK paper describes an optimization which skips some of the layers/rounds. The intuition is the following: if we removed the first round commitment (to the polynomial p_1), then

the verifier would not be able to:

- query $\mathrm{p}_1(\mathrm{v}^2)$ to verify that layer
- query $\mathrm{p1}(-\mathrm{v}^2)$ to continue the protocol and get $\mathrm{g}_1, \mathrm{h}_1$

The first point is fine, as there's nothing to check the correctness of. To address the second point, we can use the same technique we use to compute $\mathrm{p1}(\mathrm{v}^2)$. Remember, we needed $\mathrm{p}_0(\mathrm{v})$ and $\mathrm{p}_0(-\mathrm{v})$ to compute $\mathrm{g}_0(\mathrm{v}^2)$ and $\mathrm{h}_0(\mathrm{v}^2)$. But to compute $\mathrm{g}_0(-\mathrm{v}^2)$ and $\mathrm{h}_0(-\mathrm{v}^2)$, we need the quadratic residues of $-\mathrm{v}^2$, that is w, such that $\mathrm{w}^2 = -\mathrm{v}^2$, so that we can compute $\mathrm{g}_0(-\mathrm{v}^2)$ and $\mathrm{h}_0(-\mathrm{v}^2)$ from $\mathrm{p}_0(\mathrm{w})$ and $\mathrm{p}_0(-\mathrm{w})$.

We can easily compute them by using τ (`tau`), the generator of the subgroup of order 4:

```
tau = find_gen2(2)
assert tau.multiplicative_order() == 4

# so now we can compute the two roots of -v^2 as
assert (tau * v)^2 == -v^2
assert (tau^3 * v)^2 == -v^2

# and when we query p2(v^4) we can verify that it is correct
# if given the evaluations of p0 at v, -v, tau*v, tau^3*v
p0_tau_v = p0(tau * v)
p0_tau3_v = p0(tau^3 * v)
p1_v_square = (p0_v + p0_v_neg)/2 + zeta0 * (p0_v - p0_v_neg)/(2 * v)
p1_neg_v_square = (p0_tau_v + p0_tau3_v)/2 + zeta0 * (p0_tau_v - p0_tau3_v)/(2 * tau * v)
assert p2(v^4) == (p1_v_square + p1_neg_v_square)/2 + zeta1 * (p1_v_square - p1_neg_v_square)/(2
* v^2)
```

There is no point producing a new challenge `zeta1` as nothing more was observed from the verifier's point of view during the skipped round. As such, FRI implementations will usually use `zeta0^2` as "folding factor" (and so on if more foldings occur).

Last Layer Optimization

Section 3.11.2 "FRI Last Layer" of the ethSTARK paper describes an optimization which stops at an earlier round. We show this here by removing the last round.

At the end of the second round we imagine that the verifier receives the coefficients of p_2 (h_2 and g_2) directly:

```
p2_v = h2 + v^4 * g2 # they can then compute p2(v^4) directly
assert g1_square + zeta1 * h1_square == p2_v # and then check correctness
```

FRI-PCS

Given a polynomial f and an evaluation point a, a prover who wants to prove that $f(a) = b$ can prove the related statement for some quotient polynomial q of degree $\deg(f) - 1$:

$$\frac{f(x) - b}{x - a} = q(x)$$

(This is because if $f(a) = b$ then a should be a root of $f(x) - b$ and thus the polynomial can be factored in this way.)

Specifically, FRI-PCS proves that they can produce such a (commitment to a) polynomial q.

Aggregating Multiple FRI Proofs

To prove that two polynomials a and b exist and are of degree at most d, a prover simply shows using FRI that a random linear combination of a and b exists and is of degree at most d.

Note that if the FRI check might need to take into account the different degree checks that are being aggregated. For example, if the polynomial a should be of degree at most d but the polynomial should be of degree at most $d + 3$ then a degree correction needs to happen. We refer to the **ethSTARK paper** for more details as this is out of scope for this specification. (As used in the STARK protocol targeted by this specification, it is enough to show that the polynomials are of low degree.)

Notable Differences With Vanilla FRI

Besides obvious missing implementation details from the description above, the protocol is pretty much instantiated as is, except for a few changes to the folding and querying process.

As explained above, in the "vanilla FRI" protocol the verifier gets evaluations of $p_0(v)$ and $p_0(-v)$ and computes the next layer's evaluation at v^2 as

$$p_{i+1}(v^2) = \frac{p_i(v) + p_i(-v)}{2} + \zeta_i \frac{p_i(v) - p_i(-v)}{2v}$$

which is equivalent to

$$p_{i+1}(v^2) = g_i(v^2) + \zeta_i h_i(v^2)$$

where

$$p_i(x) = g_i(x^2) + x h_i(x^2)$$

The first difference in this specification is that, assuming no skipped layers, the folded polynomial is multiplied by 2:

$$p_{i+1}(x) = 2(g_i(x) + \zeta_i \cdot h_i(x))$$

This means that the verifier has to modify their queries slightly by not dividing by 2:

$$p_{i+1}(v^2) = p_i(v) + p_i(-v) + \zeta_i \cdot \frac{p_i(v) - p_i(-v)}{v}$$

The second difference is that while the evaluations of the first layer p_0 happen in a coset called evaluation domain, further evaluations happen in the original (blown up) trace domain (which is avoided for the first polynomial as it might lead to divisions by zero with the polynomials used in the Starknet STARK protocol). To do this, the prover defines the first reduced polynomial as:

$$p_1(x) = 2(g_0(9x^2) + \zeta_0 \cdot 3 \cdot h_0(9x^2))$$

Notice that the prover has also multiplied the right term with 3. This is a minor change that helps with how the verifier code is structured.

This means that the verifier computes the queries on $p_1(x)$ at points on the original subgroup. So the queries of the first layer are produced using $v' = v/3$ (assuming no skipped layers).

$$p_1((v'^2) = p_0(v) + p_0(-v) + \zeta_0 \cdot \frac{p_0(v) - p_0(-v)}{v'}$$

We assume no skipped layers, which is always the case in this specification for the first layer's reduction.

After that, everything happens as normal (except that now the prover uses the original blown-up trace domain instead of a coset to evaluate and commit to the layer polynomials).

Note that these changes can easily be generalized to work when layers are skipped.

External Dependencies

In this section we list all the dependencies and interfaces this standard relies on.

Hash Functions

We rely on two type of hash functions:

- A verifier-friendly hash. Specifically, **Poseidon**.
- A standard hash function. Specifically, **Keccak**.

Channel

See the **Channel** specification for details.

Verifying the first FRI layer

As part of the protocol, the prover must provide a number of evaluations of the first layer polynomial p_0 (based on the FRI queries that the verifier generates in the **query phase** of the protocol).

We abstract this here as an oracle that magically provides evaluations. It is the responsibility of the user of this protocol to ensure that the evaluations are correct (which most likely include verifying a number of decommitments). See the **Starknet STARK verifier specification** for a concrete usage example.

For example, the STARK protocol computes evaluations of $\mathrm{p}_0(\mathrm{v})$ (but not $\mathrm{p}_0(\mathrm{v})$) using decommitments of trace column polynomials and composition column polynomials at the same path corresponding to the evaluation point v.

Constants

We use the following constants throughout the protocol.

Protocol constants

`STARKNET_PRIME = 3618502788666131213697322783095070105623107215331596699973092056135872020481`. The Starknet prime ($2^{251} + 17 \cdot 2^{192} + 1$).

`FIELD_GENERATOR = 3`. The generator for the main multiplicative subgroup of the Starknet field. This is also used as the coset factor to produce the coset used in the first layer's evaluation.

FRI constants

`MAX_LAST_LAYER_LOG_DEGREE_BOUND = 15`. The maximum degree of the last layer polynomial (in log2).

`MAX_FRI_LAYERS = 15`. The maximum number of layers in the FRI protocol.

`MAX_FRI_STEP = 4`. The maximum number of layers that can be involved in a reduction in FRI (see the overview for more details). This essentially means that each reduction (except for the first as we specify later) can skip 0 to 3 layers.

This means that the standard can be implemented to test that committed polynomials exist and are of degree at most $2^{15+15} = 2^{30}$.

Step Generators And Inverses

As explained in the overview, skipped layers must involve the use of elements of the subgroups of order 2^{i} for i the number of layers included in a step (from 1 to 4 as specified previously).

As different generators can generate the same subgroups, we have to define the generators that are expected. Instead, we define the inverse of the generators of groups of different orders (as it more closely matches the code):

- `const OMEGA_16: felt252 = 0x5c3ed0c6f6ac6dd647c9ba3e4721c1eb14011ea3d174c52d7981c5b8145aa75;`
- `const OMEGA_8: felt252 = 0x446ed3ce295dda2b5ea677394813e6eab8bfbc55397aacac8e6df6f4bc9ca34;`
- `const OMEGA_4: felt252 = 0x1dafdc6d65d66b5accedf99bcd607383ad971a9537cdf25d59e99d90becc81e;`
- `const OMEGA_2: felt252 = -1`

So here, for example, `OMEGA_8` is $1/\omega_8$ where ω_8 is the generator of the subgroup of order 8 that we later use in the Verify A Layer's Query section.

Configuration

General configuration

The FRI protocol is globally parameterized according to the following variables. For a real-world example, check the Starknet STARK verifier specification.

`n_verifier_friendly_commitment_layers`. The number of layers (starting from the bottom) that make use of the circuit-friendly hash.

`proof_of_work_bits`. The number of bits required for the proof of work. This value should be between 20 and 50.

FRI configuration

A FRI configuration contains the following fields:

`log_input_size`. The size of the input layer to FRI, specifically the log number of evaluations committed (this should match the log of the evaluation domain size).

`n_layers`. The number of layers or folding that will occur as part of the FRI proof. This value must be within the range `[2, MAX_FRI_LAYERS]` (see constants).

`inner_layers`. An array of `TableCommitmentConfig` where each configuration represents what is expected of each commitment sent as part of the FRI proof. Refer to the Table Commitments section of the Starknet Merkle Tree Polynomial Commitments specification.

`fri_step_sizes`. The number of layers to skip for each folding/reduction of the protocol. The first step must always be zero, as no layer is skipped during the first reduction. Each step should be within the range `[1, MAX_FRI_STEP]`. For each step, the corresponding layer `inner_layers[i-1]` should have enough columns to support the reduction: `n_columns = 2^fri_step`.

`log_last_layer_degree_bound`. The degree of the last layer's polynomial. As it is sent in clear as part of the FRI protocol, this value represents the (log) number of coefficients (minus 1) that the proof will contain. It must be less or equal to `MAX_LAST_LAYER_LOG_DEGREE_BOUND` (see constants).

In addition, the following validations should be performed on passed configurations:

- for every `fri_step_sizes[i]` check:
  - that the previous layer table commitment configuration `inner_Layers[i-1]` has
    - a valid configuration, which can be verified using the expected log input size and the `n_verifier_friendly_commitment_layers`
      - expected log input size should be the input size minus all the step sizes so far
- the `log_expected_input_degree + log_n_cosets == log_input_size`
  - where `log_expected_input_degree = sum_of_step_sizes + log_last_layer_degree_bound`

Domains and Commitments

There are three types of domains:

The **trace domain**, this is the domain chosen to evaluate the execution trace polynomials. It is typically the smallest subgroup of order 2^{n_t} for some n_t, such that it can include all the constraints of an AIR constraint system. A generator for the trace domain can be found as $\omega_t = 3^{(p-1)/n_t}$ (since $\omega_t^{n_t} = 1$)

The **blown-up trace domain**, which is chosen as a subgroup of a power of two 2^{n_e} that encompasses the trace domain (i.e. $e \geq t$). The "blown up factor" typically dictates how much larger the evaluation domain is as a multiple. A generator for the blown-up trace domain can be found as $\omega_e = 3^{(p-1)/n_e}$.

The **evaluation domain**, This is a coset of the blown-up domain, computed using the generator of the main group: $\{3 \cdot \omega_e^i | i \in [[0, n_e]]\}$.

Commitments are created using table commitments as described in the **Table Commitments section of the Merkle Tree Polynomial Commitments specification**.

For the first layer polynomial, the evaluations being committed are in a coset called the evaluation domain.

For all other polynomials, commitments are made up of evaluations in the blown-up trace domain (following the correction outlined in the **Notable Differences With Vanilla FRI** section).

The reason for choosing a coset is two-folds. First, in ZK protocols you want to avoid decommitting actual witness values by querying points in the trace domain. Choosing another domain helps but is not sufficient. As this specification does not provide a ZK protocol. The second reason is the one that is interesting to us: it is an optimization reason. As the prover needs to compute the composition polynomial, they can do this in the monomial basis (using vectors of coefficient of the polynomials) but it is expensive. For this reason, they usually operate on polynomials using the lagrange basis (using vectors of evaluations of the polynomials). As such, calculating the composition polynomial leads to divisions by zero if the trace domain is used. The prover could in theory use any other domains, but they decide to use the same domain that they use to commit (the evaluation domain) to avoid having to interpolate and re-evaluate in the domain to commit (which would involve two FFTs).

Protocol

A FRI proof looks like the following:

```
struct FriUnsentCommitment {
    // Array of size n_layers - 1 containing unsent table commitments for each inner layer.
    inner_layers: Span<felt252>,
    // Array of size 2**log_last_layer_degree_bound containing coefficients for the last layer
    // polynomial.
    last_layer_coefficients: Span<felt252>,
}
```

The FRI protocol is split into two phases:

1. Commit phase
2. Query phase

We go through each of the phases in the next two subsections.

Commit Phase

The commit phase processes the `FriUnsentCommitment` object in the following way:

1. Enforce that the first layer has a step size of 0 (`cfg.fri_step_sizes[0] == 0`). (Note that this is mostly to make sure that the prover is following the protocol correctly, as the second layer is never skipped in this standard.)
2. Go through each commitment in order in the `inner_layers` field and perform the following:
    1. Absorb the commitment using the **channel**.
    2. Produce a random challenge.
3. Absorb the `last_layer_coefficients` with the channel.
4. Check that the last layer's degree is correct (according to the configuration `log_last_layer_degree_bound`, see the **Configuration section**): `2^cfg.log_last_layer_degree_bound == len(unsent_commitment.last_layer_coefficients)`.

5. return all the random challenges.

Query Phase

FRI queries are generated once, and then refined through each reduction of the FRI protocol. The number of queries that is pseudo-randomly generated is based on **configuration**.

Each FRI query is composed of the following fields:

- `index`: the index of the query in the layer's evaluations. Note that this value needs to be shifted before being used as a path in a Merkle tree commitment.
- `y_value`: the evaluation of the layer's polynomial at the queried point.
- `x_inv_value`: the inverse of the point at which the layer's polynomial is evaluated. This value is derived from the `index` as explained in the next subsection.

```
struct FriLayerQuery {
    index: felt252,
    y_value: felt252,
    x_inv_value: felt252,
}
```

That is, we should have for each FRI query for the layer $\mathrm{i}+1$ the following identity:

$$\mathrm{p_{i+1}}(1/\mathrm{x\_inv\_value}) = \mathrm{y\_value}$$

Or in terms of commitment, that the decommitment at path `index` is `y_value`.

This is not exactly correct. The Commitment section explains that `index` points to a point, whereas we need to point to the path in the Merkle tree commitment that gathers its associated points. In addition, `y_value` only gives one evaluation, so the prover will need to witness associated evaluations surrounding the `y_value` as well (see Table Commitment section).

See the **Converting A Query to a Path section of the Merkle tree specification** for details.

Generating The First Queries

The generation of each FRI query goes through the same process:

- Sample a random challenge from the **channel**.
- Truncate the challenge to obtain the lower 128-bit chunk.
- Reduce it modulo the size of the evaluation domain.

Finally, when all FRI queries have been generated, they are sorted in ascending order.

This gives you a value that is related to the path to query in a Merkle tree commitment, and can be used to derive the actual evaluation point at which the polynomial is evaluated. Commitments should reveal not just one evaluation, but correlated evaluations in order to help the protocol move forward. For example, if a query is generated for the evaluation point v, then the commitment will reveal the evaluation of a polynomial at v but also at $-\mathrm{v}$ and potentially more points (depending on the number of layers skipped).

A query q (a value within $[0, 2^{\mathrm{n_e}}]$ for $\mathrm{n_e}$ the log-size of the evaluation domain) can be converted to an evaluation point in the following way. First, compute the bit-reversed exponent:

$$\mathrm{q}' = \mathrm{bit\_reverse}(\mathrm{q} \cdot 2^{64-\mathrm{n_e}})$$

Then compute the element of the evaluation domain in the coset (with ω_{e} the generator of the evaluation domain):

$$3 \cdot \omega_{\mathrm{e}}^{\mathrm{q}'}$$

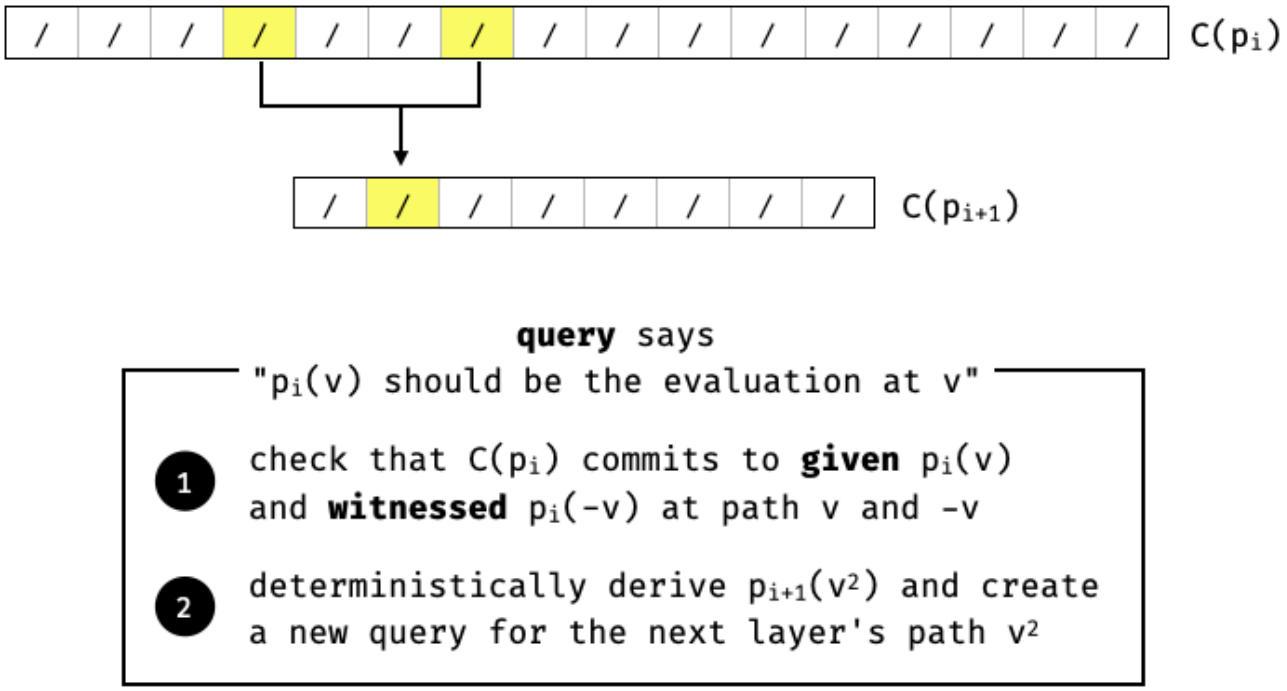
Finally, the expected evaluation can be computed using the API defined in the **Verifying the first FRI layer** section.

Verify A Layer's Query

Besides the last layer, each layer verification of a query happens by:

1. verifying the query on the current layer. This is done by effectively decommitting a layer's query following the **Merkle Tree Polynomial Commitment** specification.
2. computing the next query as explained below.

We illustrate this in the following diagram, pretending that associated evaluations are not grouped under the same path in the Merkle tree commitment (although in practice they are).



This means that when used in the STARK protocol, for example, the first layer represents the same polynomial as the aggregation of a number of FRI checks, and associated evaluations (e.g. $-v$ given v) are witnessed. This is akin to a reduction in the FRI protocol (except that the linear combination includes many more terms and scalars).

To verify the last layer's query, as the last layer polynomial is received in clear, simply evaluate it at the queried point `1/fri_layer_query.x_inv_value` and check that it matches the expected evaluation `fri_layer_query.y_value`.

Each query verification (except on the last layer) will produce queries for the next layer, which will expect specific evaluations.

The next queries are derived as:

- index: `index / coset_size`
- point: `point^coset_size`
- value: see FRI formula below

where coset_size is 2, 4, 8, or 16 depending on the layer (but always 2 for the first layer).

Queries between layers verify that the next layer p_{i+j} is computed correctly based on the current layer p_i. The next layer is either the direct next layer p_{i+1} or a layer further away if the configuration allows layers to be skipped. Specifically, each reduction is allowed to skip 0, 1, 2, or 3 layers (see the `MAX_FRI_STEP` constant).

The FRI formula with no skipping is:

- given a layer evaluations at $\pm v$, a query without skipping layers work this way:
- we can compute the next layer's *expected* evaluation at v^2 by computing $p_{i+1}(v^2) = \frac{p_i(v)+p_i(-v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-v)}{2v}$
- we can then ask the prover to open the next layer's polynomial at that point and verify that it matches

The FRI formula with 1 layer skipped with ω_4 the generator of the 4-th roots of unity (such that $\omega_4^2 = -1$):

- $p_{i+1}(v^2) = \frac{p_i(v)+p_i(-v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-v)}{2v}$
- $p_{i+1}(-v^2) = \frac{p_i(\omega_4 v)+p_i(-\omega_4 v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-\omega_4 v)}{2 \cdot \omega_4 \cdot v}$
- $p_{i+2}(v^4) = \frac{p_{i+1}(v^2)+p_{i+1}(-v^2)}{2} + \zeta_i^2 \cdot \frac{p_i(v^2)-p_i(-v^2)}{2 \cdot v^2}$

As you can see, this requires 4 evaluations of p_ at v, $-v$, $\omega_4 v$, $-\omega_4 v$.

The FRI formula with 2 layers skipped with ω_8 the generator of the 8-th roots of unity (such that $\omega_8^2 = \omega_4$ and $\omega_8^4 = -1$):

- $p_{i+1}(v^2) = \frac{p_i(v)+p_i(-v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-v)}{2v}$
- $p_{i+1}(-v^2) = \frac{p_i(\omega_4 v)+p_i(-\omega_4 v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-\omega_4 v)}{2 \cdot \omega_4 \cdot v}$
- $p_{i+1}(\omega_4 v^2) = \frac{p_i(\omega_8 v)+p_i(-\omega_8 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_8 v)-p_i(-\omega_8 v)}{2 \omega_8 v}$
- $p_{i+1}(-\omega_4 v^2) = \frac{p_i(\omega_8^3 v)+p_i(-\omega_8^3 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_8^3 v)-p_i(-\omega_8^3 v)}{2 \cdot \omega_8^3 \cdot v}$
- $p_{i+2}(v^4) = \frac{p_{i+1}(v^2)+p_{i+1}(-v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(v^2)-p_{i+1}(-v^2)}{2 \cdot v^2}$
- $p_{i+2}(-v^4) = \frac{p_{i+1}(\omega_4 v^2)+p_{i+1}(-\omega_4 v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(\omega_4 v^2)-p_{i+1}(-\omega_4 v^2)}{2 \cdot \omega_4 v^2}$
- $p_{i+3}(v^8) = \frac{p_{i+2}(v^4)+p_{i+2}(-v^4)}{2} + \zeta_i^4 \cdot \frac{p_{i+2}(v^4)-p_{i+2}(-v^4)}{2 \cdot v^4}$

As you can see, this requires 8 evaluations of p_ at v, $-v$, $\omega_4 v$, $-\omega_4 v$, $\omega_8 v$, $-\omega_8 v$, $\omega_8^3 v$, $-\omega_8^3 v$.

The FRI formula with 3 layers skipped with ω_{16} the generator of the 16-th roots of unity (such that $\omega_{16}^2 = \omega_8$, $\omega_{16}^4 = \omega_4$, and $\omega_{16}^8 = -1$):

- $p_{i+1}(v^2) = \frac{p_i(v)+p_i(-v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-v)}{2v}$
- $p_{i+1}(-v^2) = \frac{p_i(\omega_4 v)+p_i(-\omega_4 v)}{2} + \zeta_i \cdot \frac{p_i(v)-p_i(-\omega_4 v)}{2 \cdot \omega_4 \cdot v}$
- $p_{i+1}(\omega_4 v^2) = \frac{p_i(\omega_8 v)+p_i(-\omega_8 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_8 v)-p_i(-\omega_8 v)}{2 \omega_8 v}$
- $p_{i+1}(-\omega_4 v^2) = \frac{p_i(\omega_8^3 v)+p_i(-\omega_8^3 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_8^3 v)-p_i(-\omega_8^3 v)}{2 \cdot \omega_8^3 \cdot v}$
- $p_{i+1}(\omega_8 v^2) = \frac{p_i(\omega_1 6 v)+p_i(-\omega_1 6 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_1 6 v)-p_i(-\omega_1 6 v)}{2 \omega_1 6 v}$
- $p_{i+1}(-\omega_8 v^2) = \frac{p_i(\omega_1 6^5 v)+p_i(-\omega_1 6^5 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_1 6^5 v)-p_i(-\omega_1 6^5 v)}{2 \omega_1 6^5 v}$
- $p_{i+1}(\omega_8^3 v^2) = \frac{p_i(\omega_1 6^3 v)+p_i(-\omega_1 6^3 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_1 6^3 v)-p_i(-\omega_1 6^3 v)}{2 \omega_1 6^3 v}$
- $p_{i+1}(-\omega_8^3 v^2) = \frac{p_i(\omega_1 6^7 v)+p_i(-\omega_1 6^7 v)}{2} + \zeta_i \cdot \frac{p_i(\omega_1 6^7 v)-p_i(-\omega_1 6^7 v)}{2 \omega_1 6^7 v}$
- $p_{i+2}(v^4) = \frac{p_{i+1}(v^2)+p_{i+1}(-v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(v^2)-p_{i+1}(-v^2)}{2 \cdot v^2}$
- $p_{i+2}(-v^4) = \frac{p_{i+1}(\omega_4 v^2)+p_{i+1}(-\omega_4 v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(\omega_4 v^2)-p_{i+1}(-\omega_4 v^2)}{2 \cdot \omega_4 v^2}$

- $p_{i+2}(\omega_4 v^4) = \frac{p_{i+1}(\omega_8 v^2)+p_{i+1}(-\omega_8 v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(\omega_8 v^2)-p_{i+1}(-\omega_8 v^2)}{2 \cdot \omega_8 \cdot v^2}$
- $p_{i+2}(-\omega_4 v^4) = \frac{p_{i+1}(\omega_8^3 v^2)+p_{i+1}(-\omega_8^3 v^2)}{2} + \zeta_i^2 \cdot \frac{p_{i+1}(\omega_8^3 v^2)-p_{i+1}(-\omega_8^3 v^2)}{2 \cdot \omega_8^3 v^2}$
- $p_{i+3}(v^8) = \frac{p_{i+2}(v^4)+p_{i+2}(-v^4)}{2} + \zeta_i^4 \cdot \frac{p_{i+2}(v^4)-p_{i+2}(-v^4)}{2 \cdot v^4}$
- $p_{i+4}(v^16) = \frac{p_{i+3}(v^8)+p_{i+3}(-v^8)}{2} + \zeta_i^8 \cdot \frac{p_{i+3}(v^8)-p_{i+3}(-v^8)}{2 \cdot v^8}$

As you can see, this requires 16 evaluations of p_ at v, $-v$, $\omega_4 v$, $-\omega_4 v$, $\omega_8 v$, $-\omega_8 v$, $\omega_8^3 v$, $-\omega_8^3 v$, $\omega_1 6 v$, $-\omega_1 6 v$, $\omega_1 6^3 v$, $-\omega_1 6^3 v$, $\omega_1 6^5 v$, $-\omega_1 6^5 v$, $\omega_7 v$, $-\omega_7 v$.

Proof of work

In order to increase the cost of attacks on the protocol, a proof of work is added at the end of the commitment phase.

Given a 32-bit hash `digest` and a difficulty target of `proof_of_work_bits`, verify the 64-bit proof of work `nonce` by doing the following:

1. Produce a `init_hash = hash_n_bytes(0x0123456789abcded || digest || proof_of_work_bits)`
2. Produce a `hash = hash_n_bytes(init_hash || nonce)`
3. Enforce that the 128-bit high bits of `hash` start with `proof_of_work_bits` zeros (where `proof_of_work_bits` is enforced to be between 20 and 50 as discussed in the **General Configuration section**).

Full Protocol

The FRI flow is split into four main functions. The only reason for doing this is that verification of FRI proofs can be computationally intensive, and users of this specification might want to split the verification of a FRI proof in multiple calls.

The four main functions are:

1. `fri_commit`, which returns the commitment to every layers of the FRI proof.
2. `fri_verify_initial`, which returns the initial set of queries to verify the first reduction (Which is special as explained in the **Notable Differences With Vanilla FRI** section).
3. `fri_verify_step`, which takes a set of queries and returns another set of queries.
4. `fri_verify_final`, which takes the final set of queries and the last layer coefficients and returns the final result.

To retain context, functions pass around two objects:

```
struct FriVerificationStateConstant {
    // the number of layers in the FRI proof (including skipped layers)
    n_layers: u32,
    // commitments to each layer (excluding the first, last, and any skipped layers)
    commitment: Span<TableCommitment>,
```

```
    // verifier challenges used to produce each (non-skipped) layer polynomial (except the
first)
    eval_points: Span<felt252>,
    // the number of layers to skip for each reduction
    step_sizes: Span<felt252>,
    // the hash of the polynomial of the last layer
    last_layer_coefficients_hash: felt252,
}
struct FriVerificationStateVariable {
    // a counter representing the current layer being verified
    iter: u32,
    // the FRI queries for each (non-skipped) layer
    queries: Span<FriLayerQuery>,
}
```

It is the responsibility of the wrapping protocol to ensure that these three functions are called sequentially, enough times, and with inputs that match the output of previous calls.

We give more detail to each function below.

`fri_commit(channel)`.

1. Take a channel with a prologue (See the **Channel** section). A prologue contains any context relevant to this proof.
2. Produce the FRI commits according to the **Commit Phase** section.
3. Produce the proof of work according to the **Proof of Work** section.
4. Generate `n_queries` queries in the `eval_domain_size` according to the **Generating Queries** section.
5. Convert the queries to evaluation points following the **Converting A Query To An Evaluation Point** section, producing `points`.
6. Evaluate the first layer at the queried `points` using the external dependency (see **External Dependencies** section), producing `values`.
7. Produce the fri_decommitment as `FriDecommitment { values, points }`.

`fri_verify_initial(queries, fri_commitment, decommitment)`. Takes the FRI queries, the FRI commitments (each layer's committed polynomial), as well as the evaluation points and their associated evaluations of the first layer (in `decommitment`).

1. Enforce that for each query there is a matching derived evaluation point and evaluation at that point on the first layer contained in the given `decommitment`.
2. Enforce that last layer has the right number of coefficients as expected by the FRI configuration (see the **FRI Configuration** section).
3. Compute the first layer of queries as `FriLayerQuery { index, y_value, x_inv_value: 3 / x_value }` for each `x_value` and `y_value` given in the `decommitment`. (This is a correction that will help achieve the differences in subsequent layers outlined in **Notable Differences With Vanilla FRI**).
4. Initialize and return the two state objects

```
(
    FriVerificationStateConstant {
        n_layers: config.n_layers - 1,
        commitment: fri_commitment.inner_layers, // the commitments
        eval_points: fri_commitment.eval_points, // the challenges
        step_sizes: config.fri_step_sizes[1:], // the number of reduction at each steps
        last_layer_coefficients_hash: hash_array(last_layer_coefficients),
    },
    FriVerificationStateVariable { iter: 0, queries: fri_queries } // the initial queries
)
```

`fri_verify_step(stateConstant, stateVariable, witness, settings)`.

1. Enforce that `stateVariable.iter <= stateConstant.n_layers`.
2. Verify the queried layer and compute the next query following the **Verify A Layer's Query** section.
3. Increment the `iter` counter.
4. Return the next queries and the counter.

`fri_verify_final(stateConstant, stateVariable, last_layer_coefficients)`.

1. Enforce that the counter has reached the last layer from the constants (`iter == n_layers`).
2. Enforce that the `last_layer_coefficient` matches the hash contained in the state.
3. Manually evaluate the last layer's polynomial at every query and check that it matches the expected evaluations.

```
fn fri_verify_final(
    stateConstant: FriVerificationStateConstant,
    stateVariable: FriVerificationStateVariable,
    last_layer_coefficients: Span<felt252>,
) -> (FriVerificationStateConstant, FriVerificationStateVariable) {
    assert(stateVariable.iter == stateConstant.n_layers, 'Fri final called at wrong time');
    assert(
        hash_array(last_layer_coefficients) == stateConstant.last_layer_coefficients_hash,
        'Invalid last_layer_coefficients'
    );

    verify_last_layer(stateVariable.queries, last_layer_coefficients);

    (
        stateConstant,
        FriVerificationStateVariable { iter: stateVariable.iter + 1, queries: array![].span(), }
    )
}
```

Test Vectors

Refer to the reference implementation for test vectors.

Security Considerations

The current way to compute the bit security is to compute the following formula:

```
n_queries * log_n_cosets + proof_of_work_bits
```

Where:

- `n_queries` is the number of queries generates
- `log_n_cosets` is the log2 of the blow-up factor
- `proof_of_work_bits` is the number of bits required for the proof of work (see the **Proof of Work section**).

Starknet STARK Verifier

For a more up-to-date specification check https://zksecurity.github.io/RFCs/.

In this document we specify the STARK verifier used in Starknet.

Overview

The protocol specified here is not "zero-knowledge". It is purely aimed at providing succinctness. That is, it is useful to delegate computation.

Note that the protocol implemented closely resembles the high-level explanations of the **ethSTARK paper**, as such we refer to it in places.

This protocol is instantiated in several places to our knowledge:

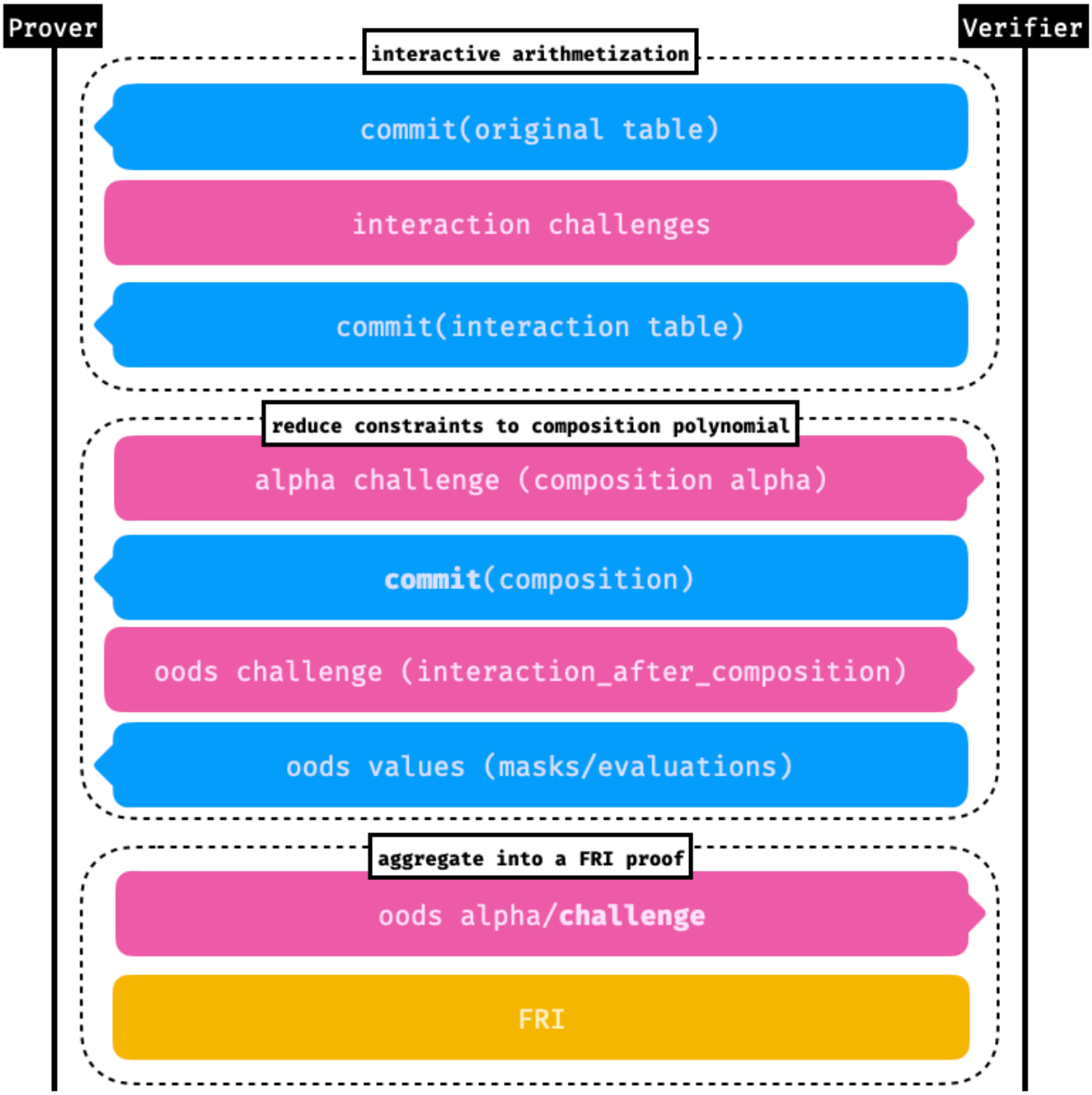
- **C++ implementation**
- **Solidity implementation**
- **Cairo0 implementation**
- **Cairo1 implementation**

In this section we give a brief overview of the Starknet STARK protocol. While the protocol used is designed to verify Cairo programs, we provide an agnostic specification. The instantiation of this protocol with Cairo should be the object of a different specification.

Before we delve into the details, let's look at the protocol from a high-level protocol diagram point of view. The Starknet STARK protocol is divided in three main phases:

1. Construction of an interactive arithmetization. In this phase the prover commits to different parts of the execution trace it wants to prove, using random challenges in-between.
2. Aggregation of constraints into a composition polynomial. In this phase the prover commits to a composition polynomial that, if checked by FRI, proves that the execution trace satisfies the constraints. It also produces evaluations of commitments at a random point so that the verifier can check that the composition polynomial is well-formed.
3. Aggregation of FRI proofs and FRI protocol. The composition polynomial FRI check as well as evaluation proofs (using FRI-PCS) of all the sent evaluations are aggregated into a single FRI check. The FRI protocol is then run to verify the aggregated FRI proof. See the **Starknet FRI Verifier specification** for more details.

We illustrate the flow in the following diagram:



In the next sections we review the different phases.

Interactive AIR Arithmetization

But first, we quickly remind the reader that the Starknet STARK protocol allows a prover to convince a verifier that an AIR (Algebraic Intermediate Representation) arithmetization is satisfied by their witness. This is generally augmented to also include a public input, usually via a public memory extension.

AIR is essentially two things:

1. an indexed table representing the execution trace of a run, where columns can be seen as registers and the rows the values they take as one steps through a program. The table takes values when a prover tries to prove an execution.
2. a list of fixed constraints that are agreed on.

The indexing of the table is chosen as the elements of the smallest subgroup of power 2 that can index the table.

Furthermore, the columns of a table can be grouped, which allows the prover to fill the table group by group, using challenges from the verifier in-between. This is useful in order to perform an interactive arithmetization where parts of the encoded circuit need verifier randomness to be computed.

We give the example of two "original" columns and one "interaction" column, indexed using the multiplicative subgroup of the 16-th roots of unity:

	original		interactive
idx	col0	col1	col2
$w^0=1$	1	2	_
w^1	3	2	_
w^2	5	1	_
w^3	6	6	_
w^4	12	0	_
w^5	13	13	_
w^6	26	3	_
w^7	29	0	_
w^8	29	-1	_
w^9	28	100	_
w^{10}	128	128	_
w^{11}	256	3	_
w^{12}	259	24	_
w^{13}	283	9	_
w^{14}	292	1049	_
w^{15}	1338	0	_

Here one constraint could be to enforce that `col0[i] + col1[i] - col0[i+1] = 0` on every row `i` except the last one.

As the columns of the table are later interpolated over the index domain, such constraints are usually described and applied as polynomials. So the previous example constraint would look like the following polynomial:

$$\frac{\mathrm{col}_0(x)+\mathrm{col}_1(x)-\mathrm{col}_0(x\cdot w)}{D_0(x)}$$

where the domain polynomial D_0 can be efficiently computed as $\frac{x^{16}-1}{w^{15}-1}$.

The first phase of the Starknet STARK protocol is to iteratively construct the trace tables (what we previously called interactive arithmetization). The prover sends commitments to parts of the table, and receives verifier challenges in

between.

In the instantiation of the Starknet STARK protocol, there are only two execution trace tables: the original trace table and the interaction trace table, the verifier challenges received in between are called the interaction challenges. Different Cairo layouts will give place to different trace tables and interaction challenges.

Composition Polynomial

The role of the verifier is now to verify constraints of the form of polynomials on the trace column polynomials, applied on a domain (a list of all the indexes on which the constraint applies).

As with our example above, we can imagine a list of constraints $\mathrm{C_i(x)}$ that need to vanish on a list of associated domains described by their domain polynomials $\mathrm{D_i(x)}$.

By definition, this can be reduced to checking that you can write each $\mathrm{C_i}$ as $\mathrm{C_i(x)} = \mathrm{D_i(x)} \cdot \mathrm{q(x)}$ for some quotient polynomial $\mathrm{q(x)}$ of degree $\mathrm{deg(C_i)} - \mathrm{deg(D_i)}$.

While protocols based on polynomial commitments like KZG would commit to the quotient polynomial and then prove the relation $\mathrm{C_i(x)} = \mathrm{D_i(x)} \cdot \mathrm{q(x)}$ at a random point (using Schwartz-Zippel), the Starknet STARK protocol uses a different approach: it uses a FRI check to prove that the commitment to the evaluations of $\mathrm{q(x)} = \frac{\mathrm{C_i(x)}}{\mathrm{D_i(x)}}$ correctly represents a polynomial of low degree.

As such, the role of the verifier is to verify that all the quotient polynomials associated with all the constraints exist and are of low-degree.

As we want to avoid having to go through many FRI checks, the verifier sends a challenge α which the prover can use to aggregate all of the constraint quotient polynomials into a **composition polynomial** $\mathrm{h(x)} := \sum_{\mathrm{i}=0} \frac{\mathrm{C_i(x)}}{\mathrm{D_i(x)}} \cdot \alpha^{\mathrm{i}}$.

This composition polynomial is quite big, so the prover provides a commitment to chunks or columns of the composition polynomials, interpreting h as $\mathrm{h(x)} = \sum_{\mathrm{i}} \mathrm{h_i(x)x^i}$.

In the instantiation of this specification with Cairo, there are only two composition column polynomials: $\mathrm{h(x)} = \mathrm{h_0(x)} + \mathrm{h_1(x)} \cdot \mathrm{x}$.

Finally, to allow the verifier to check that h has correctly been committed, Schwartz-Zippel is used with a random verifier challenge called the "oods point". Specifically, the verifier evaluates the following and check that they match:

- the left-hand side $\sum_{\mathrm{i}=0} \frac{\mathrm{C_i(oods\_point)}}{\mathrm{D_i(oods\_point)}} \cdot \alpha^{\mathrm{i}}$
- the right-hand side $\mathrm{h_0(oods\_point)} + \mathrm{h_1(oods\_point)} \cdot \mathrm{oods\_point}$

Of course, the verifier cannot evaluate both sides without the help of the prover! The left-hand side involves evaluations of the trace polynomials at the oods point (and potentially shifted oods points), and the right-hand side involves evaluations of the composition column polynomials at the oods point as well.

As such, the prover sends the needed evaluations to the verifier so that the verifier can perform the check. (These evaluations are often referred to as the "mask" values.)

With our previous example constraint, the prover would have to provide the evaluations of $\mathrm{f}_0(\mathrm{oods\_point}), \mathrm{f}_1(\mathrm{oods\_point}), \mathrm{f}_0(\mathrm{oods\_point} \cdot \mathrm{w}), \mathrm{h}_0(\mathrm{oods\_point}), \mathrm{h}_1(\mathrm{oods\_point})$.

Notice that this "oods check" cannot happen in the domain used to index the trace polynomials. This is because the left-hand side involves divisions by domain polynomials $\mathrm{D_i}(\mathrm{oods\_point})$, which might lead to divisions by zero.

If $\mathrm{oods\_point} = \mathrm{w}^3$ and the second constraint is associated to the whole 16-element domain, then the verifier would have to compute a division with $(\mathrm{w}^3)^{16} - 1$ which would be a division by zero.

This is why the oods point is called "out-of-domain sampling". Although nothing special is done when sampling this point, but the probability that it ends up in the trace domain is very low.

Aggregation and FRI Proof

The verifier now has to:

1. Perform a FRI check on $\mathrm{h}_0(\mathrm{x}) + \mathrm{x}\mathrm{h}_1(\mathrm{x})$ (which will verify the original prover claim that the trace polynomials satisfy the constraints).
2. Verify all the evaluations that were sent, the prover and the verifier can use FRI-PCS for that, as described in **the FRI-PCS section of the Starknet FRI Verifier specification**.

In order to avoid running multiple instances of the FRI protocol, the FRI Aggregation technique is used as described in **the Aggregating Multiple FRI Proofs section of the Starknet FRI Verifier specification**. The verifier sends a challenge called `oods_alpha` which is used to aggregate all of the first layer of the previously discussed FRI proofs.

Finally, the FRI protocol is run as described in **the Starknet FRI Verifier specification**.

Dependencies

In this section we list all of the dependencies and interfaces this standard relies on.

AIR Arithmetization Dependency

While this specification was written with Cairo in mind, it should be usable with any AIR arithmetization that can be verified using the STARK protocol.

A protocol that wants to use this specification should provide the following:

interactive arithmetization. A description of the interactive arithmetization step, which should include in what order the different tables are committed and what verifier challenges are sent in-between.

`eval_composition_polynomial`. A function that takes all of the commitments, all of the evaluations, and a number of Merkle tree witnesses sent by the prover and produces an evaluation of the composition polynomial at the oods point. (This evaluation will depend heavily on the number of trace columns and the constraints of the given AIR arithmetization.) The function is expected to verify any decommitment (via the Merkle tree witnesses) that it uses.

`eval_oods_polynomial`. A function that takes all of the commitments, all of the evaluations, and a number of Merkle tree witnesses sent by the prover and produces a list of evaluations of the first layer polynomial of the FRI check at a list of queried points. The function is expected to verify any decommitment (via the Merkle tree witnesses) that it uses.

We refer to the **Merkle Tree Polynomial Commitments specification** on how to verify decommitments.

Constants

We use the following constants throughout the protocol.

Protocol constants

`STARKNET_PRIME = 3618502788666131213697322783095070105623107215331596699973092056135872020481`. The Starknet prime ($2^{251} + 17 \cdot 2^{192} + 1$).

`FIELD_GENERATOR = 3`. The generator for the main multiplicative subgroup of the Starknet field. This is also used as coset factor to produce the coset used in the first layer's evaluation.

Channel

See the **Channel specification** for more details.

FRI

See the **Starknet FRI Verifier specification**.

Specifically, we expose the following functions:

- `fri_commit`
- `fri_verify_initial`
- `fri_verify_step`
- `fri_verify_final`

as well as the two objects `FriVerificationStateConstant`, `FriVerificationStateVariable` defined in that specification.

Configuration

```
struct StarkConfig {
    traces: TracesConfig,
    composition: TableCommitmentConfig,
    fri: FriConfig,
    proof_of_work: ProofOfWorkConfig,
    // Log2 of the trace domain size.
    log_trace_domain_size: felt252,
    // Number of queries to the last component, FRI.
    n_queries: felt252,
    // Log2 of the number of cosets composing the evaluation domain, where the coset size is the
    // trace length.
    log_n_cosets: felt252,
    // Number of layers that use a verifier friendly hash in each commitment.
    n_verifier_friendly_commitment_layers: felt252,
}
```

To validate:

- compute the log of the evaluation domain size as the log of the trace domain size plus the log of the number of cosets
    - if every coset is of size 2^{n_t} with n_t the **`log_trace_domain_size`**, and there is 2^{n_c} cosets, then the evaluation domain size is expected to be $2^{n_t+n_c}$
- validate the traces
- validate the composition
- the rest (proof of work, FRI configuration) is validated as part of the FRI configuration validation

Buiding Blocks

The verifier accepts the following proof as argument:

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

struct StarkUnsentCommitment {
    traces: TracesUnsentCommitment,
    composition: felt252,
```

```
    // n_oods_values elements. The i-th value is the evaluation of the i-th mask item polynomial
at
    // the OODS point, where the mask item polynomial is the interpolation polynomial of the
    // corresponding column shifted by the corresponding row_offset.
    oods_values: Span<felt252>,
    fri: FriUnsentCommitment,
    proof_of_work: ProofOfWorkUnsentCommitment,
}
```

We assume that the public input is instantiated and verified by the parent protocol, and thus is out of scope of this standard.

Domains and Commitments

Commitments to all the polynomials, before the FRI protocol, are done on evaluations of polynomials in the evaluation domain as defined in the **Domains and Commitments section of the Starknet FRI Verifier specification**.

Commitments to all the polynomials, before the FRI protocol, are done using table commitments as described in the **Table Commitments section of the Merkle Tree Polynomial Commitments specification**.

- For trace polynomials in the interactive arithmetization phase, the tables committed into the leaves represent the evaluations of each of the trace columns at the same point.
- For composition column polynomials in the composition polynomial phase, the tables committed into the leaves represent the evaluations of each of the composition columns at the same point.

STARK commit

The goal of the STARK commit is to process all of the commitments produced by the prover during the protocol (including the FRI commitments), as well as produce the verifier challenges.

1. **Interactive arithmetization to absorb the execution trace tables**:
    1. Absorb the original table with the channel.
    2. Sample the interaction challenges (e.g. z and alpha for the memory check argument (different alpha called memory_alpha to distinguish it from the alpha used to aggregate the different constraints into the composition polynomial)).
    3. Absorb the interaction table with the channel.
2. **Produce the aggregation of the constraint quotient polynomials as the composition polynomial**:
    1. Sample the alpha challenge ("composition_alpha") to aggregate all the constraint quotient polynomials (caches the powers of alpha into "traces_coefficients").
    2. Absorb the composition columns (the $\mathrm{h_i}$ in $\mathrm{h(x)} = \sum_{\mathrm{i}} \mathrm{h_i x^i}$) with the channel.
    3. Sample the oods point (`interaction_after_composition`).
    4. Absorb all evaluations with the channel.

5. Verify that the composition polynomial is correct by checking that its evaluation at the oods point is correct using some of the evaluations $\sum_{\mathrm{j}} \frac{\mathrm{C_j(oods\_point)}}{\mathrm{D_j(oods\_point)}} = \sum_{\mathrm{i}} \mathrm{h_i(oods\_point)} \times \mathrm{oods\_point^i}$.
    1. The right-hand side can be computed directly using the evaluations sent by the prover
    2. The left-hand side has to be computed using the `eval_composition_polynomial` function defined in the **AIR Arithmetization Dependency section**.
3. **Produce a challenge to aggregate all FRI checks and run the FRI protocol**:
    1. Sample the oods_alpha challenge with the channel.
    2. Call `fri_commit`.

STARK verify

The goal of STARK verify is to verify evaluation queries (by checking that evaluations exist in the committed polynomials) and the FRI queries (by running the FRI verification).

To do this, we simply call the `fri_verify_initial` function contained in the FRI specification and give it the values computed by `eval_oods_polynomial` (as defined in the **AIR Arithmetization Dependency section**) as first evaluations associated with the queried points. It should return two FRI objects.

These evaluations will only provide evaluations of the first layer polynomial $\mathrm{p_0}$ at query points $\mathrm{v_i}$. The prover will witness evaluations at $-\mathrm{v_i}$ by themselves and prove that they are present in the first FRI commitment (of the polynomial $\mathrm{p_0}$.

Full Protocol

The protocol is split into 3 core functions:

- `verify_initial` as defined below.
- `verify_step` is a wrapper around `fri_verify_step` (see the **FRI** section).
- `verify_final` is a wrapper around `fri_verify_final` (see the **FRI** section).

The verify initial function is defined as:

1. Validate the configuration and return the security_bits
2. Produce a stark domain object based on the configuration log_trace_domain_size and log_n_coset.
3. Validate the public input.
4. Compute the initial digest as `get_public_input_hash(public_input, cfg.n_verifier_friendly_commitment_layers, settings)`
5. Initialize the channel using the digest as defined in the **Channel** section.
6. Call STARK commit as defined in the **STARK commit** section.
7. Call STARK verify as defined in the **STARK verify** section and return the two FRI objects to the caller.

One can successively call them in the following order to verify a proof:

1. Call `verify_initial` on the proof and return:
    - the `FriVerificationStateConstant` object
    - the `FriVerificationStateVariable` object
    - the `last_layer_coefficients`
    - the security bits
2. Call verify_step in a loop on each layer of the proof (`n_layers` of them according to the StateConstant returned) and pass the FriVerificationStateVariable in between each calls
3. Call verify_final on the StateConstant and StateVariable objects
4. Enforce that the the StateVariable's iter field is `n_layers + 1`
5. Return the security bits.

Findings

Below are listed the findings found during the engagement.

ID	COMPONENT	NAME
00	*	Lack Of Specification Makes It Difficult To Verify The Implementation
01	air/public_input	Cairo Public Input Validation Might Not Be Sufficient
02	air/layouts/recursive/public_input.cairo	Insufficient check of public memory's main page length
03	air/layouts	Ambiguous Cairo0 Implementation Version Used
04	fri	Proofs Might Be Malleable Due To Unchecked FRI Commitments Length

00 - Lack Of Specification Makes It Difficult To Verify The Implementation

● *

Description. The Cairo implementation lacks a specification outlining the correct and expected behavior of the implementation. While one could compare this implementation to the reference Cairo0 implementation, this does not necessarily mean that the Cairo0 implementation is free of bugs and that this process would be sufficient to ensure the correctness of the Cairo implementation being audited.

The lack of a specification also makes it difficult to understand the expected behavior of the implementation. This could lead to misunderstandings and bugs in the implementation.

As part of this work, we have reverse engineered large chunks of the codebase and written up our understanding as more formal specifications. The current state of these specifications were included in the report, but more up-to-date specifications can be accessed at https://zksecurity.github.io/RFCs/.

Specifically, this includes four specifications: a FRI verifier specification, a STARK verifier specification, a Polynomial Commitment specification, and a Channel specification.

Recommendation. Use and improve the linked specifications to clarify the expected behavior of the implemented protocol. Consider performing another audit once the specifications are in a better place.

01 - Cairo Public Input Validation Might Not Be Sufficient

● air/public_input

Description. The implementation supports both Cairo 0 and Cairo 1 proofs. The verification of public inputs between the two is quite different, with a large amount of checks for Cairo 0 and a minimal amounts of check for Cairo 1:

```
fn verify_cairo1_public_input(public_input: @PublicInput) -> (felt252, felt252) {
    let public_segments = public_input.segments;

    let initial_pc = *public_segments.at(segments::PROGRAM).begin_addr;
    let initial_ap = *public_segments.at(segments::EXECUTION).begin_addr;
    let final_ap = *public_segments.at(segments::EXECUTION).stop_ptr;
    let output_start = *public_segments.at(segments::OUTPUT).begin_addr;
    let output_stop = *public_segments.at(segments::OUTPUT).stop_ptr;
    let output_len: u32 = (output_stop - output_start).try_into().unwrap();

    assert(initial_ap < MAX_ADDRESS, 'Invalid initial_ap');
    assert(final_ap < MAX_ADDRESS, 'Invalid final_ap');

    // TODO support continuous memory pages
    assert(public_input.continuous_page_headers.len() == 0, 'Invalid continuous_page_headers');

    let memory = public_input.main_page;

    // 1. Program segment
    assert(initial_pc == INITIAL_PC, 'Invalid initial_pc');
    let program = memory.extract_range_unchecked(0, memory.len() - output_len);
    let program_hash = poseidon_hash_span(program);

    // 2. Output segment
    let output = memory.extract_range_unchecked(memory.len() - output_len, output_len);
    let output_hash = poseidon_hash_span(output);
    (program_hash, output_hash)
}
```

During the audit it was not clear if these checks were sufficient. It is possible that the validity of the programs verified rely on important checks on the public inputs, both for Cairo 0 and Cairo 1 public inputs.

For example, in `src/air/layouts/*/public_input.cairo` we can observe these kind of checks:

```
let pedersen_copies = trace_length / PEDERSEN_BUILTIN_ROW_RATIO;
let pedersen_uses = (*self.segments.at(segments::PEDERSEN).stop_ptr
```

```
    - *self.segments.at(segments::PEDERSEN).begin_addr)
    / 3;
assert_range_u128_le(pedersen_uses, pedersen_copies);

let range_check_copies = trace_length / RANGE_CHECK_BUILTIN_ROW_RATIO;
let range_check_uses = *self.segments.at(segments::RANGE_CHECK).stop_ptr
    - *self.segments.at(segments::RANGE_CHECK).begin_addr;
assert_range_u128_le(range_check_uses, range_check_copies);

let bitwise_copies = trace_length / BITWISE_ROW_RATIO;
let bitwise_uses = (*self.segments.at(segments::BITWISE).stop_ptr
    - *self.segments.at(segments::BITWISE).begin_addr)
    / 5;
```

The values there represent the number of cells that a builtin occupies in the trace. So a Pedersen builtin takes 3 cells (2 inputs and 1 output), the range-check builtin takes 1 cell (the proof fails if the value in the cell is not in the range $[0, 2^{16})$), the bitwise builtin takes 5 cells.

Yet, the checks in these examples rely on the number of cells being a multiple of 3 (or 5) in the Pedersen (and bitwise builtin). It is not clear if malicious values could cause issues here.

Recommendation. In the absence of a specification, it is hard to determine if the checks are sufficient. Consider writing a specification and documenting the threat models in not checking the public input compared to more thorough checks of the public input. This applies to Cairo 0 public inputs as well.

02 - Insufficient check of public memory's main page length

- air/layouts/recursive/public_input.cairo

Description. The public memory is the portion of the memory that is visible to the verifier and is used by the verifier to check the validity of the STARK proof. Given a public memory, the verifier needs to use the public memory length value, so it verifies the public memory length by reading the public memory until the end and checking that the lastly read value is equal to the last element of the memory. However, this check is insufficient if the lastly read value is not actually the last element of the array, but has the same value as the actual last element array.

```
// Check main page len
assert(
    *memory.at(memory_index - 1) == *self.main_page.at(self.main_page.len() - 1), 'Invalid main
page len'
);
```

(Note that `memory = self.main_page`, so the check is equivalent to `*memory.at(memory_index - 1) == *memory.at(memory.len() - 1)`.) Here, even if `memory_index - 1` is not equal to `memory.len() - 1`, the check will still pass as long as the last read value is equal to the last element of the memory.

Recommendation. We recommend replacing the current check with the following code.

```
// Check main page len
assert(memory_index == self.main_page.len(), 'Invalid main page len');
```

03 - Ambiguous Cairo0 Implementation Version Used

- **air/layouts**

Description. The current Cairo verifier implementation relies on a previous Cairo 0 implementation for a number of auto-generated files. This is visible in the following snippet from `/src/air/layouts/_generator/main.py`:

```
for layout, settings in layout_settings.items():
    handle_github_file(
        f"https://raw.githubusercontent.com/starkware-libs/cairo-
lang/master/src/starkware/cairo/stark_verifier/air/layouts/{layout}/autogenerated.cairo",
        f"../{layout}/autogenerated.cairo",
        layout,
        settings
    )
```

As can be seen, the files that are fetched are always from the `master` branch of the `cairo-lang` repository, which might point to a different commit depending on when the verifier is built. This could lead to different versions of the verifier being used in different environments, which could in turn lead to unexpected behavior.

Recommendation. Instead of the `master` branch, an actual commit should be hardcoded.

04 - Proofs Might Be Malleable Due To Unchecked FRI Commitments Length

● fri

Description. A FRI proof comes with a number of FRI commitments: one for each layer that is not skipped, excluding the first and last layers. This is visible in the `inner_layers` field of the following structure:

```
struct FriUnsentCommitment {
    // Array of size n_layers - 1 containing unsent table commitments for each inner layer.
    inner_layers: Span<felt252>,
    // Array of size 2**log_last_layer_degree_bound containing coefficients for the last layer
    // polynomial.
    last_layer_coefficients: Span<felt252>,
}
```

Yet, the length of this `Span` is not validated anywhere. As can be seen in `src/fri/fri.cairo` the logic uses the claimed length (`n_layers`) instead of iterating through the whole `Span`. This could lead to a malleability issue where an attacker could provide a proof with a larger number of commitments than expected, which would be ignored and would still produce valid proofs.

```
fn fri_commit_rounds(
    ref channel: Channel,
    n_layers: felt252,
    configs: Span<TableCommitmentConfig>,
    unsent_commitments: Span<felt252>,
    step_sizes: Span<felt252>,
) -> (Array<TableCommitment>, Array<felt252>) {
    // TRUNCATED...
    let mut i: u32 = 0;
    let len: u32 = n_layers.try_into().unwrap();
    loop {
        if i == len {
            break;
        }
        // Read commitments.
        commitments.append(table_commit(ref channel, *unsent_commitments.at(i),
*configs.at(i)));
        // TRUNCATED...
        i += 1;
    };
    // TRUNCATED...
}
```

```
fn fri_commit(
    ref channel: Channel, unsent_commitment: FriUnsentCommitment, config: FriConfig
) -> FriCommitment {
    // TRUNCATED...
    let (commitments, eval_points) = fri_commit_rounds(
        ref channel,
        config.n_layers - 1,
        config.inner_layers,
        unsent_commitment.inner_layers,
        config.fri_step_sizes,
    );
    // TRUNCATED...
}
```

Recommendation. Enforce in `fri_commit` that the length of the commitments are as expected.