A Walk-Through of a Simple zk-STARK Proof *

Aleksander Berentsen† Jeremias Lenzi‡ Remo Nyffenegger§

December, 2022

Abstract

This article serves as an extension to the introductory article on zero-knowledge proofs (ZKP) by Berentsen, Lenzi and Nyffenegger (2022). We provide one specific example of a zk-STARK and discuss all mathematical steps needed. The goal is to make the example accessible and intuitive. Furthermore, this article comes with an accompanying Python notebook[1] that will let the reader execute a numerical example provided in the article.

*Disclaimer: This article relies heavily on Ben-Sasson et al. (2018), StarkWare (2019*d*), StarkWare (2019*a*), StarkWare (2019*b*), StarkWare (2019*c*), StarkWare (2020*a*), StarkWare (2020*b*), StarkWare (2020*c*), StarkWare (2020*d*), Buterin (2017*b*), Buterin (2017*c*) and Buterin (2017*a*).

†Aleksander Berentsen is a professor of economic theory at the University of Basel and a research fellow at the Federal Reserve Bank of St. Louis. E-Mail: aleksander.berentsen@unibas.ch

‡Jeremias Lenzi is a PhD student at the University of Basel.
E-Mail: jeremias.lenzi@unibas.ch

§Remo Nyffenegger is a PhD student at the University of Basel and a research assistant at the Center for Innovative Finance of the University of Basel. E-Mail: remo.nyffenegger@gmail.com

[1]The Python notebook can be found here: `https://github.com/remonyffenegger/stark-tutorial.git`

Introduction

This article serves as an extension to the introductory article on zero-knowledge proofs (ZKP) by Berentsen, Lenzi and Nyffenegger (2022). We provide one specific example of a zk-STARK and discuss all steps needed. We decide to describe a zk-STARK because we believe that it is possible to understand the easy example with a basic math knowledge. All mathematical concepts that might be new will be explained throughout the article. The goal is to make the example accessible and intuitive. For the sake of clarity, we simplify certain concepts. Not everything we discuss does generalize to more complex problems. Furthermore, we do not focus on efficiency of the proof. Nevertheless the reader should gain a lot of insights on how a ZKP works. Furthermore, this article comes with an accompanying Python notebook (link) that will let the reader execute a numerical example in the article that is described in the green boxes.

Numerical Example:

These boxes provide a numerical example.

Additionally, we provide blue boxes including definitions that explain all mathematical concepts that might be new to some readers.

Definition:

These boxes include mathematical definitions and explanations.

The proof we show is interactive. However, it can be transformed into a non-interactive one as described in Berentsen, Lenzi and Nyffenegger (2022). Instead of random queries by the verifier, the prover can apply the Fiat-Shamir heuristic, i.e. use a cryptographic hash function's output of a transcript of the protocol up to this point to get pseudorandom queries. In this article we introduce a crucial ingredient of ZKPs that is not talked about in Berentsen, Lenzi and Nyffenegger (2022): polynomials. There are two main purposes. On the one hand they allow to make a proof zero knowledge. On the other hand, they can make ZKPs

succinct, i.e. the verifier is convinced of the proof's correctness after just a few queries. If a prover cheated, he would have to cheat almost everywhere when using polynomials. This allows to catch a malicious prover with only a few queries.

The article is composed of three main parts. First, we will define the problem and the statement to be proven. Second, we will discuss the arithmetization of the statement, i.e. represent it as an algebraic problem. This involves working with polynomials and transforming them in a certain way. Third, we check whether a polynomial is of low degree. If it is, then the initial statement is true with high probability. But let's do this step by step.

1 Definition of the Problem

1.1 CI Statement, Trace and Polynomial Constraints

First, we define a computational integrity (CI) statement (see definition 1) we want to prove.

Definition 1: Computational Integrity

Computational integrity (CI) means that the output of a certain computation is correct. (Ben-Sasson, 2019)

In our example, this is:

CI statement: *The prover has a sequence A of N integers, all of which are either 0 or 1 (boolean).*

The statement is true if all elements $A_i \in A$ are either 0 or 1. The goal is to prove that this statement is true with a sufficiently high probability without revealing A, i.e. without revealing which element of A is 0 or 1. Henceforth, we call A the trace.

Numerical Example: Trace

In our example, we fix $N = 4$ such that

- $A^{true} = [1, 0, 1, 1]$: satisfies CI statement.
- $A^{false} = [2, 0, 1, 1]$: does not satisfy CI statement.

First, the prover and the verifier agree on certain conditions that have to hold if the statement is correct. In our example, these conditions can be written as the following polynomial (see definition 2) constraints

$$A_i(A_i - 1) = A_i^2 - A_i = 0 \quad \forall i = 0, 1, ..., N - 1 \tag{1}$$

Note that this only holds if A_i is either 1 or 0. The equations in (1) being true thus implies that the CI statement above is true as well. To keep the problem simple we choose N to be a power of 2.[2]

Definition 2: Polynomials and Polynomial Degree

A **polynomial** is a mathematical expression of a positive number of algebraic terms linked by addition that consist of a constant a (positive or negative) multiplied by one or several variables that are raised to a non-negative integral power. Examples:

- $f(x) = 2x^4 - 3x^2 + x - 2$
- $g(x, y, z) = 3xy^2z^4 + 2yz + 3z - 4$

In the case of a single-variable polynomial this can be generalized as $\sum_{i=0}^{d} a_i X^i$.

The **degree of a polynomial** d is defined by the highest power. In case of several variables, it is defined by the highest sum of the powers within an algebraic term. $f(x)$ has a degree of 4, which we write as $deg(f) = 4$.

[2]This is to ensure that we find a subgroup in section 2.1 given our definition of the finite field F in section 1.2. If N is a power of 2, it is a divisor of the size of the multiplicative group of F which ensures that a subgroup exists.

To find the degree of $g(x, y, z)$ we rewrite it as

$$g(x, y, z) = 3x^1y^2z^4 + 2x^0y^1z^1 + 3x^0y^0z^1 - 4x^0y^0z^0$$

The sum of the powers of the four terms is $7, 3, 2, 0$ respectively. Since 7 is the largest number we have $deg(g) = 7$.

Given a polynomial multiplication or division[a] of two polynomials it is straightforward to calculate the degree of the resulting polynomial. For illustration purposes it is enough to look at two polynomials with only one term: $h(x) = x^7, \quad l(x) = x^4$. For multiplication, i.e. $h(x) \cdot l(x)$, we just add up the degrees of the two polynomials, i.e. $deg(h)+deg(l) = 7+4 = 11$. We can check that this holds

$$h(x) \cdot l(x) = x^7 \cdot x^4 = x^{11}.$$

For division, i.e. $h(x)/l(x)$, we subtract the degree of the polynomial in the numerator by the degree of the polynomial in the denominator, i.e. $deg(h) - deg(l) = 7 - 4 = 3$. Again we check whether this holds

$$h(x)/l(x) = x^7/x^4 = x^3.$$

[a] Assuming division is possible.

1.2 Finite Field

Before creating the proof, the prover and the verifier need to define the field (see definition 3) they are working with. As it is standard in computer science problems, we will not work in the Euclidean space but with a finite field.[3] The most common way to work with a finite field is by using the modulo operator

[3] It is inefficient for a computer to work with very large numbers. Thus it is preferred whenever possible to work in finite fields that keep the numbers the computer has to work with within a certain range.

with the arithmetic modulo M (see definition 4) . We could either work with a binary field or a prime field. However, prime fields are easier to understand. In a prime field, M is a prime number and the modular arithmetic keeps many of the properties (e.g. addition, multiplication) from regular Euclidean arithmetic. We denote the finite (prime) field we work with as F. Furthermore, we want the finite field to be of size $M = 2^n + 1$ for $n \in \{1, 2, 4, 8, \ldots\}$.[4] In realistic applications the field size is very large and much larger than the size of the trace.

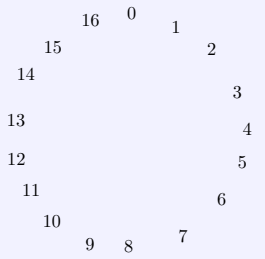
Definition 3: Fields and Finite Fields

A **field** is a set on which addition, subtraction, multiplication and division are defined and certain basic rules are satisfied.

A **finite field** or (Galois field) is a finite set on which the operations of multiplication and addition satisfy associativity, commutativity and distributivity. Also, in the finite set there exist additive and multiplicative inverses alongside with an additive and a multiplicative identity element - this is normally the 0 and 1 respectively.

Definition 4: Modular Arithmetic

Modular Arithmetic is often used when working with a finite field. It can be best illustrated by using a "clock". Assume we work in a prime field where the arithmetic modulo is $M = 17$. The finite field contains the set $\{0, 1, ..., 16\}$. Whenever an operation yields a result that is larger than 16, we follow the clock clockwise, starting with 0 again. Examples:



$$16 + 2 \mod 17 = 1, \quad 12 + 7 \mod 17 = 2, \quad 6 \cdot 7 \mod 17 = 8$$

[4]Only if n is a power of 2 will $M = 2^n + 1$ be a prime. We could relax this to other prime numbers M if $M - 1$ is a cyclic group (see definition 5) and N is a divisor of $M - 1$. However, this would complicate the low degree testing in section 3.3.

It is equivalent to write the numbers in their negative realization, e.g.

$$-5 = 17 - 5 = 12, \quad -10 = 17 - 10 = 7, \quad -2 = 17 - 2 = 15$$

Software implementations of finite fields often use both the positive and the negative realization.

Furthermore, we need to define how division works. Each element $a \in \{0, 1, ..., 16\}$ in the field has a multiplicative inverse b that is defined by $a * b = 1$.[a] Calculating x/a is then equivalent to $x * b$. In the field with $M = 17$, the multiplicative inverses are

$$[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]$$

Some examples:

$$5/4 = 5 \cdot 13 = 14, \quad 8/12 = 8 \cdot 10 = 12, \quad 15/16 = 15 \cdot 16 = 2$$

[a] Except for the 0 element. Since division by 0 is not possible, there is no multiplicative inverse.

Numerical Example: Finite Field

To illustrate our example properly, we choose a very small finite field with $M = 2^4 + 1 = 17$.[a] The finite field then is

$$F = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$$

[a] Finite fields that are used in real applications are much larger. In the zk-STARK paper, Ben-Sasson et al. (2018) mention a binary field of size 2^{64}.

2 Arithmetization

After having defined the problem and the field we work with, we now get to the arithmetization part. This means that we want to transform the problem into an

algebraic problem to reason in the natural numbers.

2.1 Evaluating Trace as a Polynomial

We want to think of the trace as the evaluation of some polynomial f, i.e. for some $X \subset F$ where $|X| = N$ we have that

$$f(X_i) = A_i \quad \forall i = 0, 1, ..., N-1.$$

This states that the outputs of the polynomial f relate to the trace. To choose X, we define a subgroup G of size N of the multiplicative group $F_{/\{0\}}$, where $F_{/\{0\}}$ is the finite field without the zero. Furthermore, we define a generator g of G (see definition 5).

Definition 5: Group, Subgroup, Generator and Cyclic Group

A **group** is a set with a binary operation that satisfies associativity, existence of an identity element and existence of the inverse (under this operation) for each element of the group.

A subset of the elements of a group is called a **subgroup** if it is a group under the same binary operation.

A **generator** g is an element of the subgroup such that all powers of g span the entire subgroup.

A **cyclic group** is a group that is generated by a single element, i.e. the generator.

Numerical Example: Subgroup G and Generator g

Since $N = 4$, the size of the subgroup $G \subset F$ is 4. We need to find a $g \in F_{/\{0\}}$ for which the powers of g uniquely yield the elements of G. In our example we find that $g = 4$ and $G = \{1, 4, 13, 16\}$. The table below illustrates that the powers of g span the entire subgroup G. The second row depicts numbers in the Euclidean space whereas the third row shows

the realization in $F_{/\{0\}}$.

$g = 4$	g^0	g^1	g^2	g^3	g^4	g^5	g^6	g^7	g^8	...
$\mathbb{R}$	1	4	16	64	256	1′024	4′096	16′384	65′536	...
$F_{/\{0\}}$	1	4	16	13	1	4	16	13	1	...

The prover now maps the elements of the subgroup G defined by the generator g to the sequence A using the polynomial f

$$f(g^i) = A_i \quad \forall i = 0, 1, ..., N-1. \tag{2}$$

This states that f maps the i'th element of the subgroup G to the i'th element of the sequence A. The next step is to find f. Intuitively, if we think about the polynomial in the Cartesian plane, the elements of G are the x-values and the elements of A are the corresponding y-values. The goal is to find a polynomial that fits through these points. Generally, there are many polynomials which could be considered. However, there is only one low-degree polynomial (see definition 6). And for our use case it is crucial that we choose the low degree one. This means that $deg(f) < N$. If all N points are distinct, this becomes $deg(f) = N-1$. We can find this polynomial by using Lagrange interpolation or the fast Fourier transformation, where the latter is more efficient for more complex problems than ours (but also more complicated).

Definition 6: Unisolvence Theorem and Lagrange Polynomial

The **Unisolvence theorem** states that given a set of N data points (x_i,y_i) where all x_i are unique and a polynomial f is defined by

$$f(x_i) = y_i, i = 0, .., N$$

then

$$\exists! f \text{ s.t. } deg(f) < N.$$

If $deg(f) < N$, then we call f a low degree polynomial or also **Lagrange polynomial** since the Lagrange interpolation is an algorithm to find this low-degree polynomial. If all points y_i are distinct, we get $deg(f) = N - 1$. If they are not distinct, we can have $deg(f) << N$. Using the theorem, it is straightforward that a polynomial of degree d can be uniquely determined by $d + 1$ data points.

To gain some intuition, let's go through the following example. If you have only one point $(x, y) = (1, 1)$ s.t. $N = 1$, you can fit a vertical line, i.e. the polynomial $f(x) = 1$ of degree 0. It holds that $deg(f) = N - 1$. If you have two points $(x, y) = (\{1, 2\}, \{1, 4\})$ s.t. $N = 2$ you can fit a function of the form $y = mx + c$, i.e. the polynomial $f = 3x - 2$ of degree 1. Again it holds that $deg(f) = N - 1$. However if the two points have the same y-coordinate in the $N = 2$ case, i.e. $(x, y) = (\{1, 2\}, \{1, 1\})$, the polynomial $f(x) = 1$ of degree 0 can fit these two points. Thus, it follows that $deg(f) << N$ is possible. Furthermore, there are polynomials that are of higher degree and also fit through these two points. For example the quadratic polynomial $f = x^2$ of degree 2 also fits through $(x, y) = (\{1, 2\}, \{1, 4\})$. However, this violates the definition of a low degree polynomial since $deg(f) \not< N$.

Numerical Example: Lagrange Interpolation

Given $A = A^{true} = [1, 0, 1, 1]$, the prover needs to find a polynomial

$$f : G \to F$$

$$s.t.$$

$$f(g^0) = f(1) = A_0 = 1$$

$$f(g^1) = f(4) = A_1 = 0$$

$$f(g^2) = f(13) = A_2 = 1$$

$$f(g^3) = f(16) = A_3 = 1.$$

Applying the Lagrange interpolation we find

$$f(x) = -x^3 - 4x^2 + x + 5 \tag{3}$$

which is equal to

$$f(x) = 16x^3 + 13x^2 + x + 5.$$

2.2 Evaluate the Polynomial on a Larger Domain

So far we have provided a statement to be proven, set up the mathematical constraints that need to hold if the statement is true, defined the field that we work in and found a polynomial that relates the subgroup G to the elements in A. Remember that we do that because we want to be able to check the CI statement mathematically with only a few queries.

The verifier could now check whether the outputs of f are indeed only 1's and 0's. However, this would not help us with zero knowledge and succinctness. Hence, the next step is to extend f to a larger domain.

To do that we evaluate f not only in G but on a larger domain L where $G \subset L \subseteq F_{/\{0\}}$ such that $f : L \to F$. By doing so we create a Reed-Solomon error correction code (see definition 7). For reasons we will see later, the size of L should be a power of 2. Both the prover and the verifier agree upfront what L is. Evaluating a polynomial in a larger domain is a very common thing with various applications in engineering, computer science, cryptography and other areas.

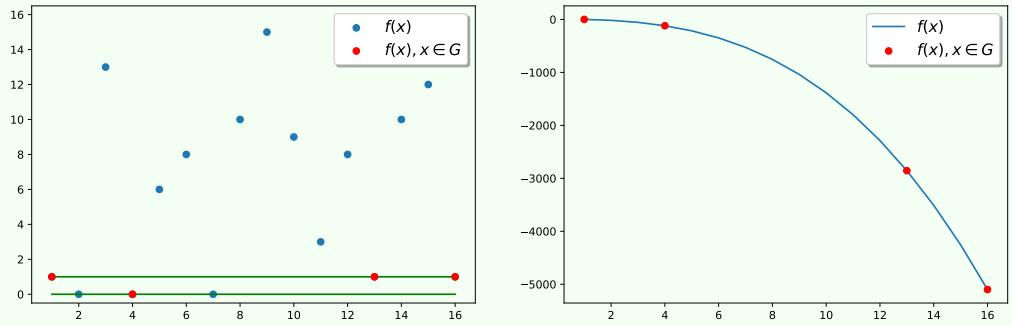
Definition 7: Reed-Solomon Error Correction Code

A **Reed-Solomon Error Correction Code** is an error correction code (ECC) which is mostly used for controlling errors in data or communication channels. One motivation is that if you store data somewhere (e.g. a CD or a DVD) you still want to be able to read the data even if some bits will be damaged. Or if you send a message over the internet, the data you send from your phone to the next antenna may get slightly corrupted when passing a thick wall and you want to ensure that the message is still transmitted properly. Using an ECC can handle these problems by adding some redundancy to the data which allows to detect whether the data is corrupted and can also correct the data up to a certain number of errors.
In the Reed-Solom ECC this is done by encoding the data into polynomials. Conceptually this works as following. Remember from the Unisolvence theorem that a set X of N distinct points (think of them as data points we store) can be uniquely defined by a polynomial f of degree $N-1$. If we extend f to a larger domain, such that we e.g. evaluate the polynomial over a domain of size $2N$, we can loose up to 50% of any points and are still able to derive the original polynomial with degree $N-1$ from the N remaining points. As a consequence, we can also restore the original set X.

Numerical Example: Extend to Larger Domain

For the sake of clarity and simplicity we choose this larger domain to be equal to the multiplicative group of the finite field, i.e. $L = F_{/\{0\}}$. However, it is important to note that in more complex problems, $L << F_{/\{0\}}$. If L

is too big, the computations become inefficient, we will see later why this is. In the figure below we illustrate the polynomial defined in equation (3). The graph on the left shows the polynomial f in the finite field. The red dots are the evaluations of G which match the trace A. The graph on the right shows the polynomial in the Euclidean space for illustration purposes.



2.3 Apply Constraints to Polynomial

Remember that in (1) we defined a set of constraints which hold iff the trace contains only 0's and 1's or in other words iff the CI statement is true. Since the polynomial f relates directly to the trace, the prover can now combine the constraints in (1) with the polynomial defined in (2)

$$A_i^2 - A_i = 0 \Rightarrow f(g^i)^2 - f(g^i) = 0 \quad \forall i = 0, 1, ..., N-1$$

and define this expression as the constraint polynomial $c : L \to F$

$$c(x) = f(x)^2 - f(x). \tag{4}$$

For all elements of the subgroup G, the polynomial c is 0, i.e.

$$c(x) = f(x)^2 - f(x) = 0 \quad \forall x \in G. \tag{5}$$

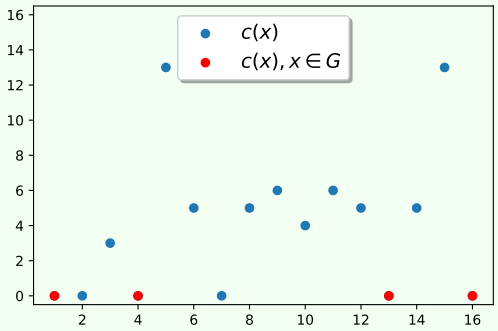
In other words, iff the CI statement is true, then c has roots for all elements of G. Note that c can also have roots for $x \notin G$.

Numerical Example: Get Constraint Polynomial c(x)

Using equation (3) we get

$$\begin{aligned} c(x) =& f(x)^2 - f(x) \\ =& x^6 + 8x^5 - 3x^4 - x^2 - 8x + 3. \end{aligned} \tag{6}$$

c is illustrated in the figure below. The red dots show that the polynomial has roots (equals 0) for all elements of G.



2.4 Create the Composition Polynomial

In the next step, the prover transforms $c(x)$ using the properties of roots of polynomials (see definition 8) to get another polynomial $p(x)$

$$p(x) = \frac{c(x)}{\prod_{i=0}^{N-1}(x - g^i)} \quad \forall x \in F \text{ and } x \notin \{g^0, g^1, \ldots, g^{N-1}\}. \tag{7}$$

with degree $deg(p) = deg(c) - N$. We henceforth call $p(x)$ the composition polynomial.[5] Definition 8 states that $p(x)$ is only a polynomial if all elements

[5]Note that this is a simplification from the real STARK implementation. Here we only have one constraint defined in equation (1). In more complex problems there is not only one constraint but several. Thus, there are several $p(x)$ polynomials. In that case the prover creates a linear combination of all $p(x)$ which is then called the composition polynomial. For example if there are three constraints and consequently we get three low degree polynomials $p_1(x), p_2(x), p_3(x)$, then what we call the composition polynomial is $P(x) = \alpha_1 p_1(x) + \alpha_2 p_2(x) +$

of G are roots of $c(x)$. And as we discussed earlier, only if the CI statement is correct, then all elements in G are roots of $c(x)$ (see section 4 to gain some intuition on what happens if the CI statement is not correct). Therefore, checking that $p(x)$ is a polynomial of degree at most $deg(c) - N$ is equivalent to stating that the CI statement is correct.

Definition 8: Roots of Polynomials

If a set $\{g^0, g^1, \ldots, g^{N-1}\}$ are N roots of a polynomial $c(x)$[a], then there exists another polynomial $p(x)$ that can be defined as

$$p(x) = \frac{c(x)}{\prod_{i=0}^{N-1}(x - g^i)} \quad \forall x \in F \text{ and } x \notin \{g^0, g^1, \ldots, g^{N-1}\}$$

whose degree is $deg(p) = deg(c) - N$.[b] In other words, we can define a polynomial $p(x)$ that agrees with the RHS for all $x \in F$ except for $x \in \{g^0, g^1, \ldots, g^{N-1}\}$.[c] The important part is that $p(x)$ only is a polynomial, if $\{g^0, g^1, \ldots, g^{N-1}\}$ really are the roots of $c(x)$.[d]

[a] There might be other roots as well.

[b] x in the denominator is multiplied N times. Thus the degree of the polynomial in the denominator will be N. Along the lines of definition 2 it is straightforward why $deg(p) = deg(c) - N$.

[c] For $x \in \{g^0, g^1, \ldots, g^{N-1}\}$, the denominator is 0.

[d] To gain some intuition assume that $c(x) = (x-1)(x-2)(x-3)$ s.t. the roots are $z = \{1, 2, 3\}$. If we apply the equation above and only include roots in the denominator the result will be another polynomial, e.g. $p(x) = \frac{(x-1)(x-2)(x-3)}{(x-1)(x-2)} = x - 3$ or $p(x) = \frac{(x-1)(x-2)(x-3)}{(x-3)} = (x-1)(x-2)$. However, if we do not include a root in the denominator, $p(x)$ will not be another polynomial, e.g. $p(x) = \frac{(x-1)(x-2)(x-3)}{(x-7)}$.

Note that the verifier has all information to derive the highest degree that still passes as low degree. N is known and $f(x)$ must be of degree at most $N - 1$. Additionally the verifier knows the constraint in equation (1) and can derive that the degree of $c(x)$ will be at most $2deg(f) = 2(N-1)$. Using equation (7) he can calculate that the degree of $p(x)$ is at most $deg(c) - N = 2(N-1) - N = N - 2$.

Furthermore, we can think more about the denominator of the RHS in (7). Since we chose g in a way that its powers form a subgroup, we can apply the following

$\alpha_3 p_3(x)$, where α are pseudorandom numbers.

formula[6]

$$\prod_{i=0}^{N-1}(x - g^i) = x^N - 1 = u(x). \tag{8}$$

On first sight, this might seem like a minor step. However, the RHS in (8) is considerably less computationally expensive (running time (see definition 9) is logarithmic in N) compared to the LHS (running time is linear in N). This simplifies the expression in (7) to

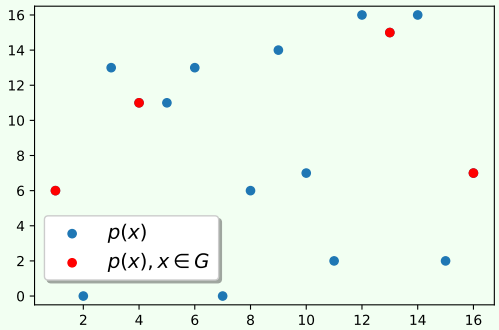
$$p(x) = \frac{c(x)}{u(x)} = \frac{f(x)^2 - f(x)}{x^N - 1} \quad \forall x \in F \text{ and } x \notin \{g^0, g^1, \ldots, g^{N-1}\} \tag{9}$$

Numerical Example: Get Composition Polynomial

Using equation (9) we get

$$\begin{aligned} p(x) =& \frac{x^6 + 8x^5 - 3x^4 - x^2 - 8x + 3}{x^4 - 1} \\ p(x) =& x^2 + 8x - 3. \end{aligned}$$

It is straightforward that the upper bound of the degree on $p(x)$, i.e. $deg(p) = N - 2 = 2$, holds.



[6]You can check in the numerical example that this holds. By multiplying $(x-1)(x-4)(x-13)(x-16)$ you will get $(x^4 - 1)$.

Definition 9: Runtime and Big O Notation

The **running time** of an algorithm is the number of computation steps a computer performs for an input of size N. If an algorithm's running time is linear (logarithmic) in N, then the number of computation steps increases linear (logarithmic) in N.

Big O notation is used to write the running time in compact form. See the table below for examples.

$O(N)$	Algorithm runs in linear time.
$O(logN)$	Algorithm runs in logarithmic time.
$O(N^2)$	Algorithm runs in quadratic time.

2.5 Commitments

We have now made big progress. The way we transformed the problem allows us to state that the CI statement is true if $p(x)$ is a polynomial of low degree. But before we go into detail on how low-degree testing works, there are some other caveats we need to deal with first. For now just assume that the verifier knows how to do the low degree testing.

So far we know that equation (9) needs to hold if the CI statement is true. To verify the equation, the verifier needs data points on $p(x)$ and $f(x)$ from the prover. However, for each query a malicious prover could respond with any random values for $p(x)$ and $f(x)$ that fit equation (9). To ensure that a prover cannot just respond with random numbers we introduce commitments via Merkle trees.[7]

After the prover has calculated $f(x)$ and $p(x)$, he creates a Merkle tree for $f(x)$

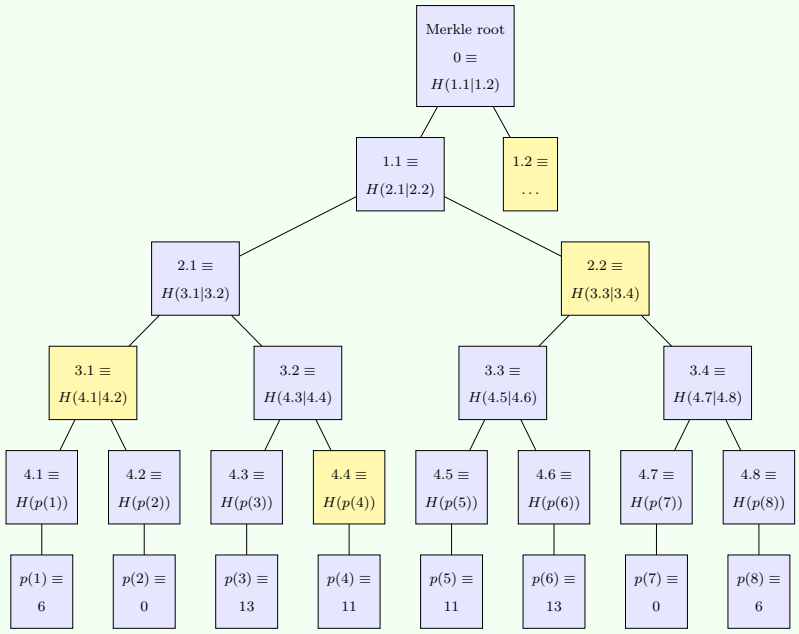
[7]There are more efficient ways to commit to polynomials. For example by using KZG polynomial commitments introduced by Kate, Zaverucha and Goldberg (2010). Due to their higher complexity we stick to Merkle trees which are easier to understand.

and one for $p(x)$ denoted as MT_f and MT_p respectively. The data blocks at the bottom of the tree consist of $f(x)\ \forall x \in L$ for MT_f and of $p(x)\ \forall x \in L$ for MT_p.

After calculating the Merkle trees, the prover sends the Merkle roots of the two trees to the verifier. This eliminates the problem that the prover might respond with random values, because the verifier can check whether these values fit the commitment (i.e. the Merkle roots).

Numerical Example: Merkle Trees

The prover calculates the Merkle trees for $p(x)$ and $f(x)$. Below the LHS of MT_p that corresponds to our example is illustrated. H denotes the hash function. The numbers in the boxes represents the level of the tree and the leave number in a given level. 3.1 is the first leave in the third level.



The prover then sends both Merkle roots

$MT_f(root) =$

`3ffb41e31ea9f86466a4871ac55d4ee78dde430fbed741b8cceb3f43321cf96e`

$MT_p(root) =$

`2e9b0882df3c19559754ea9cae76babc40fbc7950f79326046ac8479f066717e`

to the verifier.

2.6 Querying

As discussed above the verifier wants to check whether the composition polynomial $p(x)$ is of low degree. However, this test is not sufficient since the prover could just choose any random low degree $p(x)$. Thus, the verifier needs to assert that the low degree polynomial $p(x)$ relates to the constraint polynomial $c(x)$. He verifies that a rearranged form of equation (9) holds, i.e.

$$\begin{aligned} p(x)u(x) =& c(x) \\ p(x)(x^N - 1) =& f(x)^2 - f(x). \end{aligned} \tag{10}$$

A verifier checks whether equation (10) holds by choosing a random x, querying for the corresponding $f(x)$ and $p(x)$ and calculating $u(x)$ by himself. We discuss the querying protocol in more detail below but first want to motivate why this test is relevant.

If a verifier chooses a random x-value and the prover is honest, equation (10) will hold for all possible x-values. This is only true if $p(x)$ and $f(x)$ are the correct polynomials. Now assume there is a malicious prover who wants to cheat by changing $p(x)$ to

$$p_{false}(x) = 15x^2 + 6x + 13.$$

Furthermore he uses equation (10) to calculate the corresponding $c_{false}(x)$, i.e.

$$c_{false}(x) = p_{false}(x)u(x) = 15x^6 + 6x^5 + 13x^4 + 2x^2 + 11x + 4.$$

Figure 1 depicts the correct $p(x)$ and $c(x)$ from the numeric example and their corresponding false counterparts. You can see that the polynomials differ quite a lot. $p(x)$ and $p_{false}(x)$ are equal in only two points, $c(x)$ and $c_{false}(x)$ in six. If a verifier receives the correct $f(x)$ to calculate the RHS of equation (10) and receives the wrong $p_{false}(x)$ to calculate the LHS, he would spot that the equation does not hold with a relatively high probability.

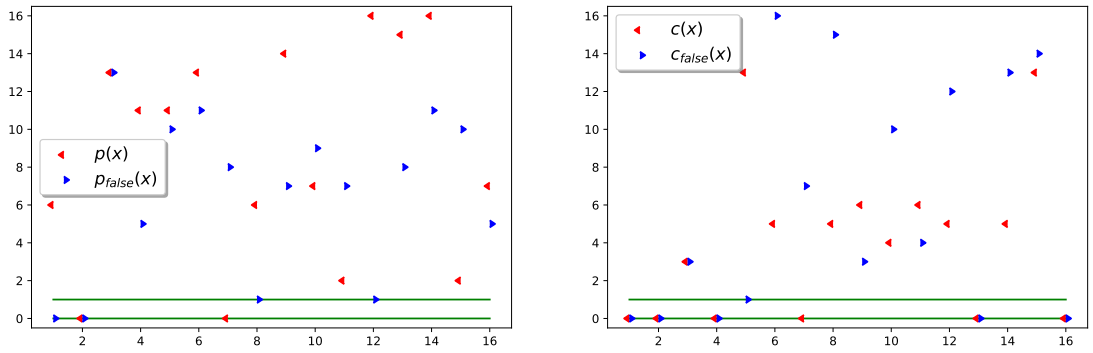


Figure 1: Illustration of a malicious prover choosing a random low degree polynomial $p_{false}(x)$ and the corresponding $c_{false}(x)$ and their correct counterparts.

This is due to another property of polynomials. Two polynomials of degree d have at most d intersections (see definition 10). This is why $p(x)$ and $p_{false}(x)$ which have a degree of 2 intersect in two points and $c(x)$ and $c_{false}(x)$ which have a degree of 6 have six intersections. In this example the probability of choosing a x for which $c(x)$ and $c_{false}(x)$ intersect is still pretty high. However, this is only the case because we chose a small domain L for illustrative purposes. In a real application we would evaluate $c(x)$ in a much larger domain which increases the number of distinct points by keeping the intersections constant. Assume we changed the domain to $L(1000)$ and also adjust the finite field to $F(1000)$. Of all the 1000 points in which the polynomial is evaluated, the two polynomials are equal only in 6. This means that if we evaluate two polynomials of equal degree in a larger domain, they are quite far apart, i.e. have many different values. Therefore, the probability that a verifier queries at a point in which both polynomials are equal is very small (0.6%) which in turn means that the probability of catching a malicious prover is very high. The probability of doing so in the numerical example with $L(1000)$ above is 94.4% after only one query. After 10 queries the probability would already go up to $1 - 6 * 10^{-23}$. Thus a verifier would spot that equation (10) does not hold almost certainly.

Definition 10: Intersections of Polynomials

If two polynomials f and g both have degree d, then there are at most d intersections.

To see this, we define another polynomial $h = f - g$ that also has degree d. The roots of h will obviously be the intersections of f and g. Using a finite field version of the fundamental theorem of algebra we know that a single-variable polynomial of degree d has at most d roots.[a] This shows that two polynomials of degree d have at most d intersections.

[a] The fundamental theorem of algebra states that a single-variable polynomial of degree d has **exactly** d roots, because it considers complex roots as well. Since in finite fields we do not work with complex numbers, the theorem changes to **at most**.

We have seen that it is very unlikely that the prover can cheat by choosing another low degree polynomial $p_{false}(x)$ that does not relate to $f(x)$ and $c(x)$. What if a prover would want to change both $p(x)$ and $c(x)$ such that equation (10) holds? It is easy to find another $c_{false}(x)$ that relates to $p_{false}(x)$. However, remember that the verifier does not check $c(x)$ but recalculates $c(x) = f(x)^2 - f(x)$ with the value for $f(x)$ he receives from the prover. So basically the prover would need to find a different polynomial $f_{false}(x)$ which is defined by $c_{false}(x) = f_{false}(x)^2 - f_{false}(x)$. It is however not feasible to find a $f_{false}(x)$ that fits the constraint. As we know from earlier there is only one low degree polynomial f that has roots for all elements of the group G. And as known from definition 8, if f does not have roots for all elements in G then equation (10) does not hold in the first place. If it does not hold, the polynomials in the LHS and RHS of equation (10) would be again far apart similar to what you saw in figure 1 which makes it relatively easy to catch a malicious prover.

After describing the motivation for verifying equation (10), we describe the protocol. As mentioned above the verifier chooses a random x. However, it is crucial in which domain the verifier is allowed to query. If the verifier can query in the whole domain L including in G, then the verifier can derive information about

the trace. Because for all $x \in G$ he receives $f(x)$ from the prover and as we know we have

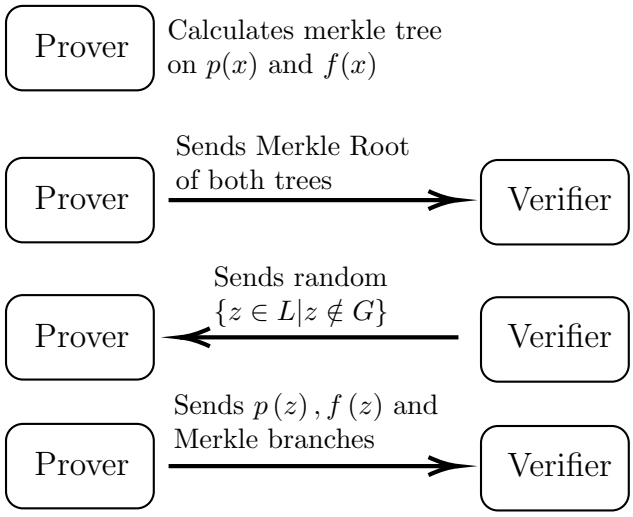
$$f(x_i) \to A_i \quad \forall x_i \in G.$$

As a consequence the zero knowledge property would be violated. If we allow for queries $x \in L$ we have a validity proof for which we care only about the computational integrity of the statement and a succinct verification thereof. If we want to further ensure that zero knowledge is guaranteed, we can only allow the verifier to query for $\{z \in L | z \notin G\}$.[8] This is what we henceforth assume.

Upon a query for a value z by the verifier, the prover returns $p(z), f(z)$ and the Merkle paths of both trees.[9] The verifier now checks whether $p(z)$ and $f(z)$ (i) correspond to the committed Merkle root using the Merkle paths and (ii) fit in equation (10), i.e.

$$p(z)(z^N - 1) = f(z)^2 - f(z).$$

Th verifier can query repeatedly for different z to increase the probability to catch a malicious prover. The interaction in this part of the proof is illustrated below.



[8] In real applications a verifier can normally query for all committed values. This would complicate things a bit which is why we decide to set up the protocol in that way.

[9] A Merkle path is the set of all hashing partners of $f(z)$ or $p(z)$ at each level of the tree that are needed to calculate the Merkle root.

Numerical Example: Querying

Assume that the verifier queries for $z = 3$. The prover then returns $f(3) = 13$, $p(3) = 13$ and the Merkle paths of MT_f and MT_p. We here only conceptually describe how the commitment is verified. Upon receiving $p(3) = 13$ the verifier hashes it to get the first leave in the tree. This would be leave 4.3 in the illustration of the Merkle tree in section 2.5. The Merkle path for MT_p the prover sends is the set of all yellow boxes, i.e. $\{4.4, 3.1, 2.2, 1.1\}$. The verifier than hashes 4.3 pairwise with the nodes in the Merkle path until he ends up at the top and the Merkle root results. If the calculated Merkle root does not fit the Merkle root received upfront, the proof is invalid.

Additionally, the verifier checks whether equation (10) holds using the values received from the prover and calculating $(z^N - 1)$ himself

$$\begin{aligned} p(z)(z^N - 1) &= f(z)^2 - f(z) \\ p(3)(3^4 - 1) &= f(3)^2 - f(3) \\ 13 \cdot 12 &= 13^2 - 13 \\ 3 &= 3 \end{aligned}$$

which is true. Again, if this did not hold, the proof is invalid. The verifier can query several times until he is convinced with a sufficiently high probability that equation (10) holds.

3 Low-Degree Testing

By now the verifier knows that $p(x)$ is the polynomial for which the constraints hold with high probability. Thus, we can talk about low-degree testing. Before we start and discuss why low-degree testing is even necessary, we do a quick recap of what we have done so far. We denote the prover as **P** and the verifier as **V**.

- **P** and **V** agree on a statement to be proven, i.e. the trace only contains 0's and 1's.
- **P** and **V** agree on constraints. If they hold, the statement is true.
- **P** calculates a polynomial f that maps a certain set of inputs G to the trace.
- **P** extends the polynomial f to a larger domain, i.e. to $L \subseteq F$.
- **P** calculates the composition polynomial p.
- **P** commits to f and p.
- **V** queries **P** to check whether p is the correct polynomial for which the constraints defined before hold.

3.1 Motivation

So far we know that a verifier will check whether equation (10) holds. Additionally, we mentioned that the second check a verifier does is whether $p(x)$ is of low degree. Before we explain how this works, we motivate how a malicious prover could cheat if $p(x)$ is not of low degree.

If we did not have low degree testing, then the only check a verifier makes is checking whether equation (10), i.e.

$$\underbrace{p(x)(x^N - 1)}_{c_1(x)} \stackrel{?}{=} \underbrace{f(x)^2 - f(x)}_{c_2(x)}$$

holds. Above we already discussed that a malicious prover cannot cheat by using another low degree $p(x)$. But what if $p(x)$ and $f(x)$ are of high degree? Is it now possible to satisfy equation (10)?

The idea is that a malicious prover could search for a high degree $p^H(x)$ and a corresponding $f^H(x)$ such that equation (10) holds for most x. This is possible

because higher degree polynomials can be "shaped" however needed. If a verifier queries for some random points, he would not detect the false proof with high probability.

One (of many possible) example would be the two high degree polynomials

$$\begin{aligned} f^H(x) =& x^{15} + 4x^{14} + 6x^{13} + 11x^{12} + 6x^{11} + 5x^{10} + 6x^9 + 13x^8 \\ &+ 7x^7 + 7x^6 + 12x^5 + 8x^4 + 2x^3 + 2x^2 + 9x + 3 \\ p^H(x) =& 13x^{15} + 8x^{14} + 11x^{13} + 2x^{12} + 16x^{11} + 9x^{10} + 3x^9 + 11x^8 \\ &+ 6x^7 + 14x^6 + 13x^5 + 4x^4 + 8x^3 + 3x^2 + 2x + 14 \end{aligned}$$

The left graph of figure 2 depicts $f^H(x)$. The prover tries to cheat by having a number different from 0 or 1 in $g^4 = 16$. Remember that given a query for $\{z \in L | z \notin G\}$ the prover responds with $p^H(z)$ and $f^H(z)$. The verifier then checks whether the left hand side of equation (10) denoted as $c_1^H(x) = p^H(x)(x^N - 1)$ is equal to the right hand side denoted as $c_2^H(x) = f^H(x)^2 - f^H(x)$. The right graph of figure 2 shows $c_1^H(x)$ and $c_2^H(x)$.

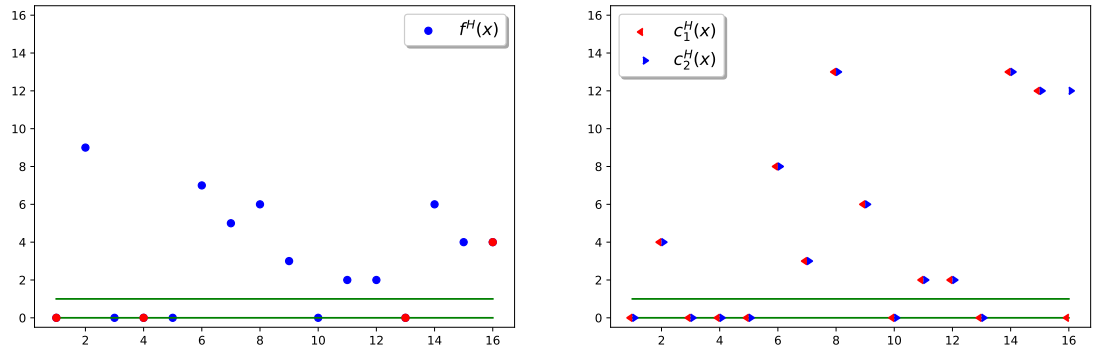


Figure 2: Cheat with high degree $p^H(x)$ and $f^H(x)$.

It follows that

$$c_1^H(x) = c_2^H(x) \quad \forall \{x \in L | x \neq 16\}$$

If the verifier queries for random points, he will only discover that the proof is false

if he queries for $x = 16$ and hence the proof would pass with a high probability. If we further consider the zero knowledge property, i.e. the verifier cannot query for $x \in G$, the probability that the false proof passes is 1.

Contrary, we have seen that this attack is not possible if $p(x)$ is a low degree polynomial because any other polynomial of equal degree is sufficiently different such that the verifier would discover the false proof. Thus, we need low degree testing.

3.2 Setup

We want to check whether $p(x)$ is a polynomial of low-degree. Remember that in section 2.4 we showed that for our example the degree of the composition polynomial is

$$deg(p) \leq N - 2 = \bar{d}$$

where we denote the highest acceptable degree as $\bar{d}$.

The most obvious way of testing whether a polynomial has degree d is querying for $d + 2$ values. Remember from the Unisolvence theorem that a polynomial of degree d can be uniquely determined by $d + 1$ distinct values. Thus, using the $d+1$ values we can determine the polynomial and ensure with the additional query that the polynomial is correct. However, if we have more complex problems, N and thus d might be very large such that it would be inefficient to do $d+1$ queries.

There is a way how the verifier can ensure with just a few querying steps that $p(x)$ is a polynomial that has at most degree $\bar{d}$ (i.e. satisfies the constraints) with high probability. We here show the FRI protocol[10] developed by Ben-Sasson et al. (2017). The basic idea is to reduce the dimensionality of the problem by transforming the polynomial in a way that in each step the degree of the

[10]Fast Reed-Solomon Interactive Oracle Proof of Proximity.

polynomial is halved. After a certain number of steps λ we end up with a constant. Since λ is directly related to the degree of the polynomial, we know that if λ does not exceed a certain number, the polynomial is with high probability of low degree. The FRI protocol is much more efficient than just querying for $d+2$ values since its running time is $O(logN)$.

3.3 Commitment Round

The polynomial $p(x)$ is transformed in each iteration using the FRI operator. Applying it yields a new polynomial $p_1(x^2)$ for which $deg(p_1) = deg(p)/2$.[11] Furthermore, the domain in which the polynomial is evaluated is also halved such that $p_1(x^2)$ is not evaluated in L but in L^2.[12] Here it becomes important that we chose the size of L to be a power of 2 such that the halving of the domain works. We can repeatedly apply the FRI operator such that after $\lambda = log_2(deg(p)) + 1$ iterations a constant remains. This is illustrated below

$$
\begin{aligned}
&p(x) \text{ where } deg(p) = \bar{d} \text{ and } p \text{ is evaluated over } L \\
&\qquad \downarrow \text{ FRI operator} \\
&p_1(x^2) \text{ where } deg(p_1) = \bar{d}/2 \text{ and } p \text{ is evaluated over } L^2 \\
&\qquad \downarrow \text{ FRI operator} \\
&p_2(x^4) \text{ with } deg(p_2) = \bar{d}/4 \text{ and } p \text{ is evaluated over } L^4 \\
&\qquad \downarrow \text{ FRI operator} \\
&p_3(x^8) \text{ with } deg(p_3) = \bar{d}/8 \text{ and } p \text{ is evaluated over } L^8 \\
&\qquad \vdots \\
&p_\lambda(x^{2d}) = const \quad (deg(p_\lambda) = 0)
\end{aligned}
$$

[11] We assume here that $deg(p) = 2^n$ for $n = \{0, 1, 2, \ldots\}$. In general it holds that if $2^{n-1} < deg(p) < 2^n$, then $2^{n-2} < deg(p_1) < 2^{n-1}$.

[12] Since the size of L is a power of 2, calculating L^2 in a finite field is equal to halving the size of L. See the numeric example for the intuition.

If after λ FRI operations the resulting polynomial is not a constant, then the composition polynomial $p(x)$ is not of low degree and the proof fails.

The FRI operator works as following. The prover splits up the polynomial $p(x)$ into its even and odd parts

$$p(x) = g_0(x^2) + xh_0(x^2) \tag{11}$$

where g_0 contains the even parts and h_0 the odd parts. The next step in the protocol is that the prover receives a random $\alpha_0 \in F$ from the verifier and uses it to calculate a new function

$$p_1(x) = g_0(x) + \alpha_0 h_0(x) \text{ for } x \in L^2. \tag{12}$$

The last step in an iteration of the FRI operator is that the prover commits $p_1(x)$ in a Merkle tree by sending the Merkle root[13] of the tree to the verifier.

This algorithm is repeatedly applied. So generally we can say that in the each iteration for $k \in \{0, 1, 2, \ldots, \lambda\}$, the prover splits

$$p_k(x) = g_k(x^2) + xh_k(x^2)$$

gets a random $\alpha_k \in F$ from the verifier, calculates

$$p_{k+1}(x) = g_k(x) + \alpha_k h_k(x).$$

and commits $p_{k+1}(x)$ in a Merkle tree over $L^{(2^k)}$. The algorithm is stopped after λ iterations when a constant remains. The prover sends the constant to the verifier.

[13] In a non-interactive proof, the prover could use this Merkle root to create a pseudorandom α.

Numerical Example: FRI Commitment Round

Recall that the composition polynomial is defined as

$$p(x) = x^2 + 8x - 3.$$

The degree of $p(x)$ is $\bar{d} = deg(p) = N - 2 = 2$. Thus we expect to need $\lambda = log_2(deg(p)) + 1 = log_2(2) + 1 = 2$ iterations.

In the first iteration of the FRI operator, the prover splits

$$\begin{aligned} p(x) =& x^2 + 8x - 3 \\ =& \underbrace{x^2 - 3}_{g_0(x^2)} + x \cdot \underbrace{8}_{h_0(x^2)} \end{aligned}$$

and thus

$$\begin{aligned} g_0(x) &= x - 3 \\ h_0(x) &= 8. \end{aligned}$$

Next, the verifier sends α_0, assume $\alpha_0 = 10$. The prover then calculates

$$\begin{aligned} p_1(x) &= g_0(x) + \alpha_0 h_0(x) \\ &= x - 3 + 10 \cdot 8 \\ &= x + 9 \end{aligned}$$

and commits $p_1(x)$ over L^2 by sending the Merkle root to the verifier. So what is L^2? We illustrate this below and realize that L^2 is perfectly symmetric why it is half the size of L.

$$\begin{aligned} L &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\} \\ L^2 &= \{1, 4, 9, 16, 8, 2, 15, 13, 13, 15, 2, 8, 16, 9, 4, 1\} \\ L^2 &= \{1, 2, 4, 8, 9, 13, 15, 16\} \end{aligned}$$

In the second iteration we have

$$\begin{aligned} p_1(x) =& x + 9 \\ =& \underbrace{9}_{\substack{g_1(x^2) \\ = g_1(x)}} + x \cdot \underbrace{1}_{\substack{h_1(x^2) \\ = h_1(x)}} \end{aligned}$$

Assume the verifier sends $\alpha_1 = 15$. The prover will calculate

$$\begin{aligned} p_2(x) &= g_1(x) + \alpha_1 h_1(x) \\ &= 9 + 15 \cdot 1 \\ &= 7 \end{aligned}$$

and send it to the verifier.

3.4 Querying Round

The verifier now needs to check whether the prover performed the FRI protocol correctly and that the last polynomial really is a constant.

Specifically, the verifier chooses a $\{z \in L | z \notin G\}$ and "replicates" the FRI protocol for this z. After λ iterations, the verifier should end up with the constant he received from the prover in the commitment round.

The verifier sends z to the prover who then returns $p(z)$ and $p(-z)$. The crucial part is that with this information the verifier can solve the following system of equations

$$\begin{aligned} p(z) &= g_0(z^2) + z h_0(z^2) \\ p(-z) &= g_0(z^2) - z h_0(z^2) \end{aligned}$$

to get $g_0(z^2)$ and $h_0(z^2)$ and then compute

$$p_1(z^2) = g_0(z^2) + \alpha_0 h_0(z^2).$$

The verifier then queries the prover for $p_1(z^2)$ and the corresponding Merkle path. He can now check whether $p_1(z^2)$ fits (i) his own calculation and (ii) the Merkle root received in the commitment round.

Now the next iteration in the querying round for z starts. The verifier also queries for $p_1(-z^2)$ and repeats the exercise above. He solves

$$\begin{aligned} p_1(z^2) &= g_1(z^4) + z^2 h_1(z^4) \\ p_1(-z^2) &= g_1(z^4) - z^2 h_1(z^4). \end{aligned}$$

to find $g_1(z^4)$ and $h_1(z^4)$ and then calculates

$$p_2(z^4) = g_1(z^4) + \alpha_1 h_1(z^4).$$

In general, the verifier chooses a $z \in L$ and in each of $k \in \{0, 1, \ldots, \lambda\}$ iterations he queries for $p_k(z^{2^k})$ and $p_k(-z^{2^k})$, calculates

$$\begin{aligned} p_k(z^{2^k}) &= g_k(z^{2^{k+1}}) + z^{2^k} h_k(z^{2^{k+1}}) \\ p_k(-z^{2^k}) &= g_k(z^{2^{k+1}}) - z^{2^k} h_k(z^{2^{k+1}}). \end{aligned}$$

to find $g_k(z^{2^{k+1}})$ and $h_k(z^{2^{k+1}})$. Lastly, he computes

$$p_{k+1}(z^{2^{k+1}}) = g_k(z^{2^{k+1}}) + \alpha_k h_k(z^{2^{k+1}})$$

and queries for the Merkle paths of $p_{k+1}(z^{2^{k+1}})$. He checks whether this fits the Merkle root received from the prover in the commitment round.

In the last iteration, the verifier can check whether the number he computes

compares to the constant he received from the prover in advance. If this is is the case, the proof is successful for this query.

The verifier can repeat this query several times for different z until he is convinced that the prover did not cheat with a sufficiently high probability.

Numerical Example: FRI Commitment Round

Assume that the verifier queries for $z = 12$. The prover then sends $p(12) = 16$ and $p(-12) = p(5) = 11$ and the verifier solves

$$16 = g_0(z^2) + 12h_0(z^2)$$
$$11 = g_0(z^2) - 12h_0(z^2)$$

which yields $g_0(z^2) = 5$ and $h_0(z^2) = 8$. Next, he calculates

$$\begin{aligned} p_1(z^2) =& g_0(z^2) + \alpha_0 h_0(z^2) \\ =& 5 + 10 \cdot 8 \\ =& 0 \end{aligned}$$

Note that $z^2 = 12^2 = 8$ and that $z^2 = 8 \in L^2$. The verifier now queries the prover for $p_1(z^2) = p_1(8)$.[a] The prover returns $p_1(8) = 0$ and the corresponding Merkle paths. The verifier can confirm that this fits his calculation $p_1(z^2) = 0$ and the Merkle root.

For the second iteration, the verifier also queries for $p_1(-z^2)$. We have $-z^2 = -8 = 9 \in L^2$. The prover returns $p_1(9) = 1$

$$0 = g_1(z^4) + 8h_1(z^4)$$
$$1 = g_1(z^4) - 8h_1(z^4)$$

which yields $g_1(z^4) = 9$ and $h_1(z^4) = 1$. Again the verifier then calculates

$$\begin{aligned} p_2(z^4) =& g_1(z^4) + \alpha_1 h_1(z^4) \\ =& 9 + 15 \cdot 1 \\ =& 7. \end{aligned}$$

This equals the constant that the verifier received upfront from the prover and the proof is successful.

[a] Remember that we had $p_1(x) = x + 9$.

3.5 Another Caveat

There is another caveat. When applying the FRI protocol it is possible to end up with a constant with less than λ steps. To see how consider the following composition polynomial which has nothing to do with the problem so far and is solely for illustration purposes.

$$p(x) = 6x^7 + 3x^6 + 2x^5 + 5x^4 + x^3 + 9x^2 + 10x + 4$$

Splitting it up in the even and odd parts yields

$$\begin{aligned} g_0(x) =& 3x^3 + 5x^2 + 9x + 4 \\ h_0(x) =& 6x^3 + 2x^2 + x + 10. \end{aligned}$$

Note that if we compute the linear combination to find $p_1(x)$, i.e.

$$p_1(x) = g_0(x) + \alpha_0 h_0(x) = 3x^3 + 5x^2 + 9x + 4 + \alpha_0(6x^3 + 2x^2 + x + 10)$$

$\exists! \tilde{\alpha}_0 \in F$ for which the x^3 term disappears. This is the case when $\tilde{\alpha}_0 = 8$:

$$\begin{aligned} p_1(x) &= 3x^3 + 5x^2 + 9x + 4 + 8(6x^3 + 2x^2 + x + 10) \\ &= 3x^3 + 5x^2 + 9x + 4 + 14x^3 + 16x^2 + 8x + 12 \\ &= 0x^3 + 4x^2 + 0x + 16 \\ &= 4x^2 + 16. \end{aligned}$$

This shows that if $\tilde{\alpha}_0$ is chosen, we end up with a lower degree polynomial than expected and would approach the constant faster than we are supposed to. This would allow to cheat with a high degree polynomial. Thus it is crucial that α is chosen randomly. Nevertheless, it is possible that by accident the verifier selects $\tilde{\alpha}$. However, there exists only one $\tilde{\alpha}$, and thus the probability that the verifier selects it is $1/M$. For large fields this probability is very small.

4 Attack With False Trace

Assume that a malicious prover's trace is $A^{false} = [2, 0, 1, 1]$. In this case, the polynomials $f_f(x)$ and $c_f(x) = f_f^2(x) - f_f(x)$ are

$$\begin{aligned} f_f(x) &= 12x^3 + 9x^2 + 14x + 1 \\ c_f(x) &= 8x^6 + 12x^5 + 9x^4 + 9x^3 + x^2 + 14x \end{aligned}$$

The roots of $c_f(x)$ are $x = \{0, 5, 8, 12\}$ which is different to G. This contradicts definition 8 and hence $p_f(x)$ should not be a polynomial. Indeed we get

$$p_f(x) = \frac{c_f(x)}{x^N - 1} = 8x^2 + 12x + 9, \text{ Reminder: } R(x) = 9x^3 + 9x^2 + 9x + 9$$

What if a malicious prover commits to $p_f(x)$ and $f_f(x)$? The verifier would still

check equation (10), ie

$$\underbrace{p_f(x)(x^N - 1)}_{\bar{c}(x)} = \underbrace{f_f(x)^2 - f_f(x)}_{c_f(x)}$$

but since definition 8 is violated, the LHS of the equation denoted as $\bar{c}(x)$ would be a different polynomial that is far from $c_f(x)$. This is illustrated in figure 3. Since $c_f(x)$ and $\bar{c}(x)$ have both a degree of six, they intersect at most six times and hence a verifier will spot a wrong trace with a very high probability if the domain is large enough.

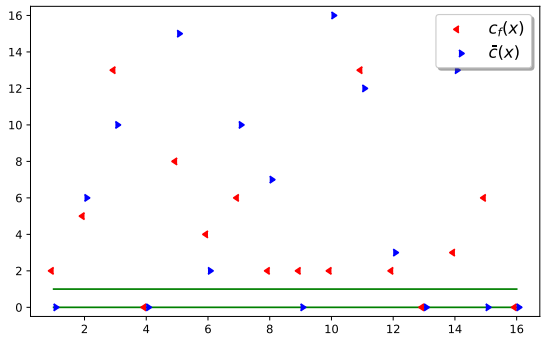


Figure 3: Cheat With Wrong Trace.

5 Summary of the Protocol

To conclude we summarize the protocol. The prover is denoted by **P**, the verifier as **V**.

1 Definition of the problem

1.1 **P** and **V** agree on a CI statement and constraints that must hold for the CI statement to be correct.

1.2 **P** and **V** agree on the finite field F they work in.

2 Arithmetization

2.1 **P** defines a low-degree polynomial f that relates a subgroup $G \subset F$ to the trace.

2.2 **P** evaluates f on a larger domain L. L is known by **V** as well.

2.3 **P** combines constraints with the polynomial to get a new polynomial c.

2.4 **P** creates the composition polynomial p which is a polynomial of low degree iff the CI statement is true.

2.5 **P** commits over f and p using Merkle trees and sends the Merkle roots to **V**.

2.6 **V** queries **P** for values of $f(x)$ and $p(x)$ and checks whether they fit the commitment and the required relationship between f and p.

3 Low-degree testing

3.3 Commitment round: **P** applies the FRI protocol, i.e. he reduces the degree of the polynomial by repeatedly applying the FRI operator and makes commitments until a constant remains which he sends to **V**. He uses random inputs by the verifier and sends the commitment (i.e. the Merkle root) in each round to **V**.

3.4 Querying round: **V** verifies that the FRI protocol was performed correctly by querying for values in L and "replicating" the protocol for this value. The proof is successful if the commitments are correct and the constant found by **V** matches the constant sent by **P** before.

References

Ben-Sasson, Eli. 2019. "StarkWare: Transparent Computational Integrity with Eli Ben Sasson." *https://softwareengineeringdaily.com/2019/03/04/starkware-transparent-computational-integrity-with-eli-ben-sasson/ (Date Accessed: 2021-11-10).*

Ben-Sasson, Eli, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2017. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity." *Electron. Colloquium Comput. Complex.*, TR17.

Ben-Sasson, Eli, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. "Scalable, transparent, and post-quantum secure computational integrity." *IACR Cryptol. ePrint Arch.*, 2018: 46.

Berentsen, Aleksander, Jeremias Lenzi, and Remo Nyffenegger. 2022. "An Introduction to Zero-Knowledge Proofs in Blockchains and Economics."

Buterin, Vitalik. 2017*a*. "STARKs, Part 3: Into the Weeds." *https://vitalik.ca/general/2018/07/21/starks_part_3.html (Date Accessed: 2022-04-19).*

Buterin, Vitalik. 2017*b*. "STARKs, Part I: Proofs with Polynomials." *https://vitalik.ca/general/2017/11/09/starks_part_1.html (Date Accessed: 2022-04-19).*

Buterin, Vitalik. 2017*c*. "STARKs, Part II: Thank Goodness It's FRI-day." *https://vitalik.ca/general/2017/11/22/starks_part_2.html (Date Accessed: 2022-04-19).*

Kate, Aniket, Gregory M. Zaverucha, and Ian Goldberg. 2010. "Constant-Size Commitments to Polynomials and Their Applications." *International Conference on the Theory and Application of Cryptology and Information Security.*

StarkWare. 2019*a*. "Arithmetization I." *https://medium.com/starkware/arithmetization-i-15c046390862 (Date Accessed: 2022-04-19).*

StarkWare. 2019*b*. "Arithmetization II." *https://medium.com/starkware/arithmetization-ii-403c3b3f4355 (Date Accessed: 2022-04-19).*

StarkWare. 2019*c*. "Low Degree Testing." *https://medium.com/starkware/low-degree-testing-f7614f5172db (Date Accessed: 2022-04-19).*

StarkWare. 2019*d*. "STARK Math: The Journey Begins." *https://medium.com/starkware/stark-math-the-journey-begins-51bd2b063c71 (Date Accessed: 2022-04-19).*

StarkWare. 2020*a*. "STARK 101 Part 1 - Statement, LDE and Commitment." *https://www.youtube.com/watch?v=Y0uJz9VL3Fo&list=PLcIyXLwiPilWoXrDbmwHPxaH8Gxk5I_fG (Date Accessed: 2022-04-19).*

StarkWare. 2020*b*. "STARK 101 part 2 - Polynomial Constraints." *https://www.youtube.com/watch?v=fg3mFPXEYQY&list=PLcIyXLwiPilWoXrDbmwHPxaH8Gxk5I_fG&index=3 (Date Accessed: 2022-04-19).*

StarkWare. 2020*c*. "STARK 101 Part 3 - FRI Commitment." *https://www.youtube.com/watch?v=gd1NbKUOJwA&list=PLcIyXLwiPilWoXrDbmwHPxaH8Gxk5I_fG&index=4 (Date Accessed: 2022-04-19).*

StarkWare. 2020*d*. "STARK 101 Part 4 - The Proof." *https://www.youtube.com/watch?v=CxP28qM4tAc&list=PLcIyXLwiPilWoXrDbmwHPxaH8Gxk5I_fG&index=5 (Date Accessed: 2022-04-19).*