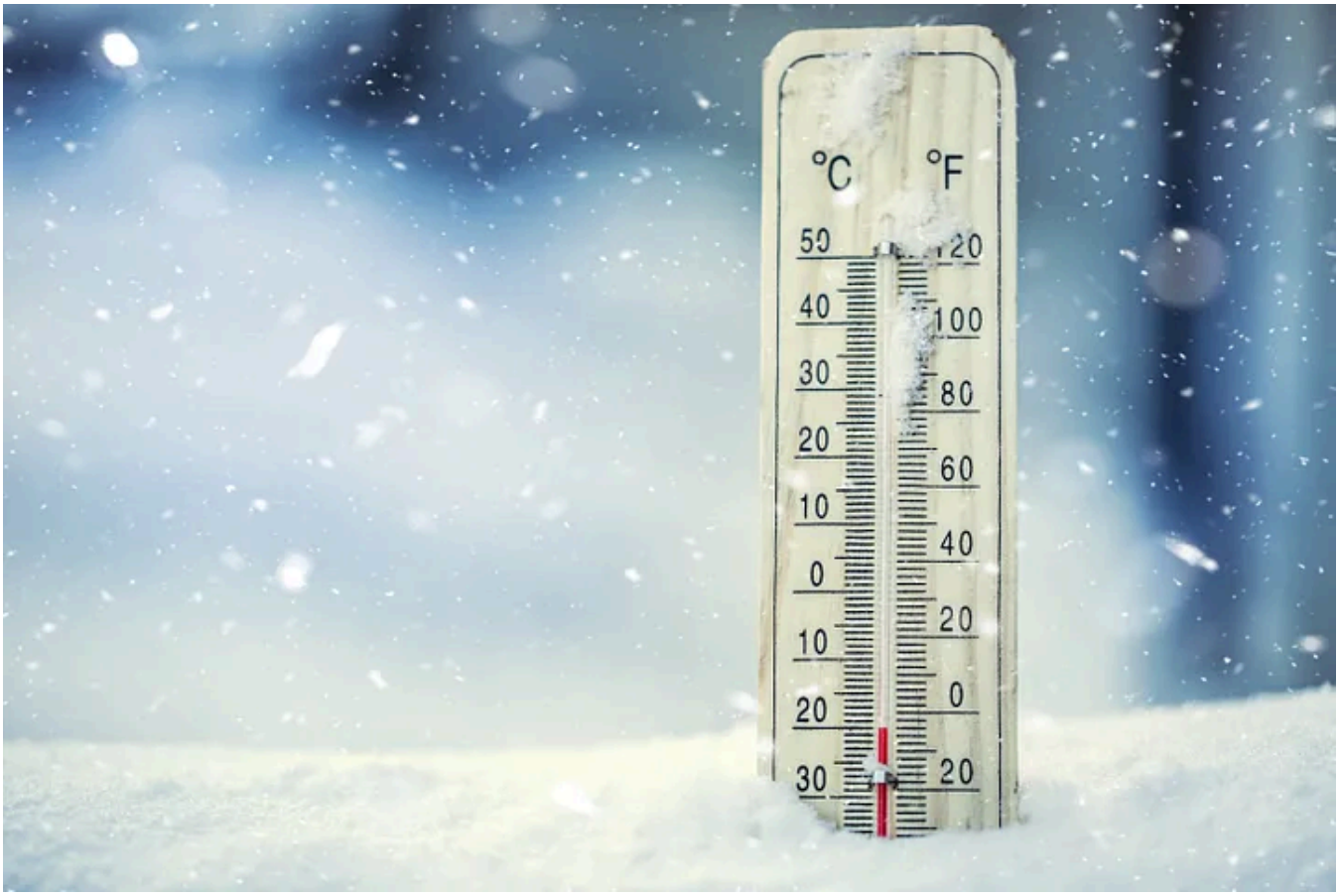
Low Degree Testing

The Secret Sauce of Succinctness

StarkWare · Follow
Published in StarkWare
15 min read · Mar 28, 2019



This is the fourth post in our STARK Math series. If you have not yet done so, we recommend that you read posts 1 (*The Journey Begins*), 2 (*Arithmetization I*) and 3 (*Arithmetization II*) before you read this one. We have so far explained how, in

STARK, the process of Arithmetization enables us to reduce the problem of Computational Integrity to a low degree testing problem.

L*ow degree testing* refers to the problem of deciding whether a given function is a polynomial of some bounded degree, by making only a small number of queries to the function. Low degree testing has been studied for more than two decades, and is a central tool in the theory of probabilistic proofs. The goal of this blog post is to explain low degree testing in more detail, and to describe *FRI*, the protocol that we use for low degree testing in STARK. This post assumes familiarity with polynomials over finite fields.

To make this very mathy post easier to digest we marked certain paragraphs like so:

> *paragraphs which contain proofs or explanations that are not necessary to understand the bigger picture will be marked like this*

Low Degree Testing

Before we discuss low-degree testing, we first present a slightly simpler problem as a warm-up: We are given a function and are asked to decide whether this function is equal to some polynomial of degree *less than* some constant d, by querying the function at a "small" number of locations. Formally, given a subset L of a field F and a degree bound d, we wish to determine if f:L→F is equal to a polynomial of degree less than d, namely, if there exists a polynomial

$$p(x) = c_0 + c_1 x + \cdots + c_{d-1} x^{d-1}$$

over F for which p(a) = f(a) for every a in L. For concrete values, you may think of a field of size which is very large, say 2^{128}, and L which is of size approximately 10,000,000.

Solving this problem requires querying f at the entire domain L, as f might agree with a polynomial everywhere in L except for a single location. Even if we allow a constant probability of error, the number of queries will still be linear in the size of L.

For this reason, the problem of low degree testing actually refers to an approximate relaxation of the above problem, which suffices for constructing probabilistic proofs and also can be solved with a number of queries which is logarithmic in |L|

(note that if L≈10,000,000, then $\log_2(L)\approx 23$). In more detail, we wish to distinguish between the following two cases.

- **The function f is equal to a low degree polynomial.** Namely, there exists a polynomial p(x) over F, of degree less than d, that agrees with f everywhere on L.
- **The function f is far from ALL low degree polynomials.** For example, we need to modify *at least* 10% of the values of f before we obtain a function that agrees with a polynomial of degree less than d.

Note that there is another possibility — the function f may be mildly close to a low degree polynomial, yet not equal to one. For example, a function in which 5% of the values differ from a low-degree polynomial does not fall in either of the two cases described above. However, the prior arithmetization step (discussed in our previous posts) ensures the third case never arises. In more detail, arithmetization shows that an honest prover dealing with a true statement will land in the first case, whereas a (possibly malicious) prover attempting to "prove" a false claim will land, with high probability, in the second case.

In order to distinguish the two cases, we will use a probabilistic polynomial-time test that queries f at a small number of locations (we discuss what "small" means later).

> *If f is indeed low degree, then the test should accept with probability 1. If instead f is far from low degree, then the test should reject with high probability. More generally, we seek the guarantee that if f is δ-far from any function of degree less than d (i.e., one must modify at least δ|L| locations to obtain a polynomial of degree less than d), then the test rejects with probability at least Ω(δ) (or some other "nice" function of δ). Intuitively, the closer δ is to zero, the more difficult it is to distinguish between the two cases.*

In the next few sections we describe a simple test, then explain why it does not suffice in our setting, and finally we describe a more complex test that is exponentially more efficient. This latter test is the one that we use in STARK.

The Direct Test

The first test we consider is a simple one: it tests whether a function is (close to) a polynomial of degree less than d, using d+1 queries. The test relies on a basic fact about polynomials: *any polynomial of degree less than d is fully determined by its values*

at any d distinct locations of F. This fact is a direct consequence of the fact that a polynomial of degree *k* can have at most *k* roots in F. Importantly, the number of queries, which is d+1, can be significantly less than the size of the domain of f, which is |L|.

We first discuss two simple special cases, to build intuition for how the test will work in the general case.

- **The case of a constant function (d=1).** This corresponds to the problem of distinguishing between the case where f is a constant function (f(x)=c for some c in F), and the case where f is far from any constant function. In this special case there is a natural 2-query test that might work: query f at a *fixed* location z1 and also at a *random* location w, and then check that f(z1)=f(w). Intuitively, f(z1) determines the (alleged) constant value of f, and f(w) tests whether all of f is close to this constant value or not.
- **The case of a linear function (d=2).** This corresponds to the problem of distinguishing between the case where f is a linear function (f(x)=ax+b for some a,b in F), and the case where f is far from any linear function. In this special case there is a natural 3-query test that might work: query f at two *fixed* locations z1,z2 and also at a *random* location w, and then check that (z1,f(z1)), (z2,f(z2)), (w,f(w)) are collinear, namely, we can draw a line through these points. Intuitively, the values of f(z1) and f(z2) determine the (alleged) line, and f(w) tests whether all of f is close to this line or not.

The above special cases suggest a test for the **general case of a degree bound d.** Query f at d *fixed* locations z1,z2,…,z*d* and also at a *random* location w. The values of f at z0,z1,…,z*d* define a unique polynomial h(x) of degree less than d over F that agrees with f at these points. The test then checks that h(w)=f(w). We call this the *direct test*.

By definition, if f(x) is equal to a polynomial p(x) of degree less than d, then h(x) will be identical to p(x) and thus the direct test passes with probability 1. This property is called “perfect completeness”, and it means that this test has only 1-sided error.

We are left to argue what happens if f is δ-far from any function of degree less than d. (For example, think of δ=10%.) We now argue that, in this case, the direct test rejects with probability at least δ. Indeed, let μ be the probability, over a random choice of w, that h(w)≠f(w). Observe that μ must be at least δ.

> *This is because if we assume towards contradiction that μ is smaller than δ, then we deduce that f is δ-close to h, which contradicts our assumption that f is δ-far from any function of degree less than d.*

The Direct Test Does Not Suffice For Us

In our setting we are interested in testing functions f:L→F that encode computation traces, and hence whose degree d (and domain L) are quite large. Merely running the direct test, which makes d+1 queries, would be too expensive. In order to gain the exponential savings of STARK (in verification time compared to the size of the computation trace), we need to solve this problem with only O(log d) queries, which is exponentially less than the degree bound d.

This, unfortunately, is impossible because if we query f at less than d+1 locations then *we cannot conclude anything*.

> *One way to see this is to consider two different distributions of functions f:L→F. In one distribution we uniformly pick a polynomial of degree* exactly *d and evaluate it on L. In the other distribution we uniformly pick a polynomial of degree* less *than d and evaluate it on L. In both cases, for any d locations z1,z2,…,z*d*, the values f(z1),f(z2),…,f(z*d*) are uniformly and independently distributed. (We leave this fact as an exercise for the reader.) This implies that information-theoretically we cannot tell these two cases apart, even though a test would be required to (since polynomials from the first distribution should be accepted by the test while those of degree exactly d are very far from all polynomials of degree less than d, and thus should be rejected).*

We seem to have a difficult challenge to overcome.

A Prover Comes to the Rescue

We have seen that we need d+1 queries to test that a function f:L→F is close to a polynomial of degree less than d, but we cannot afford this many queries. We avoid this limitation by considering a slightly different setting, which suffices for us. Namely, we consider the problem of low degree testing when a prover is available to supply useful auxiliary information about the function f. We will see that in this "prover-aided" setting of low-degree testing we can achieve an *exponential* improvement in the number of queries, to O(log d).

In more detail, we consider a *protocol* conducted between a *prover* and a *verifier*, wherein the (untrusted) prover tries to convince the verifier that the function is of low degree. On the one hand, the prover knows the *entire* function f being tested. On

the other hand, the verifier can query the function f at a small number of locations, and is willing to receive help from the prover, but does NOT trust the prover to be honest. This means that the prover may cheat and not follow the protocol. However, if the prover does cheat, the verifier has the liberty to "reject", regardless of whether the function f is of low degree or not. The important point here is that the verifier will not be convinced that f is of low degree unless this is true.

Note that the direct test described above is simply the special case of a protocol in which the prover does nothing, and the verifier tests the function unassisted. To do better than the direct test we will need to leverage the help of the prover in some meaningful way.

Throughout the protocol the prover will want to enable the verifier to query auxiliary functions on locations of the verifier's choice. This can be achieved via *commitments*, a mechanism that we will discuss in a future blog post. For now it suffices to say that the prover can commit to a function of its choice via a Merkle tree, and subsequently the verifier can request the prover to reveal any set of locations of the committed function. The main property of this commitment mechanism is that once the prover commits to a function, it must reveal the correct values and cannot cheat (for example, it cannot decide what the values of the function are after seeing the requests from the verifier).

Halving the number of queries for the case of two polynomials

Let's start with a simple example that illustrates how a prover can help to reduce the number of queries by a factor of 2. We will later build on this example. Suppose that we have two polynomials f and g and we want to test that they are *both* of degree less than d. If we simply run the direct test individually on f and g then we would need to make 2 * (d + 1) queries. Below we describe how with the help of a prover we can reduce the number of queries to (d + 1) plus a smaller-order term.

First, the verifier samples a random value α from the field and sends it to the prover. Next, the prover replies by committing to the evaluation on the domain L (recall that L is the domain of the function f) of the polynomial $h(x) = f(x) + \alpha\, g(x)$ (in other words, the prover will compute and send the root of a Merkle tree whose leaves are the values of h on L). The verifier now tests that *h* has degree less than d, via the direct test, which requires d+1 queries.

Intuitively, if *f* or *g* has degree at least d, then with high probability so does h. For example, consider the case where the coefficient of x^n in f is not zero for some n≥d. Then, there is at most one choice of α (sent by the verifier) for which the coefficient of x^n in h is zero, which means that the probability that h has degree less than d is roughly 1/|F|. If the field is large enough (say, $|F|>2^{128}$), the probability of error is negligible.

The situation, however, is not this simple. The reason is that, as we explained, we cannot literally check that h is a polynomial of degree less than d. Instead we only can check that h is *close* to such a polynomial. This means that the analysis above is not accurate. Is it possible that f will be far from a low degree polynomial and the linear combination h will be close to one with a non-negligible probability over α? Under mild conditions the answer is no (which is what we want), but it is outside the scope for this post; we refer the interested reader to this paper and this paper.

Moreover, how does the verifier know that the polynomial *h* sent by the prover has the form f(x)+α g(x)? A malicious prover may cheat by sending a polynomial which is indeed of low degree, but is different from the linear combination that the verifier asked for. If we already know that h is close to a low degree polynomial, then testing that this low degree polynomial has the correct form is straightforward: the verifier samples a location *z* in L at random, queries *f, g, h* at z, and checks that the equation h(z)=f(z)+α g(z) holds. This test should be repeated multiple times to increase accuracy of the test, but the error shrinks exponentially with the number of samples we make. Hence this step increases the number of queries (which so far was d+1) only by a smaller-order term.

Splitting a polynomial into two smaller-degree polynomials

We saw that, with the prover's help, we can test that two polynomials are of degree less than d with less than 2*(d+1) queries. We now describe how we can turn one polynomial of degree less than d into two polynomials of degree less than d/2.

Let f(x) be a polynomial of degree less than d and assume that d is even (in our setting this comes without loss of generality). We can write $f(x)=g(x^2)+xh(x^2)$ for two polynomials g(x) and h(x) of degree less than d/2. Indeed, we can let g(x) be the polynomial obtained from the even coefficients of f(x), and h(x) be the polynomial obtained from the odd coefficients of f(x). For example, if d=6 we can write

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$$

which means that

$$g(x) = a_0 + a_2x + a_4x^2$$

and

$$h(x) = a_1 + a_3x + a_5x^2.$$

which is an n*log(n) algorithm for polynomial evaluation (improving over the naive n2 algorithm).

The FRI Protocol

We now combine the two above ideas (testing two polynomials with half the queries, and splitting a polynomial into two smaller ones) into a protocol that only uses O(log d) queries to test that a function f has (more precisely, is close to a function of) degree less than d. This protocol is known as FRI (which stands for **F**ast **R**eed — Solomon **I**nteractive Oracle Proof of Proximity), and the interested reader can read more about it here. For simplicity, below we assume that d is a power of 2. The protocol consists of two phases: a *commit phase* and a *query phase*.

Commit phase

The prover splits the original polynomial $f_0(x)=f(x)$ into two polynomials of degree less than d/2, $g_0(x)$ and $h_0(x)$, satisfying $f_0(x)=g_0(x^2)+xh_0(x^2)$. The verifier samples a random value α_0, sends it to the prover, and asks the prover to commit to the polynomial $f_1(x)=g_0(x) + \alpha_0 h_0(x)$. Note that $f_1(x)$ is of degree less than d/2.

We can continue recursively by splitting $f_1(x)$ into $g_1(x)$ and $h_1(x)$, then choosing a value α_1, constructing $f_2(x)$ and so on. Each time, the degree of the polynomial is halved. Hence, after log(d) steps we are left with a constant polynomial, and the prover can simply send the constant value to the verifier.

A note about the domains: for the above protocol to work, we need the property that for every z in the domain L, it holds that -z is also in L. Moreover, the commitment on $f_1(x)$ will not be over L but over $L^2=\{x^2: x \in L\}$. Since we iteratively apply the FRI step, L^2 will also have to satisfy the {z, -z} property, and so on. These natural algebraic requirements are easily satisfied via natural choices of domains L (say, a multiplicative subgroup whose size is a power of 2), and in fact coincide with those

that we anyways need in order to benefit from efficient FFT algorithms (which are used elsewhere in STARK, e.g., to encode execution traces).

Query phase

We now have to check that the prover did not cheat. The verifier samples a random z in L and queries $f_0(z)$ and $f_0(-z)$. These two values suffice to determine the values of $g_0(z^2)$ and $h_0(z^2)$, as can be seen by the following two linear equations in the two "variables" $g_0(z^2)$ and $h_0(z^2)$:

$$f_0(z) = g_0(z^2) + z\, h_0(z^2)$$
$$f_0(-z) = g_0(z^2) - z\, h_0(z^2)$$

The verifier can solve this system of equations and deduce the values of $g_0(z^2)$ and $h_0(z^2)$. It follows that it can compute the value of $f_1(z^2)$ which is a linear combination of the two. Now the verifier queries $f_1(z^2)$ and makes sure that it is equal to the value computed above. This serves as an indication that the commitment to $f_1(x)$, which was sent by the prover in the commit phase, is indeed the correct one. The verifier may continue, by querying $f_1(-z^2)$ (recall that $(-z^2) \in L^2$ and that the commitment on $f_1(x)$ was given on L^2) and deduce from it $f_2(z^4)$.

The verifier continues in this way until it uses all these queries to finally deduce the value of f_{log d}(z) (denoting f with a subscript log d, that we can't write due to Medium's lack of support for fully fledged mathematical notation). But, recall that f_{log d}(z) is a constant polynomial whose constant value was sent by the prover in the commit phase, prior to choosing z. The verifier should check that the value sent by the prover is indeed equal to the value that the verifier computed from the queries to the previous functions.

Overall, the number of queries is only **logarithmic** in the degree bound d.

> *To get a feeling why the prover cannot cheat, consider the toy problem where f_0 is zero on 90% of the pairs of the form {z,-z}, i.e., $f_0(z) = f_0(-z) = 0$ (call these the "good" pairs), and non-zero on the remaining 10% (the "bad" pairs). With probability 10% the randomly selected z falls in a bad pair. Note that only one α will lead to $f_1(z^2)=0$, and the rest will lead to $f_1(z^2)\neq 0$. If the prover cheats on the value of $f_1(z^2)$, it will be caught, so we assume otherwise. Thus, with a high probability $(f_1(z^2), f_1(-z^2))$ will also be a bad pair in the next*

> *layer (the value of $f_1(-z^2)$ is not important as $f_1(z^2)\neq 0$). This continues until the last layer where the value will be non-zero with high probability.*

On the other hand, since we started with a function with 90% zeros, it is unlikely that the prover will be able to get close to a low degree polynomial other than the zero polynomial (we will not prove this fact here). In particular, this implies that the prover must send 0 as the value of the last layer. But then, the verifier has a probability of roughly 10% to catch the prover. This was only an informal argument, the interested reader may find a rigorous proof here.

In the test described so far (and the above analysis) the probability that the verifier catches a malicious prover is only 10%. In other words the error probability is 90%. This can be exponentially improved by repeating the above query phase for a few independently sampled z's. For example, by choosing 850 z's, we get an error probability of 2^{-128} which is practically zero.

Wrapping Up

In this post we have described the problem of low degree testing. The direct solution (test) requires too many queries to achieve the succinctness required by STARK. To attain logarithmic query complexity, we use an **interactive** protocol called FRI, in which the prover adds more information in order to convince the verifier that the function is indeed of low degree. Crucially, FRI enables the verifier to solve the low-degree testing problem with a number of queries (and rounds of interaction) that is *logarithmic* in the prescribed degree. The next post will summarize the last three posts and explain how we can remove the interactive aspect used in FRI and in earlier parts of the protocol, in order to get succinct non-interactive proofs.

Alessandro Chiesa & Lior Goldberg
StarkWare

Stark Math | Zero Knowledge Proofs | Mathematics | Blockchain | Zkstark

Published in StarkWare

3.8K Followers · Last published Mar 1, 2024

Developing the Full Proof Stack for STARK

Follow

Written by StarkWare

12K Followers · 3 Following

STARK-Based Scaling Solutions

Responses (4)

What are your thoughts?

Respond

Intel Chen
about 3 years ago

> Then, there is at most one choice of α (sent by the verifier) for which the coefficient of x^n in h is zero,

why

1 1 reply Reply

Graphicaldot (Saurav verma)
10 months ago

This article is not as good as the other article in the series.

Reply

Intel Chen
about 3 years ago

Then, there is at most one choice of α (sent by the verifier) for which the coefficient of x^n in h is zero,

More from StarkWare and StarkWare

In StarkWare by StarkWare

Moving from Solidity to Cairo

Guiding Devs through the Expansion into Cairo Programming

Jun 23, 2023 139

In StarkWare by StarkWare

A Framework for Efficient STARKs

STARK Math final part: combining probabilistic proofs and hash functions

Apr 17, 2019 338

In StarkWare by StarkWare

StarkDEX Deep Dive: the STARK Core Engine

Part 3 of 4

Jul 16, 2019 223 1

In StarkWare by StarkWare

Part 3: Starknet Token Design

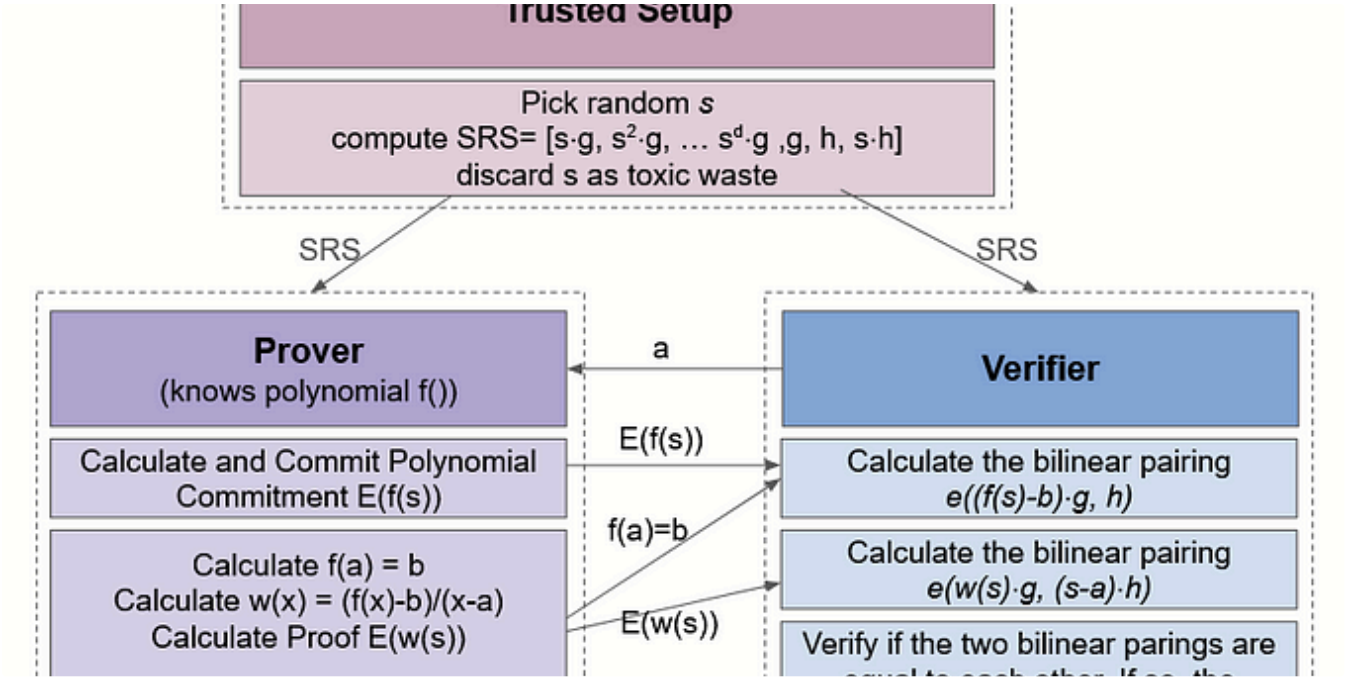
In this post we dive deeper into the design of the Starknet Token, its minting schedule and expected timeline.

Jul 13, 2022 672 9

See all from StarkWare

See all from StarkWare

Recommended from Medium

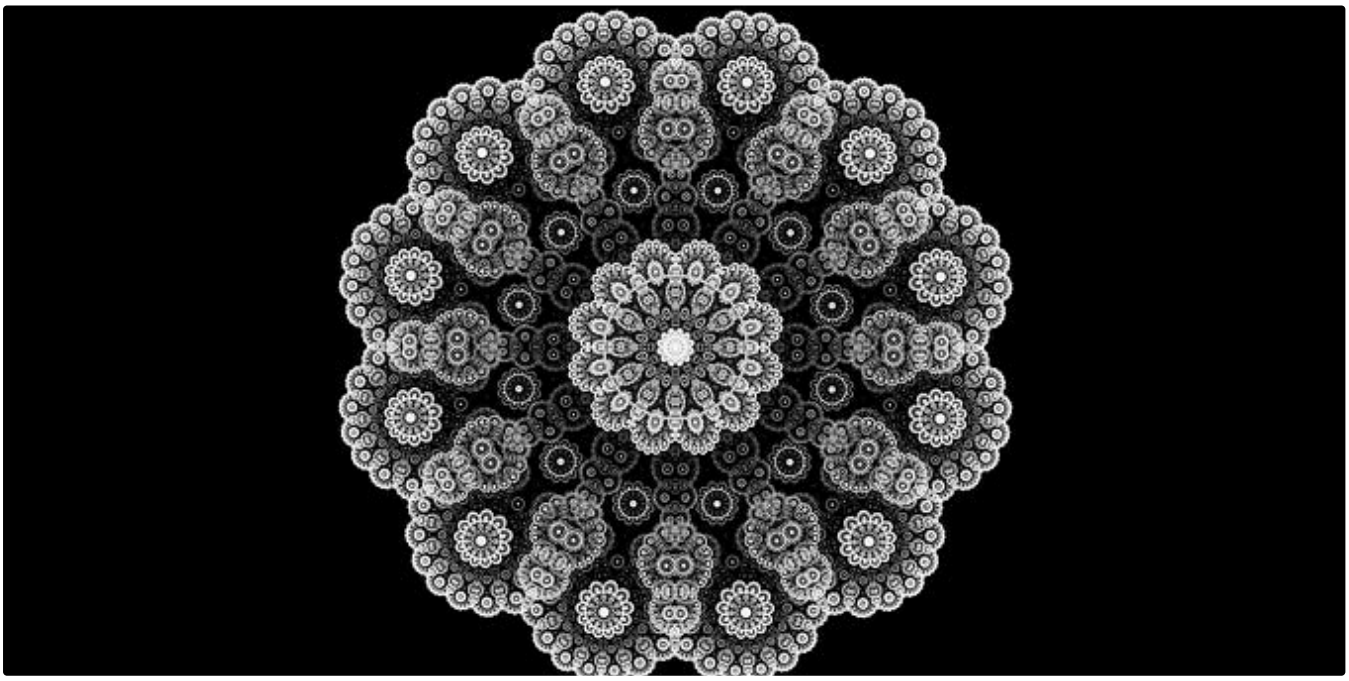


Bhaskar Krishnamachari

Understanding Zero-Knowledge Proofs: Part 4— Polynomial Commitments

In part 2 of this series, we went over some basic mathematical background needed for cryptography, namely finite fields, cyclic groups, and...

Jul 14, 2024 86



Law and Ordnung

The Fractal Allure: A Journey into the World of Repeating Patterns

What are fractals and why do they calm our minds?

Dec 27, 2024 1

Lists

My Kind Of Medium (All-Time Faves)
106 stories · 630 saves

MODERN MARKETING
204 stories · 976 saves

Medium's Huge List of Publications Accepting Submissions
377 stories · 4337 saves

Natural Language Processing
1883 stories · 1527 saves

In Coinmonks by Mustafa Akbulut

Smart Contract Vulnerabilities Unveiled: Flash Loan Attacks

Decentralized Finance (DeFi) has revolutionized the financial world by enabling users to engage in financial activities without...

Jul 8, 2024 146 2

Mohd Anas

Learning Go (Golang)

As a quite new programming language developed by Google, Go—or Golang—aims at providing a more liberal setting for building complex...

Dec 26, 2024 107 8

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26, 2024 8.6K 136

Bosun Sogeke

How to Test a Private Blockchain: A Comprehensive Guide

Testing a private blockchain is an essential step in ensuring the reliability, security, and functionality of your blockchain application...

Nov 25, 2024 549

See more recommendations