Rescue STARK Documentation – Version 1.0

StarkWare Team[*]

July, 2020

Abstract

This document is intended to accompany the ethSTARK codebase, describing the computational integrity statement proved by that code and the specific STARK construction used to prove the statement.

[*]StarkWare Industries Ltd., Israel. Send inquiries to `info@starkware.co`.

Contents

1 Introduction

On July 2, 2018, the Ethereum Foundation gave StarkWare a 2-year milestone-based grant to select a STARK friendly hash (SFH) function, to be used in combination with transparent and plausibly post-quantum secure proof systems within blockchains, and release an open source efficient STARK system for it. Under the grant agreement, StarkWare committed to publishing, among other things:

> *"Production-quality software released under a software license, approved by the Ethereum Foundation, for the STARK-friendly hash function:*
>
> 1. *Arithmetised circuit with proofs compressing 100,000 hashes (3.2MB of data) to 200kB with 80 bits of security*
> 2. *Prover that compresses 100 hashes per second with a quad-core CPU and 16GB of RAM*
> 3. *Verifier verifying proofs in 10ms on a single-core CPU with 4GB of RAM*
> 4. *Detailed specification of the prover and verifier mechanics, inluding optimisations . . ."*

The ethSTARK code released by StarkWare answers items 1–3 above, and the purpose of this document it to address item 4. We assume familiarity with the notion of interactive proofs [GMR89], non-interactive Computationally Sound proofs [Mic00], Interactive Oracle Proofs (IOPs) [RRR16, BCS16] and Scalable Transparent ARgument of Knowledge (STARK) systems [BBHR19].

Organization of the document Section 2 describes the Rescue hash function family and the particular member of it that ethSTARK implements. Section 3 describes in great detail the specific STARK protocol used in the code, including a description of the full system of constraints included in the Algebraic Intermediate Representation (AIR) of the system. Section 4 provides measurements and benchmarks of the system and discusses them. Security and soundness analysis are omitted in this initial version and will be added in a future version of the paper.

Acknolodgement We would like to thank Justin Drake from the Ethereum Foundation for his thoughtful and detailed comments.

2 Rescue

In this section we give a short description of the Rescue cipher family, referring the interested reader to [AAB+19] for full details. We then present an instantiation of a hash function based on this Rescue cipher family.

2.1 Rescue Cipher

Rescue is a family of ciphers based on substitution-permutation networks (SPNs). A Rescue cipher manipulates a state of $m > 1$ elements in the vector space $\mathbb{F}_p^m$ where $\mathbb{F}_p$ is a field of characteristic $p \equiv 5 \pmod 6$.

A Rescue permutation is an iterative application of a round function R times where R is determined by the desired security level. The inputs to the first round are the plaintext and a master-key, and the output of the last round is the ciphertext. Each round takes as inputs the previous state and a subkey, derived from the master-key, and outputs a new state.

A round of a Rescue permutation includes two steps. In each step an S-box is applied to each of the m state elements, followed by a multiplication by a Maximum Distance Separable (MDS) matrix which mixes the elements together. At the end of each step a subkey is injected into the state. The S-boxes π_1 and π_2 that are used in the first and second step of each round, consist of the power maps $x^{1/\alpha}$ and x^{α}, respectively, for an integer α that does not divide $p-1$ (in which case $1/\alpha$ is well-defined).

2.2 Rescue Hash Function

The Rescue hash function is a sponge construction hash function, based on an un-keyed Rescue permutation, in which the secret key is set to zero and round constants are used instead of keys. A sponge construction generates a hash function from an underlying permutation by iteratively applying it to a large state. The state consists of $m = r + c$ field elements, where r and c are called the *rate* and the *capacity* of the sponge, respectively.

We now present our instantiation of a Rescue hash function. Henceforth, the term "Rescue" refers to this particular instantiation, not to the larger family defined in [AAB+19]. The native field in which Rescue operates is $\mathbb{F}_p$ where $p = 2^{61} + 20 \cdot 2^{32} + 1$. The state is viewed as a column vector of $m = 12$ field elements. For the S-boxes π_1 and π_2 we use $\alpha = 3$ such that the power maps are $x^{1/3}$ and x^3, respectively. Since 3 does not divide $p-1$ it holds that $(2p-1)/3$ is an integer and furthermore,

$$\forall x \in \mathbb{F}_p, \quad \left(x^3\right)^{(2p-1)/3)} = \left(x^{(2p-1)/3}\right)^3 = x.$$

Therefore we use $1/3$ to denote $(2p-1)/3$, noticing that $x \mapsto x^{1/3}$ is indeed the cube-root permutation, modulo p. To compute the Rescue permutation from a given input, the round function is iterated $R = 10$-times with constants injected before the first round, between each two consecutive steps (within and between rounds), and after the last round (a total of 21 constant vectors).

Let $K = \{K_0, \ldots, K_{20}\}$ denote the constants used in the Rescue hash function such that K_{2r+1}, K_{2r+2} are the constants used in the rth round for $r \in [0, 9]$ and K_0 is the constant used before the first round. Note that each $K_i \in K$ is in fact a field element vector of length $m = 12$. Thus, adding a constant to a state is merely a vector addition. Figure 1 is a graphic description of a single round of the Rescue permutation.

To transform the Rescue permutation to a hash function, we apply the Sponge construction: The first 8 elements of the *state* are the *rate* and the last 4 elements are the *capacity*. The hash of two inputs $w_0, w_1 \in \mathbb{F}_p^4$ is defined by applying the rescue permutation to $(w_0, w_1, 0) \in \mathbb{F}_p^{12}$ and taking the first 4 elements. A graphic description of the Rescue hash function is given in Fig. 2, and its pseudo-code appears in Algorithm 1.

3 The STARK Protocol

STARKs (Scalable Transparent ARguments of Knowledge) are a family of proof systems characterized by scalability and transparency. Scalability – via quasilinear proving time and poly-logarithmic verification time, and transparency – meaning all verifier-side messages are public random coins (requiring no trusted setup). We assume familiarity with the general definition of STARKs, as described in [BBHR19]. In this section we describe the STARK proof system as an interactive protocol

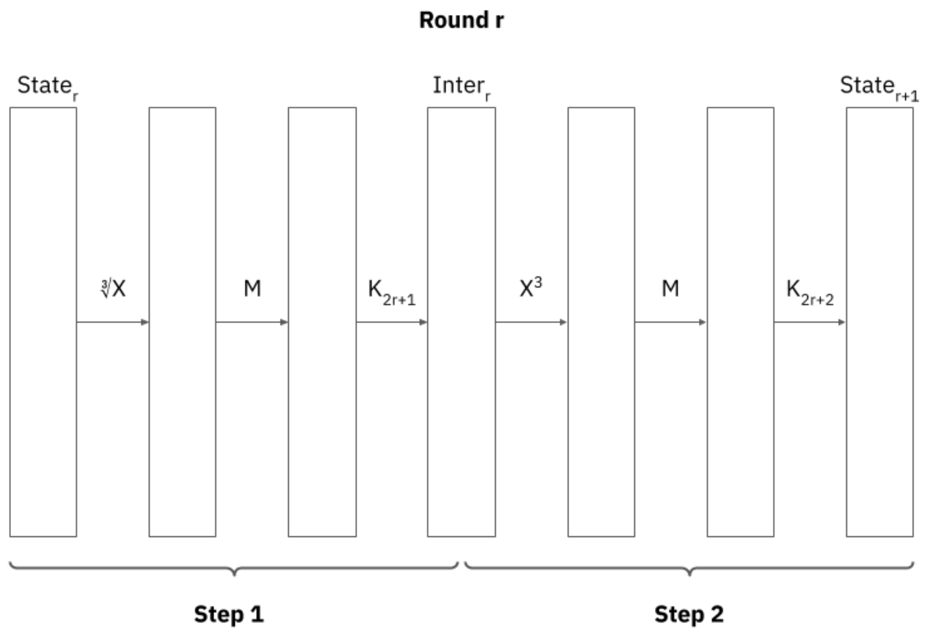


Figure 1: Round r of the Rescue permutation. M denotes a multiplication by the MDS matrix. $\texttt{Inter}_\texttt{r}$ represents the state in the middle of the round.

Algorithm 1 The Rescue permutation with an MDS matrix M

```
INPUT: $w_0, w_1 \in \mathbb{F}_p^4$, round constants $K$
OUTPUT: Rescue$(w_0, w_1)$
  Let $\texttt{State}_\texttt{in}$ be the vector $(w_0, w_1, 0) \in \mathbb{F}_p^{12}$.
  $\texttt{State}_0 = \texttt{State}_\texttt{in} + K_0$
  for $r = 0$ to 9 do
    for $i = 0$ to 11 do
      $\texttt{Inter}_\texttt{r}[i] = \sum_{j=0}^{m-1} M[i,j](\texttt{State}_\texttt{r}[j])^{1/3} + K_{2r+1}[i]$
    end for
    for $i = 0$ to 11 do
      $\texttt{State}_\texttt{r+1}[i] = \sum_{j=0}^{m-1} M[i,j](\texttt{Inter}_\texttt{r}[j])^{3} + K_{2r+2}[i]$
    end for
  end for
  return $\texttt{State}_{10}$
```

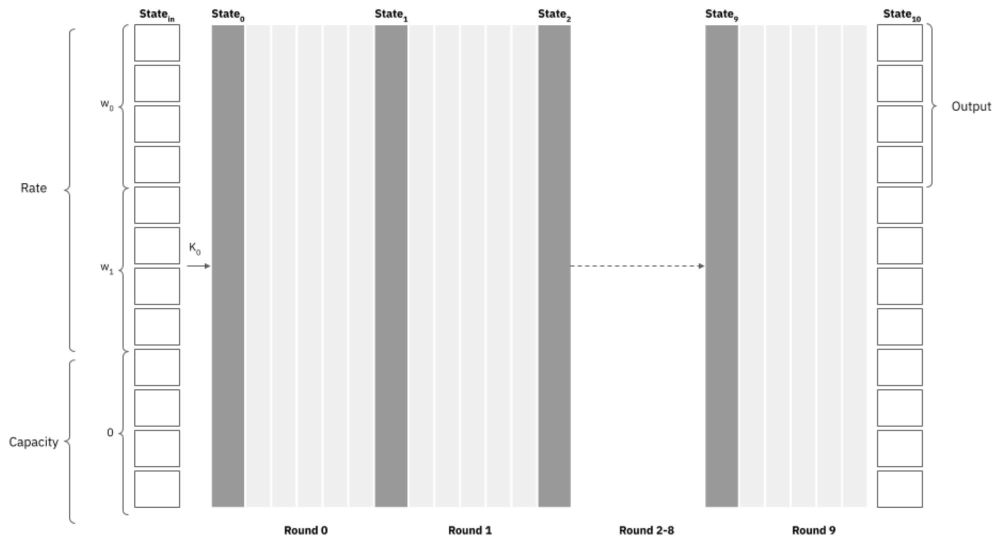


Figure 2: Instantiating the Rescue hash function as a sponge construction based on the Rescue permutation.

between two parties, the prover and the verifier. The prover sends a series of oracle messages in an attempt to convince the verifier that a certain computation on some input (the proven statement) was executed correctly. The verifier responds to the messages with public random values. After the interaction ends, the verifier uses more public random coins to query a small number of entries from the oracle messages sent by the prover. Based on the answers to these queries, the verifier reaches a decision whether to accept the statement as correct, or reject it. The completeness and soundness properties of the STARK system imply that correct statements proved by honest provers are guaranteed to be accepted by the verifier (with probability 1 over the random coin-tosses made by the verifier). Conversely, incorrect statements, and statements whose witness is unknown to the prover will be rejected with all but negligible probability (which we set here to be at most 2^{-80}).

While we describe the system below as an interactive protocol, it is noted that this interactivity is eventually replaced by a transformation to a non-interactive system wherein the prover provides a proof and the verifier decides whether to accept or reject it. See Section 3.10.

3.1 The Trace

An *execution trace* of a computation, or *trace*, in short, is a sequence of machine states, one per clock cycle. If the computation requires W registers and lasts for N cycles, the execution trace can be represented by a table of size $N \times W$. Given a statement regarding the correctness of a computation, the prover first builds a trace.

Denote the columns of the trace by $f_1, \ldots, f_W$. Each f_j is of a fixed length, N, that is a power of two. The values in the trace cells are elements in a finite field[1] $\mathbb{F}_p$. The *trace evaluation domain* is defined to be a multiplicative subgroup of $\mathbb{F}_p^{\times}$ of size N, generated by an element g; we denote

[1] We use the same field in which Rescue operates ($\mathbb{F}_p$ where $p = 2^{61} + 20 \cdot 2^{32} + 1$).

this subgroup by $\langle g \rangle$. Effectively, we enumerate the trace rows using the elements of $\langle g \rangle$, where the ith row is enumerated by g^i (the first row is the 0th row, enumerated by $1 = g^0$). Each trace column is interpreted as N point-wise evaluations of a polynomial[2] of degree smaller than N over the trace evaluation domain. These polynomials are referred to as the *trace column polynomials* or *column polynomials* in short.

The Rescue trace has 12 columns, corresponding to the $m = 12$ field elements of the state. Applying each hash requires slightly more than 10 rows (one per round). The hashes[3] can be computed in batches of 3 hashes that fit into 32 rows as follows (see Fig. 3):

- Row 0: initial state of the first hash (8 input field elements and 4 zeros).
- Rows 1 to 10: state in the middle[4] of every round of the first hash.
- Rows 11 to 20: state in the middle of every round of the second hash.
- Rows 21 to 30: state in the middle of every round of the third hash.
- Row 31: final state of the third hash (the first 4 field elements in this state are the output).

3.2 Periodic Columns

Many cryptographic primitives involve using some list constants. Applying the same cryptographic primitive many times, yields a periodic list of constants. For this, we use a technique we refer to as *periodic columns*. The periodic structure of each such column leads to a column polynomial which can be represented succinctly. In the classic representation of a polynomial $\sum a_i x^i$ as a vector of its coefficients $(a_0, a_1, \ldots)$, a succinct representation means that most of the a_is are zeros. This enables the verifier to efficiently compute the point-wise evaluations of these polynomials.

We maintain the round constants of Rescue using periodic columns. For each trace column we have two periodic columns, one for each of the two steps of a round, up to the following small modifications. Each periodic column is of length 32 (corresponding to 3 hashes, see Section 3.1 for more details). For technical reasons that will be explained in Section 3.3, we decided to add K_0 to the round constants that correspond to the second step of a round, in the first four columns in rows 10 and 20 (the left inputs of the second and third hash invocations).

3.3 The Constraints

An execution trace is *valid* if (1) certain boundary constraints hold and (2) each pair of consecutive states satisfies the constraints dictated by the computation. For example, if at time t the computation should add the contents of the 1st and 2nd registers and place the result in the 3rd

[2] Such interpolation polynomial (uniquely) exists since for any N distinct points $x_0, \ldots, x_{N-1}$ and corresponding values $y_0, \ldots, y_{N-1}$, there exists a unique polynomial of degree at most $N - 1$ that interpolates the data $(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})$.

[3] A chain of n hash invocations consists of $n+1$ inputs $w_0, \ldots, w_n$, and a single output. Let O_i denote the output of the ith invocation, the output of the chain is O_n. The inputs to the first invocation are (w_0, w_1) and the inputs to the ith invocation, for $i \geq 2$, are (O_{i-1}, w_i).

[4] We refer to the value of $\texttt{Inter}_\texttt{r}$ after the first inner loop in Algorithm 1 as the state in the middle of every round.

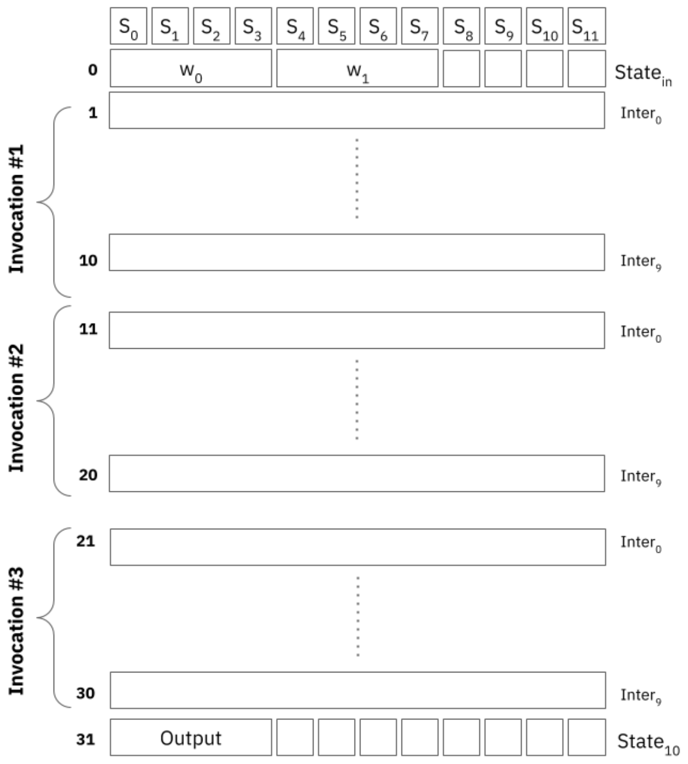


Figure 3: A batch of three hashes in the Rescue execution trace.

register, the relevant constraint would be $f_1(g^t) + f_2(g^t) - f_3(g \cdot g^t) = 0$ where f_j is the jth column polynomial and g is the generator of the trace evaluation domain.

The constraints are expressed as polynomials composed over the trace cells that are satisfied if and only if the computation is correct. Hence, they are referred to as the *Algebraic Intermediate Representation (AIR) Polynomial Constraints* on the trace. For example, in the context of proving the computational integrity of an execution of a Rescue hash function, the constraints are such that they all hold if and only if the output of the hash function matches the input, where the input and output are designated cells within the trace (these cells will be part of the boundary constraints defining validity of the statement).

Some examples of constraints over the trace cells:

1. $f_2(x) - a = 0$ for $x = g^7$ (the value in column 2 row 7 is equal to a).
2. $f_6^2(x) - f_6(g^3 x) = 0$ for all x (the squared value in each row in column 6 is equal to the value three rows ahead).

By writing a set of polynomial constraints which are satisfied if and only if the computation is valid, we reduce the original problem – proving the correctness of a computation – to proving that the polynomial constraints are satisfied. This reduction is a special case of the general process known in theoretical computer science as "arithmetization".

The AIR for the Rescue hash chain corresponds to the following claim:

"I know a sequence of inputs $w = \{w_0, \ldots, w_n\}$ such that

$$H(\ldots H(H(w_0, w_1), w_2) \ldots, w_n) = \mathtt{output}\text{"}, \tag{1}$$

where H is the Rescue hash function, each w_i is a 4-tuple of field elements and `output` is the public output of the hash (which consists of 4 field elements). Recall that the hashes are computed in batches of 3 hashes, hence, $|w| = 3k + 1 = n + 1$ for some $k \in \mathbb{Z}$. We note that the number of hash invocations, which is known to the verifier, is $|w| - 1 = n$ and refer to n as the *chain length.*

3.3.1 Intermediate Values

There are numerous ways to capture the correctness of an execution trace via polynomial constraints. When designing an AIR, one should take the tradeoffs that each approach yields into consideration. For example, consider the following synthetic trace with only two cells:

X
$\sqrt[3]{X+1}$

A naïve constraint linking the two cells is $f_0(g) - \sqrt[3]{f_0(1) + 1} = 0$. Recall that in Rescue's native field $x^{1/3} = x^{(2p-1)/3}$. Therefore, the degree[5] of this polynomial constraint, $(2p-1)/3$, is huge.

A different possible approach would be to maintain intermediate execution values within the trace. For example, adding a column to the trace with the cubed values of the original column:

[5] In the following sections we will see that the degrees of the polynomial constraints play a major role in the efficiency of the prover.

X	X^3
$\sqrt[3]{X+1}$	$X+1$

Now, we can replace the former constraint with the following ones:

$$f_0^3(x) - f_1(x) = 0 \text{ for } x = 1, g \tag{2}$$

$$f_0(1) + 1 - f_1(g) = 0 \tag{3}$$

Note that while the maximal degree of these constrains is 3, there are now three constraints instead of one. Also, the size of the trace is twice the size of the original trace. Both measures, number of constraints and trace size, affect the prover efficiency. Crucially, in both approaches we force the prover to place $\sqrt[3]{X+1}$ in a certain well-defined trace cell, as needed to achieve computational integrity.

A third approach is to compute the intermediate values for the constraints in which they are used, instead of adding them to the trace and asserting their validity by more constraints. Consequently, constraint (3) becomes $f_0(1) + 1 - A = 0$, where $A = f_0^3(g)$ and the intermediate value A does not appear in the trace. Note that this constraint is defined over the original trace and is of degree 3. Although intermediate values are not part of the trace, it is helpful to think of them as trace intermediate columns.

Recall that the rows in the Rescue trace are the states in the middle of rounds (except for the first and last rows of every batch of 3 hashes). Thus, as can be seen in Figure 4, calculating the second step of a round from a row in the trace should yield the same result as calculating the first step of a round, in reverse, from its consecutive row.

In the Rescue AIR, for each trace column we have 3 intermediate columns. We denote these intermediate columns by:

1. `x_cube`: computes the third powers of the state. Corresponds to transition A in Figure 4.
2. `after_linperm`: computes the state at the end of a full round (half round forward from the current row). Corresponds to transition B in Figure 4.
3. `before_next_linperm_cubed`: computes the state at the beginning of the next full round (half round backward from the next row). Corresponds to transition C in Figure 4. Note that this intermediate value depends on the next row, so for a given column polynomial $f(x)$, `before_next_linperm_cubed` corresponds to $f(gx)$.

In the following section, for each intermediate column we use brackets notation to denote the corresponding trace column. For example, `x_cube`$[j](g^i)$ refers to the third power of the value of the cell in the jth column and the ith row.

3.3.2 The Rescue Constraints

We now describe the constraints used in the Rescue AIR. For each constraint, we first describe the meaning of the constraint, that is, what the constraint enforces on the trace values. We then state the polynomial constraint itself, followed by the domain and columns to which the constraint should apply.

In the following constraints let K denote the constants used in Rescue as in Section 2. We write $K_i[j]$ for the constant used in the ith step of the algorithm for the jth column. As mentioned above, we pack 3 hash invocations into 32 trace rows.

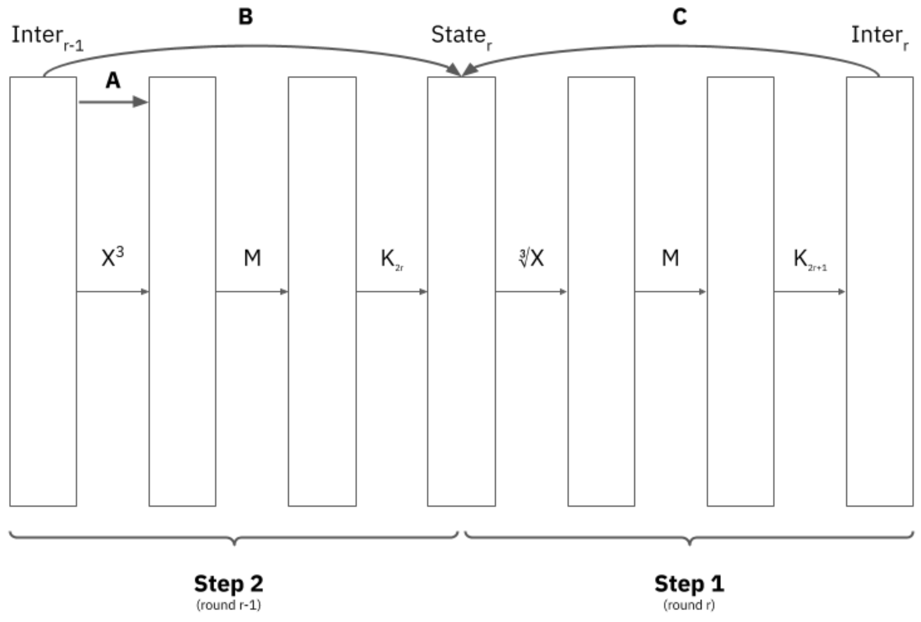


Figure 4: A "shifted" round of the Rescue permutation. Represents the execution between consecutive rows of the trace.

1. The capacity part at the beginning of each hash is zero.

   (a) The capacity part of the first row of each batch of 3 hashes is zero.

   $$f_j(x) = 0$$

   for $x = g^i$ where $i \equiv 0(32)$ and $j \in [8, 11]$.

   (b) The capacity part of the second and third hashes is zero.

   $$K_0[j] - \texttt{before\_next\_linperm\_cubed}[j](x) = 0$$

   for $x = g^i$ where $i \equiv 10, 20(32)$ and $j \in [8, 11]$: First row of the second and third hashes in each batch.

   To see why this holds, recall that between consecutive hashes the capacity is nullified and the constant K_0 is injected before the first step of each invocation.

2. The second row of each batch is obtained by applying the first half round of Rescue.

   $$f_j(x) + K_0[j] - \texttt{before\_next\_linperm\_cubed}[j](x) = 0$$

   for $x = g^i$ where $i \equiv 0(32)$ and $j \in [0, 11]$.

3. The connection between the middle of a round (current row) with the middle of the next round (next row).

STARKWARE

(a) For $\texttt{State}[0], \ldots, \texttt{State}[11]$.

$$\texttt{after\_linperm}[j](x) - \texttt{before\_next\_linperm\_cubed}[j](x) = 0$$

for $x = g^i$ where $i \not\equiv 0, 10, 20, 30, 31 (32)$ and $j \in [0, 11]$: All rows except for the first and last rows of each batch and the last row of each hash.

Note that this constraint does not apply to rows 10 and 20 since in the connection between consecutive hashes, the capacity is nullified and the right input ($\texttt{State}[4], \ldots, \texttt{State}[7]$) is reset to some nondeterministic witness. However, between consecutive hashes, the left input ($\texttt{State}[0], \ldots, \texttt{State}[3]$) must be equal to the output of the former hash. Since we add K_0 to the round constants that correspond to the constants used in the second step of a round in the first four columns in rows 10 and 20 (See Section 3.2), we have the following constraint.

(b) For $\texttt{State}[0], \ldots, \texttt{State}[3]$.

$$\texttt{after\_linperm}[j](x) - \texttt{before\_next\_linperm\_cubed}[j](x) = 0$$

for $x = g^i$ where $i \equiv 10, 20 (32)$ and $j \in [0, 3]$.

To see why this holds, recall that the state after the first hash (corresponds to $\texttt{after\_linperm}$) is: OUTPUT0|JUNK|JUNK. The state before the second hash (which corresponds to $\texttt{before\_next\_linperm\_cubed}$ - K_0) is: INP0|INP1|0. We require that OUTPUT0 = INP0. But instead of writing the equation:

$$\texttt{after\_linperm} = \texttt{before\_next\_linperm\_cubed} - K_0,$$

we in fact add K_0 to the round constants that correspond to the constants used in the second step of a round in the first four columns in rows 10 and 20, and thus $+K_0$ is already part of $\texttt{after\_linperm}$.

4. The connection between the last two rows of each batch (final half round of the third hash).

$$\texttt{after\_linperm}[j](x) - f_j(gx) = 0$$

for $x = g^i$ where $i \equiv 30(32)$ and $j \in [0, 11]$: Last row of the third hash in each batch.

5. The output of the third hash of a batch is the input of the first hash of the next batch.

$$f_j(x) - f_j(gx) = 0$$

for $x = g^i$ where $i \equiv 31(32), i < N - 1$ and $j \in [0, 3]$ where N is the length of the trace: Last row of the third hash in each batch except for the last row of the trace.

6. The output of the hash chain is the expected output.

$$f_j(x) - \texttt{output} = 0$$

for $x = g^{32(\texttt{chain\_length}/3-1)+31}$ where $j \in [0, 3]$, $\texttt{chain\_length}$ is the number of hash invocations and $\texttt{output}$ is the expected output of the hash chain.

Note, each of the constraints listed above represents multiple constraints. For example, the last constraint (6) represents four constraints that correspond to the first four columns of the trace (as the output of the function is the first four elements of the state's rate). There is a total of 52 constraints in the Rescue AIR[6].

3.3.3 From Polynomial Constraints to Low Degree Testing Problem

Next, we represent each constraint as a rational function. Recall that the trace evaluation domain is of order N and generated by g. Hence, $\langle g \rangle = \{x \in \mathbb{F}_p \mid x^N = 1\}$. Constraint (1a) above is translated into the rational function:

$$C_1(x) = \frac{f_j(x)}{x^{N/32} - 1} \tag{4}$$

which is a polynomial of degree at most $\deg(f_j) - N/32$ if and only if the constraint holds over $\langle g \rangle$.

Represented as rational functions, the constraints are such that each numerator defines a relevant rule needed to be enforced over the trace cells, and each denominator defines the domain in which the corresponding rule should hold.

Two remarks are in order. First, in order for the process of representing each constraint as a rational function to be well-defined, we need to make sure that the denominators are never zero. As we will see in the next section, the constraint polynomials will not be evaluated over the trace evaluation domain, but rather on a (larger) *disjoint* domain, which we call the *evaluation domain*. Thus, while the denominator can zero out over a subset of the trace evaluation domain, no denominator equals zero over the evaluation domain and expressions like Eq. (4) will be well-defined over the evaluation domain. Second, since the verifier needs to evaluate these rational functions, it is important for the succinctness of the STARK protocol that the domains are such that their corresponding denominators can be evaluated efficiently, i.e., that all high-degree polynomials be sparse, as indeed is the case with Eq. (4).

3.4 Trace Low Degree Extension

Recall that each trace column is viewed as N evaluations of a polynomial of degree less than N. In order to achieve a secure protocol, each such polynomial is evaluated over a larger domain, 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 between the size of the evaluation domain and the trace evaluation domain as the *blowup factor*, β. (Those familiar with coding theory notation will notice that β is the inverse of the rate and the LDE is in fact simply a Reed-Solomon code of the trace.)

The trace LDE is computed in two steps. First, we calculate the interpolation polynomial of each trace column using the Inverse Fast Fourier Transform (IFFT). Then, we evaluate each interpolation polynomial on the evaluation domain using the Fast Fourier Transform (FFT).

In order to make sure that the evaluation domain and the trace evaluation domain are disjoint, we use a non-unit[7] coset of the multiplicative subgroup of size $(\beta \cdot N)$ of $\mathbb{F}_p^\times$ as the evaluation

[6]Constraint 3 as described above, represents 16 constraints, which leads to a total of 56 constraints. However, in our implementation, we write constraint 3a for $i \not\equiv 0, 10, 20, 30, 31 (32)$ and $j \in [4, 11]$ and constraint 3b for $i \not\equiv 0, 30, 31 (32)$ and $j \in [0, 3]$. Resulting in 12 constraints for constraint 3 and a total of 52 constraints.

[7] By non-unit coset we mean a coset with offset different than 1. Specifically, we use the generator 3 of $\mathbb{F}_p^\times$ as the coset's offset.

domain.

For Rescue, we use a blowup factor of 4 so the evaluation domain is of size $4 \cdot N$.

3.5 Commitment Scheme

Following the generation of the trace LDE, the prover commits to it. Throughout the system, commitments are implemented by building Merkle trees over the series of field elements and sending the Merkle roots to the verifier. We use BLAKE2s with digest size of 20 bytes, or 160 bits, as the underlying hash function, to reach 80-bit security.

To gain better efficiency we use two optimizations for the Merkle tree implementation.

1. The leaves of the Merkle tree are selected such that if a decommitment is likely to involve multiple field elements together, they are grouped into a single Merkle leaf. In the case of the trace LDE, this implies we group all field elements in a trace LDE "row" into a single Merkle leaf.

2. When the size of each element (Merkle leaf) is smaller than the input size of the hash used by the Merkle tree, feeding individual elements into such a tree is wasteful. Instead, we group several elements together into a package that fits as a single input to the hash and use these packages as the input elements for the tree.

3.6 Composition Polynomial

In order to efficiently prove the validity of the execution trace, we strive to achieve the following two goals:

1. Compose the constraints on top of the trace polynomials to enforce them on the trace. (Described in Section 3.3.)

2. Combine the constraints into a single (larger) polynomial, called the *Composition Polynomial*, so that a single low degree test can be used to attest to their low degree. To reach this goal, some of the composed polynomials will require adjustment to their degrees, so that all composed polynomials have the same designated degree (the ethSTARK code indeed enforces this degree adjustment).

Before we continue to describe how the above is performed, we introduce another finite field. Recall that the elements in the trace are from $\mathbb{F}_p$, Rescue's native field, whose size is between 2^{61} and 2^{62}. In order for the STARK protocol to be secure, one must use a larger field in some places. For this, we use the quadratic extension field $\mathbb{F}_p(\phi)$ where ϕ is a root of the irreducible polynomial $X^2 - X - 1$. Thus, $\mathbb{F}_p(\phi)$ is the field $\mathbb{F}_p[X]/(X^2 - X - 1)$. In the following sections it is crucial, for the soundness of the protocol, that each field element used in the protocol is from the appropriate field.

3.6.1 Degree Adjustment

In order to ensure soundness, we need to show that all individual constraints composed with the trace column polynomials are of low degree. Let `max_deg` be the highest degree of all the constraints. We adjust the degree of the constraints to degree $D-1$, where D is the smallest power of 2 such that $D >$ `max_deg`.

Degree adjustment is performed as follows: Given a constraint $C_j(x)$ of degree D_j, we define a polynomial of the form:

$$C_j(x)(\alpha_j x^{D-D_j-1} + \beta_j)$$

where α_j and β_j are random field elements from the extension field $\mathbb{F}_p(\phi)$, chosen by the verifier. As a result, if the new constraint is of degree lower then D, it automatically follows (w.h.p) that the original constraint is of degree at most D_j, as desired.

3.6.2 Combining the Constraints

Once the prover has committed to the trace LDE, the verifier provides random coefficients for creating a random linear combination of the constraints[8] resulting in the composition polynomial. Instead of checking each constraint individually, it suffices to apply a low degree test to the composition polynomial.

Thus, the composition polynomial takes the form:

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

where k is the number of constraints.

Since the constraints used in the Rescue AIR are of degree three or below, the degree of the composition polynomial is $< 4N$.[9] Hence, we can represent the composition polynomial $h(x)$ as a single column of evaluations of length $4N$. Instead, we prefer to represent it as four columns $h_0(x), \ldots, h_3(x)$ of length N, where $h(x) = h_0(x^4) + xh_1(x^4) + x^2h_2(x^4) + x^3h_3(x^4)$. We "break" $h(x)$ into $h_i(x)$ by computing partial (two layer) IFFT.

3.6.3 Committing to the Composition Polynomial

Next, the prover performs yet another low degree extension of the four composition polynomial columns $h_0(x), \ldots, h_3(x)$. As these columns are of the same length as the trace columns, we sometimes refer to them as the *Composition Polynomial Trace* and we address extending and committing to them in the same manner as with the execution trace. This step includes extending them by the same blowup factor, grouping the rows (of field element quadruples) into leaves of a Merkle tree, calculating the hash values and sending the root of the tree as the commitment.

3.7 Consistency Check on a Random Point (the DEEP Method)

The value of $h(x)$ for a given point (an extension field element) $z \in \mathbb{F}_p(\phi)$ can be obtained in two ways: by calculating the above mentioned linear combination of constraints (the composition polynomial) or from $h_0(z^4), \ldots, h_3(z^4)$. For the former, the composition polynomial calculation induces a set of points over the trace columns that are needed in order to compute $h(z)$. This set of points, required to calculate $h(x)$ for a single point, is called the *mask*. Hence, given a point z, we can check the consistency between the commitment on the execution trace and the commitment

[8] There are 52 constraints in Rescue, as described in Section 3.3.2. Therefore, in Rescue the verifier sends 104 random field elements – two for each constraint, as described in Section 3.6.1.

[9] The constraint used in the Rescue AIR are of degree three, but we prefer to have a degree bound for the composition polynomial which is a power of two.

on the composition trace. For this we need the values of the induced mask on the trace and the values of $h_0(z^4), \dots, h_3(z^4)$.

Recall that in the Rescue AIR, the constraints assert the transition between consecutive rows (except for the boundary constraints, which deal with the first and last rows of the trace). Thus, the mask of a given point z in the Rescue AIR consists of 24 elements. For each of the 12 polynomial columns, $f_j(x)$, there are two mask points: $f_j(z)$ and $f_j(g \cdot z)$, where g is the generator of the trace evaluation domain.

At this phase, the verifier sends a randomly sampled point $z \in \mathbb{F}_p(\phi)$. The prover sends back 28 elements: the evaluations of the relevant elements in the mask required for calculating $h(z)$, along with the evaluations of $h_0(z^4), \dots, h_3(z^4)$. Denote the mask values sent by the prover by $\{y_{j,s}\}_{j \in [0,11], s \in \{0,1\}}$, and the evaluations of $h_i(z^4)$ sent by the prover by $\{\hat{y}_i\}_{i \in [0,3]}$. For an honest prover, the value of each $\hat{y}_i$ equals $h_i(z^4)$, and the value of each $y_{j,s}$ equals to $f_j(zg^s)$ where j is the column of the corresponding cell and s is its row offset. The verifier may then calculate $h(z)$ in two ways: based on $h_0(z^4), \dots, h_3(z^4)$ (using $h(z) = \sum_{i=0}^{3} x^i h_i(z^4)$) and based on the mask values $y_{j,s}$. It verifies that the two results are identical.

It remains to show that the values sent by the prover in this phase are correct (i.e., indeed equal to the evaluation of the composition polynomial trace and the mask values of the point z), which will be done in the next section. This method of checking consistency between two polynomials by sampling a random point from a large domain is called *Domain Extension for Eliminating Pretenders (DEEP)*; see [BGKS20] for more details about it.

3.8 The DEEP Composition Polynomial

Verifying that the DEEP values sent by the prover are correct includes two parts:

1. Verifying that they are equal to the mask values of the point z.

2. Verifying that the trace is defined over $\mathbb{F}_p$, the native field in which Rescue operates (as opposed to the extension field $\mathbb{F}_p(\phi)$).

In the rest of this section we describe how these verifications are performed.

3.8.1 Verifying the Mask Values

In order to verify the values sent by the prover, we create a second set of constraints and then translate them to a problem of low degree testing, similar to the composition polynomial. For each mask value $y_{j,s}$, sent by the prover, we define the following constraint:

$$\frac{f_j(x) - y_{j,s}}{x - zg^s}$$

where j, s are the column and row offset of the corresponding cell. This rational function is a polynomial of degree $(\deg(f_j) - 1)$ if and only if $f_j(zg^s) = y_{j,s}$ for some polynomial $f_j(X)$ of degree $\deg(f_j)$.

Likewise, for each value $\hat{y}_i$ that the prover sent, we define the following constraint:

$$\frac{h_i(x) - \hat{y}_i}{x - z^4}$$

where i is the corresponding column index of the composition polynomial trace. This rational function is a polynomial of degree $(\deg(h_i(x)) - 1)$ if and only if $h_i(z^4) = \hat{y}_i$.

Denote the size of the mask by M_1, the mask values $\{y_{j,s}\}$ by $\{y_\ell\}_{\ell \in [0, M_1 - 1]}$ and the number of columns in the composition polynomial trace by M_2. The verifier samples $M = M_1 + M_2$ random elements from the extension field $\gamma_0 \ldots, \gamma_{M-1} \in \mathbb{F}_p(\phi)$. We define the *DEEP Composition Polynomial* as follows:

$$\sum_{\ell=0}^{M_1-1} \gamma_\ell \cdot \frac{f_{j_\ell}(x) - y_\ell}{x - zg^{s_\ell}} + \sum_{i=0}^{M_2-1} \gamma_{M_1+i} \cdot \frac{h_i(x) - \hat{y}_i}{x - z^4}$$

where j_ℓ and s_ℓ are the column and row offset corresponding to y_ℓ. This is a (random) linear combination of constraints of the form:

$$\frac{f(x) - y}{x - z}$$

where f is either a trace column polynomial or h_i polynomial. Thus, proving that this linear combination is of low degree implies proving the low degreeness of the trace column polynomials and that of the h_i polynomials, as well as that the DEEP values are correct.

3.8.2 Verifying the Trace Values

In order to verify that the trace is defined over $\mathbb{F}_p$, we add yet another set of constraints that assert that the coefficients of each column polynomial is indeed from $\mathbb{F}_p$ (rather than $\mathbb{F}_p(\phi)$).

Denote the conjugate of an element $x \in \mathbb{F}_p(\phi)$ by $\overline{x}$. Recall that the mask of the Rescue AIR consists of two consecutive rows – two elements in each column. We pick a single row[10], and for each column add the following constraint:

$$\frac{f_j(x) - \overline{y_{j,0}}}{x - \overline{z}}$$

This rational function is a polynomial of degree $(\deg(f_j) - 1)$ if and only if $f_j(\overline{z}) = \overline{y_{j,0}}$. Let m denote the number of columns in the trace. The verifier then chooses another m random extension field elements $\delta_0, \ldots, \delta_{m-1}$ and adds the following linear combination to the DEEP composition polynomial:

$$\sum_{j=0}^{m-1} \delta_i \cdot \frac{f_j(x) - \overline{y_{j,0}}}{x - \overline{z}}$$

For a column polynomial $f(x)$, if it holds for a random $z \in \mathbb{F}_p(\phi)$ that $f(\overline{z}) = \overline{f(z)}$, then (w.h.p) all the coefficients of $f(x)$ are from $\mathbb{F}_p$. Thus, proving that the new DEEP composition polynomial is of low degree, now also implies that the trace is defined over $\mathbb{F}_p$ as desired.

3.9 The FRI Protocol for Low Degree Testing

For low degree testing, we use an optimized variant of a protocol known as *FRI* (which stands for *Fast Reed-Solomon Interactive Oracle Proof of Proximity*) described in [BBHR18], with improved soundness bounds appearing in [BKS18, BGKS20, BCI$^+$20]. The optimizations we use are described in Section 3.11. The FRI protocol consists of two phases: a *commit phase* and a *query phase*.

[10] Since we verify that the coefficients of each column polynomial are from the appropriate field, adding the constraint for both mask values of each column is redundant. This is in contrast to verifying the mask values, where both mask values of each column are needed.

3.9.1 Commit Phase

In the basic FRI version, the prover splits the original DEEP composition polynomial of degree less than N, denoted here as $p_0(x)$, into two polynomials of degree less than $N/2$, call them $g_0(x)$ and $h_0(x)$, satisfying $p_0(x) = g_0(x^2) + x \cdot h_0(x^2)$. The verifier chooses a random value $\zeta_0 \in \mathbb{F}_p(\phi)$, sends it to the prover, and asks the prover to commit (using a Merkle commitment scheme) to the polynomial $p_1(x) = g_0(x) + \zeta_0 \cdot h_0(x)$. Note that $p_1(x)$ is of degree less than $N/2$. (Looking ahead, in our optimized FRI version the degree reduction from $p_0(x)$ to $p_1(x)$ is actually from N to $N/2^i$ for some $i \geq 1$, see Section 3.11.1.)

We then continue recursively by splitting $p_1(x)$ into $g_1(x)$ and $h_1(x)$, then constructing $p_2(x)$ with a random $\zeta_1 \in \mathbb{F}_p(\phi)$ chosen by the verifier, and so on. Each time, the degree of the polynomial is halved. Hence, after $\log_2(N)$ steps we are left with a constant polynomial, and the prover can simply send the constant value to the verifier.

For the above protocol to work, we need the property that for every v in the evaluation domain L, it holds that $-v$ is also in L, i.e., that L be closed under negation. Moreover, the commitment on $p_1(x)$ will not be over L but over $L^2 := \{x^2 : x \in L\}$. Since we iteratively apply the FRI step, L^2 also has to be closed under negation, and so on. These algebraic requirements are satisfied via our choice of a multiplicative[11] coset of size 2^k for integer k as our evaluation domain.

3.9.2 Query Phase

We now have to check that the prover did not cheat. Let L be the evaluation domain. The verifier samples a random $v \in L$ and queries $p_0(v)$ and $p_0(-v)$. These two values suffice to determine the values of $g_0(v^2)$ and $h_0(v^2)$, as can be seen by the following two linear equations in the two "variables" $g_0(v^2)$ and $h_0(v^2)$:

$$\begin{aligned} p_0(v) &= g_0(v^2) + v \cdot h_0(v^2) \\ p_0(-v) &= g_0(v^2) - v \cdot h_0(v^2) \end{aligned}$$

The verifier can solve this system of equations and deduce the values of $g_0(v^2)$ and $h_0(v^2)$. It follows that it can compute the value of $p_1(v^2)$ which is a linear combination of the two. Now the verifier queries $p_1(v^2)$ and makes sure that it is equal to the value computed above. This serves as an indication that the commitment to $p_1(x)$, which was sent by the prover in the commit phase, is indeed the correct one. The verifier may continue, by querying $p_1(-v^2)$ (recall that $(-v) \in L^2$ and that the commitment on $p_1(x)$ was given on L^2) and deduce from it $p_2(v^4)$.

The verifier continues in this way until it uses all these queries to finally deduce the value of $p_{\log(d)}(v^d)$. Recall that $p_{\log(d)}(x)$ is a constant polynomial whose constant value was sent by the prover in the commit phase, prior to choosing v. The verifier checks that the value sent by the prover is indeed equal to the value that the verifier computed from the queries to the previous functions.

All query responses received by the verifier also need to be checked for consistency with the Merkle commitments sent by the prover during the commit phase. Hence, the prover sends decommitment information (Merkle paths) together with these responses to allow the verifier to enforce this.

[11] Recall that Rescue's native field is $\mathbb{F}_p$ where $p = 2^{61} + 20 \cdot 2^{32} + 1$, thus $|\mathbb{F}_p^\times|$ is divisible by 2^{32}.

In addition, the verifier must also verify the values $p_0(v)$ and $p_0(-v)$ it received from the prover. Recall that the verifier does not maintain the DEEP composition polynomial p_0. For this, the prover also sends the values of the trace f_j and the composition polynomial trace h_j, induced by the DEEP composition polynomial, together with their decommitments. Then, the verifier checks the consistency of these values with the commitments on the traces, calculates the values of $p_0(v), p_0(-v)$ and checks consistency with the values sent by the prover.

In order to achieve the required soundness of the protocol, the query phase is repeated multiple times. In particular, to reach soundness error below $2^{-\lambda}$, and using a blowup factor of 2^k, we make a number λ/k of queries, using [BCI+20, Conjecture 7.3] (with $c_1 = c_2 = 1$ there), i.e., each query roughly contributes k "bits of soundness" to the protocol.

3.10 Transformation to Non-Interactive Protocol (the Fiat-Shamir heuristic)

So far, we described the proof generation process as an interactive protocol between a prover and a verifier. We now transform this interactive protocol into a non-interactive version, in which the prover generates a proof in the form of a file (or equivalent binary representation) and the verifier receives it to verify its correctness.

The fundamental idea behind this construction is that the prover simulates receiving the randomness from the verifier. This is done by the Fiat-Shamir heuristic applied to the transformation of [BSCS16] that converts interactive oracle proofs (IOPs) into non-interactive random oracle proofs (NIROPs). We extract randomness from a hash function that is applied to prior data sent by the prover (and appended to the proof). We initialize the seed by hashing a description of the statement – "Rescue hash chain", and the public input, which are known to both the prover and the verifier.

Recall that the AIR for the Rescue hash chain corresponds to the claim stated by Eq. (1). We use the `chain_length` ($|w| - 1$) and the four field elements of `output` as the seed to the hash chain.

3.11 Proof Length Optimizations

We employ several optimization techniques in order to reduce the proof size. These techniques are described in this section.

3.11.1 Skipping FRI Layers

Instead of committing to each of the FRI layers in the commitment phase of the FRI protocol, the prover can skip layers and commit only to a subset of them. Doing that, the number of Merkle trees is reduced, which means that the prover has less decommitment paths to send to the verifier. There is a trade off, though. If, for example, the prover commits only to every third layer, in order to answer a query, it needs to decommit to 8 elements of the first layer (instead of only 2 in the standard case). This fact is taken into account in the commitment phase. It packs together neighbor elements in each leaf of the Merkle tree. For more details see Section 3.5. Thus, the cost of skipping layers is sending more field elements, but not more authentication paths.

Skipping FRI layers can be configured using the `fri_step_list` parameter. The FRI reduction in the ith layer will be $2^{\texttt{fri\_step\_list[i]}}$ and the total reduction factor will be $2^{\sum_i \texttt{fri\_step\_list[i]}}$.

3.11.2 FRI Last Layer

Another FRI optimization used to reduce the proof size, is to terminate the FRI protocol earlier than when the last layer reaches a constant value. In such a case, instead of having the prover send only the constant value of the last layer as a commitment, the prover sends the coefficients of the polynomial representing the last layer. This allows the verifier to complete the protocol as before, without the need for commitments (and sending decommitments for field elements in following layers). The degree bound for early termination of the FRI protocol can be configured using the `last_layer_degree_bound` parameter.

3.11.3 Grinding

As mentioned in Section 3.9, every query adds a certain number of bits to the security (soundness) of the proof. However, it also implies sending more decommitments which increases the proof size. One mechanism to reduce the need for many queries is to increase the cost of generating a false proof by a malicious prover. We achieve this by adding to the above protocol a requirement that following all the commitments made by the prover, the prover must find a 64 bit nonce that when hashed together with the state of the hash chain, results in a required number of leading zeros. The number of the leading zeros defines a certain amount of work that the prover must perform before generating the randomness representing the queries. As a result, a malicious prover that attempts to generate favorable queries will need to repeat the grinding process every time that a commitment is changed. On the other hand, an honest prover only needs to perform the grinding process once.

This is similar to the grinding performed on many block-chains. The nonce found by the prover is sent to the verifier as part of the proof and in turn the verifier checks its consistency with the state of the hash chain by running the hash function once.

The required number of leading zeros is configured by the `proof_of_work_bits` parameter.

4 Measurements and Benchmarks

To estimate the concrete efficiency of our system, we ran experiments measuring the proving and verification time, the maximal memory consumption, and the generated proofs size, for different numbers of hash invocations, security levels and blowup factors. All the experiments, for both the prover and the verifier, were run on the same machine with the following specifications:

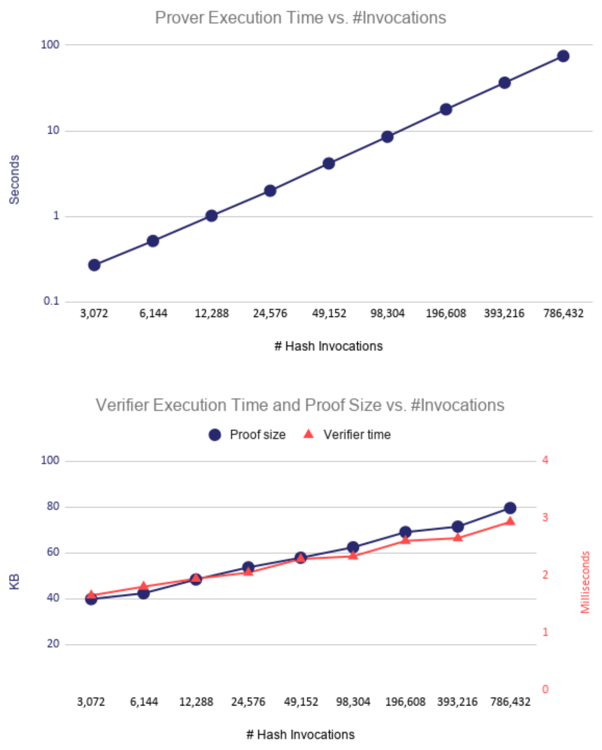
1. Operating-System: Linux 5.3.0-51-generic x86_64.
2. CPU: Intel(R) Core(TM) i7-7700K @ 4.20GHz (4 cores, 2 threads per core).
3. RAM: 16GB DDR4 (8GB $\times$ 2, Speed: 2667 MHz)

We note that while the prover uses multi-threading, in all of the experiments the verifier was restricted to utilize only a single thread. In addition, measurements corresponding to 80 bits of security are done using BLAKE2s with a digest size of 20 bytes (160 bits) as the underlying hash function, whereas measurements corresponding to 100 bits of security used BLAKE2s with a digest size of 25 bytes (200 bits).

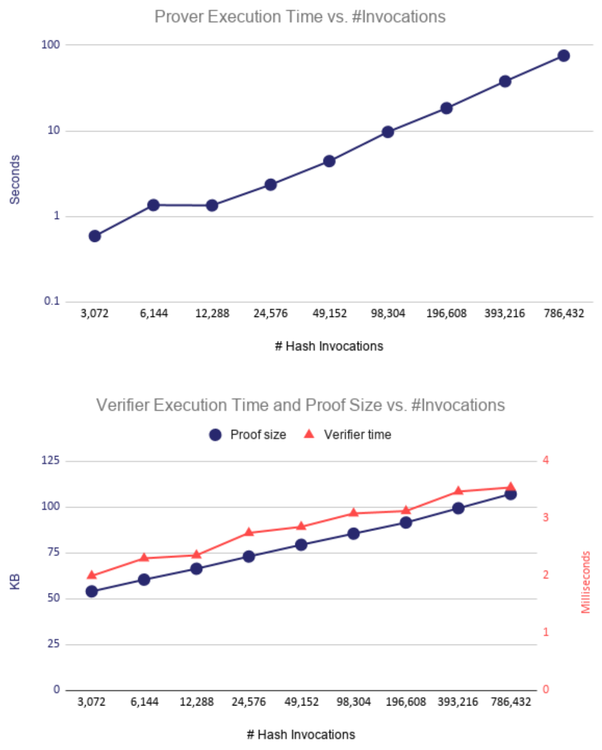
4.1 Prover/Verifier Time and Proof Size vs. Number of Hash Invocations

In Figure 5 we present measurements of proving and verification time as well as proof size, as a function of the number of Rescue hash invocations. Recall that we fit batches of 3 hashes into 32 rows in the Rescue trace, see Section 3.1 for more details. Therefore, the number of hash invocations, also referred to as the *chain length*, is divisible by 3. Since the actual traces we produce must have a length that is a power of 2, we use 3×2^i, for $i \in [10, 18]$, as the number of hash invocations for our measurements.

Since the values for the x-axis grow exponentially (3×2^i), and the y-axis is on a logarithmic scale, the measurements in the top graphs in Figure 5 match our theoretical predictions that the amount of time spent by the prover scales nearly-linearly in the number of hash invocations. Whereas verification time and proof size scale poly-logarithmically in the number of hash invocations.



(a) 80 bits of security.



(b) 100 bits of security.

Figure 5: Verification time and proof size (bottom graphs) and proving time (top graphs) as a function of the number of Rescue hash invocations, measured for 80 bit security (left side) and 100 bits of security (right side). In the top graphs, the prover time is measured in seconds, while in the bottom graphs, the verifier time is measured in milliseconds.

4.2 Prover/Verifier Time and Proof Size vs. Blowup Factor

Recall that the blowup factor is the ratio between the size of the evaluation domain and the trace evaluation domain, see Section 3.4 for more details. In Fig. 6 we present measurements of proving/verifying time and proof size as a function of the blowup factor. The measurements are done with 80 bits of security, a chain length of size roughly 98K (98, 304, to be precise) and blowup factors 4, 8 and 16.

It is evident from Figure 6 that the blowup factor enables shifting computation overheads between the prover and the verifier. For fixed security level, increasing the blowup factor increases prover time (blue bars) but reduces proof size (red bars) and verification time (green bars). Notice that none of the changes are linear, but rather sub-linear. I.e., as the blowup factor doubles ($4 \rightarrow 8$ and $8 \rightarrow 16$) proving time increases only by $\approx 50\%$ while proof size and verification time decrease by $\approx 25\%$.

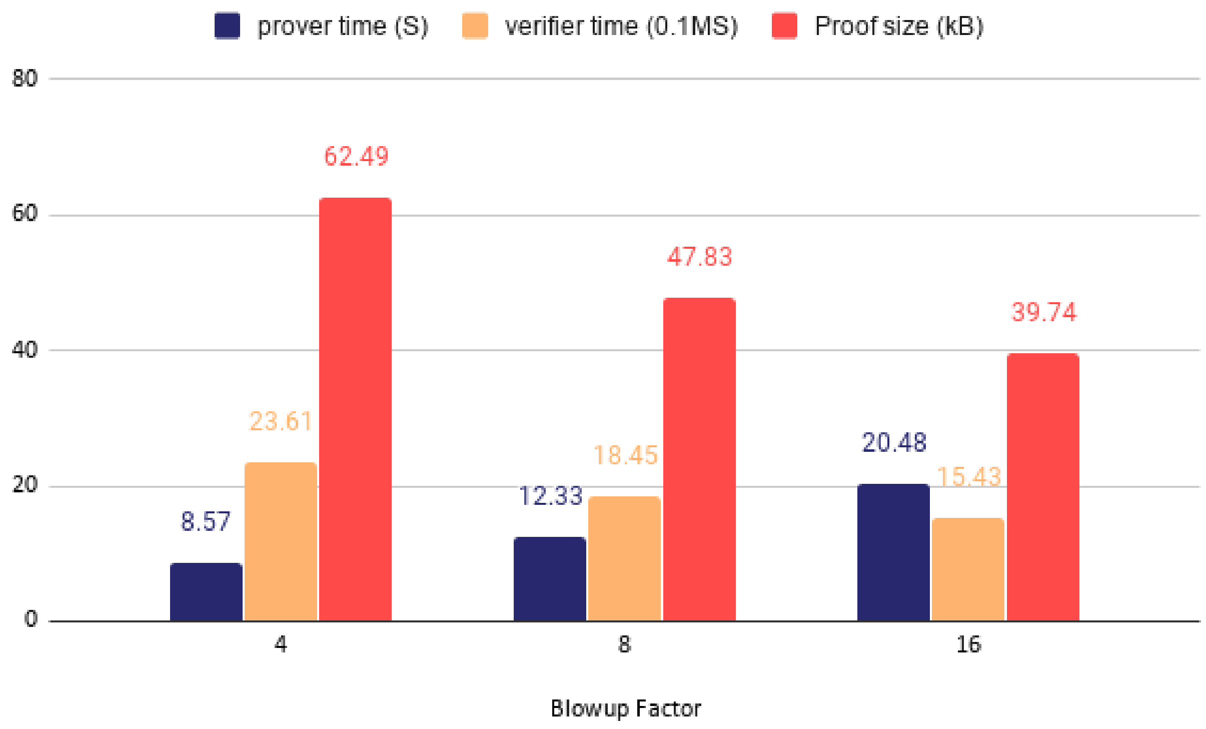
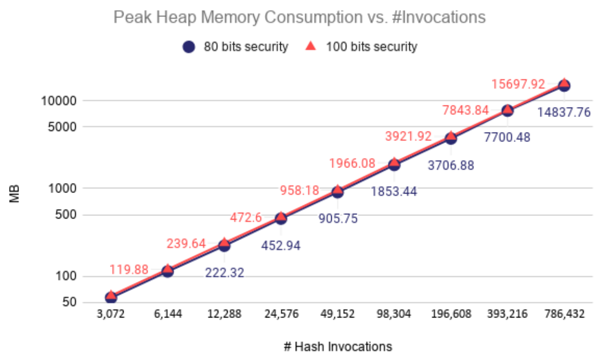


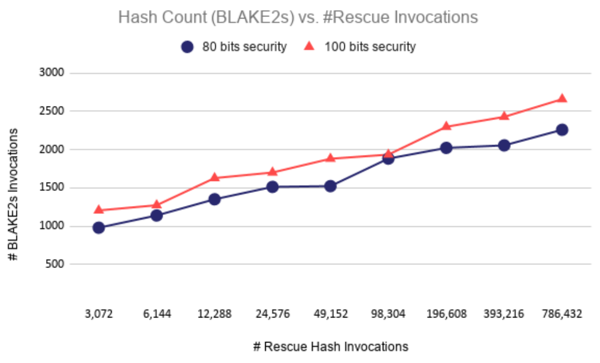
Figure 6: Proving/verification time and proof size as a function of the blowup factor. Measurements are done with 80 bits of security and a chain length of size 98304.

4.3 Memory Consumption and Recursive Proof Composition

Memory Figure 7a depicts the prover's peak memory (RAM) consumption as a function of the number of Rescue hash invocations. It is readily apparent from the figure that: (i) memory consumption measured for 80 bits of security and 100 bits of security are fairly similar, (ii) prover memory requirements are satisfied by a machine with standard specifications, even for chain length nearing one million hashes, and (iii) as long as the computation fits inside the machine's available RAM, memory consumption matches the theoretical prediction of linear growth with the number of Rescue hash invocations. However, once memory consumption becomes larger than the available



(a) Prover peak memory consumption.



(b) Hash (BLAKE2s) count.

Figure 7: Prover peak memory consumption and the number of underlying hash (BLAKE2s) invocations as a function of Rescue hash invocations.

RAM, a deterioration in performance is expected (not discussed in the scope of this work). We stress that memory consumption need not scale linearly with chain length but rather, memory and proving time can be traded off, one against the other.

Recursion Any universal and succinct proof/argument of knowledge system (in particular, STARKs) can be used to incrementally verify computation [Val08, BCCT13]. This means that a computation may generate a proof that attests to the correctness of a previous instance of that computation, a concept known informally as “recursive proof composition”, or, in our case, “recursive STARKs”. In other words, a recursive STARK prover would generate a proof for a statement saying the state of a system can be moved from x_i to x_{i+1} because the prover has verified a (recursive) proof attesting to the computational integrity of x_i and has faithfully executed the computation on the state x_i, reaching the new state x_{i+1}.

While the impact of recursion depth is a delicate matter (cf. [Val08]), it is clear that a major part of the prover’s computation in this case is focused on verifying a STARK proof. This requires verifying all the hashes in the decommitment paths of a previous STARK. For instance, if the statement proved recursively roughly matches our Eq. (1), the size of the recursive computation (i.e., the AIR and execution trace) would likely be dominated by the need to verify the correctness of the hash decommitments.

In Figure 7b we present the number of hash invocations used in a proof of Eq. (1) for varying chain length, ranging between 3K and 786K hashes. Crucially, the number of hashes involved in decommitments of these statements does not reach even the lower end and ranges between ≈ 1000 and ≈ 2700. This suggests that for simple computational statements proved via STARKs that use the Rescue hash (instead of Blake2s) to commit to proof oracles, and for secure recursion depth (as discussed in [Val08]), recursive STARKs could be efficiently constructed.

References

[AAB$^+$19] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Efficient symmetric primitives for advanced cryptographic protocols (A marvellous contribution). *IACR Cryptology ePrint Archive*, 2019:426, 2019.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Proceedings of the 39th Annual International Cryptology Conference*, CRYPTO '19, pages 733–764, 2019.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 111–120. ACM, 2013.

[BCI$^+$20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. *Electronic Colloquium on Computational Complexity (ECCC)*, 27:83, 2020.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC (B2)*, volume 9986 of *Lecture Notes in Computer Science*, pages 31–60, 2016.

[BGKS20] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: sampling outside the box improves soundness. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 5:1–5:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[BKS18] Eli Ben-Sasson, Swastik Kopparty, and Shubhangi Saraf. Worst-case to average case reductions for the distance to a code. In Rocco A. Servedio, editor, *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 24:1–24:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[BSCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam Smith, editors, *Theory of Cryptography*, pages 31–60, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. Preliminary version appeared in STOC '85.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[RRR16] Omer Reingold, Ron Rothblum, and Guy Rothblum. Constant-round interactive proofs for delegating computation. In *Proceedings of the 48th ACM Symposium on the Theory of Computing*, STOC '16, pages 49–62, 2016.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.