Lesson 13

Lesson 13 : Stark Theory

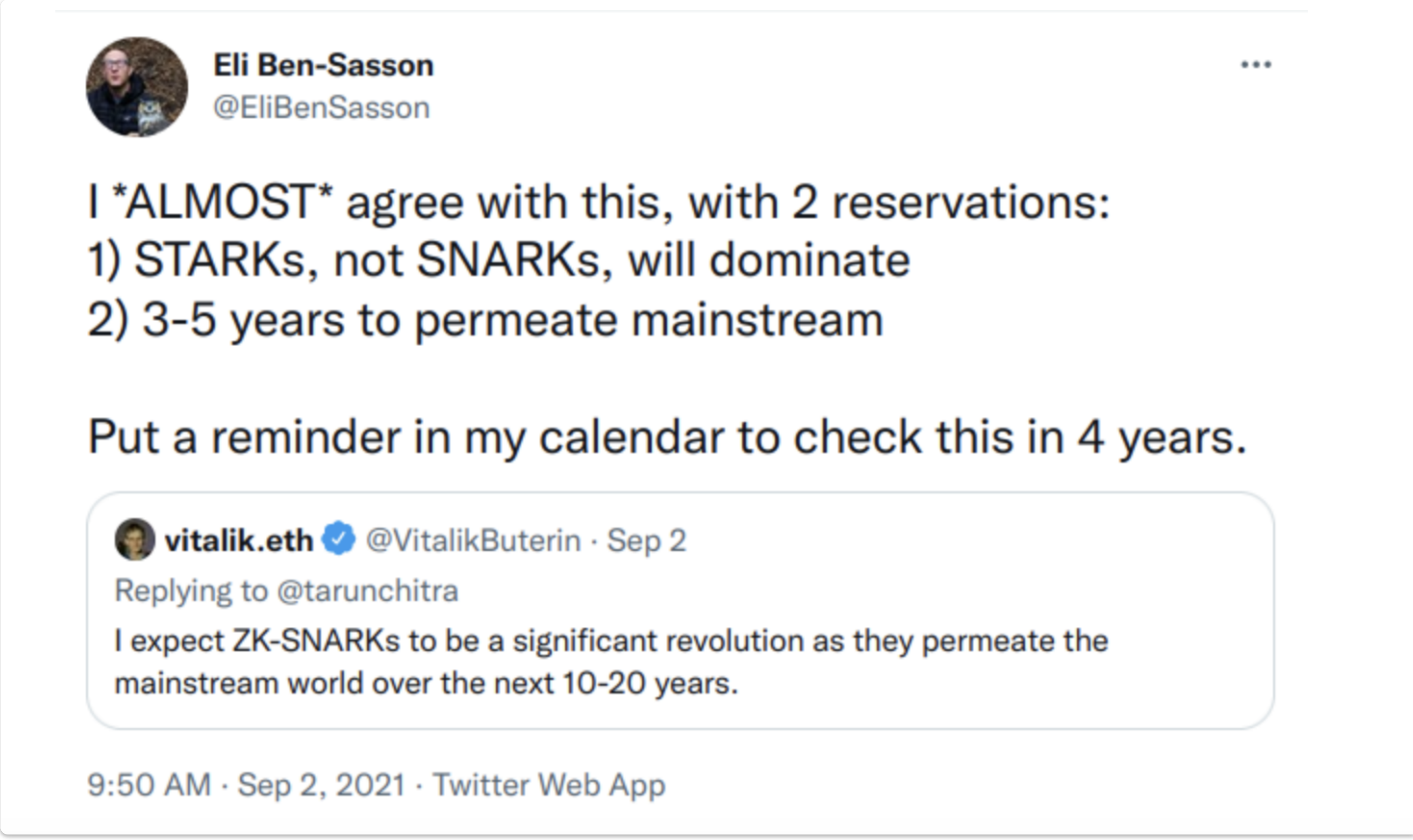
Lesson 14 : Cryptographic Alternatives

Lesson 15 : zkML

Lesson 16 : Research and review

Today's topics

- Polynomial Recap
- Error correcting codes
- Computational Integrity
- Overview of the STARK process
- FRI



Eli Ben-Sasson
@EliBenSasson

I *ALMOST* agree with this, with 2 reservations:
1) STARKs, not SNARKs, will dominate
2) 3-5 years to permeate mainstream

Put a reminder in my calendar to check this in 4 years.

> vitalik.eth @VitalikButerin · Sep 2
> Replying to @tarunchitra
> I expect ZK-SNARKs to be a significant revolution as they permeate the mainstream world over the next 10-20 years.

9:50 AM · Sep 2, 2021 · Twitter Web App

Resources

Whiteboard session about SNARKs versus STARKS

Polynomial Recap

A basic fact about polynomials and their roots is that if $p(x)$ is a polynomial, then $p(a) = 0$ for some specific value a,

if and only if there exists
a polynomial $q(x)$ such that
$(x - a)q(x) = p(x)$,

and therefore

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

and $deg(p) = deg(q) + 1$.

This is true for all roots

Error correcting codes

The basic idea is that we have a message, we add redundancy and this gives us a `code word`

Reed Solomon codes

See Introduction
Reed-Solomon codes are systematic linear codes, meaning that the original data (message) is included as part of the encoded data (codeword), which also contains extra symbols.

A Reed-Solomon code is a set of length n vectors (known as codewords), where the elements of the vector (known as symbols) consist of m binary digits. Our only restriction is that n must be chosen no larger than 2^m. Of the n symbols in each code word, k of them carry information and the other $(n-k)$ are redundant symbols.

Assume that, of the total n symbols, exactly t of them are received in error (and the other $n-t$ are received correctly).
Reed-Solomon codes have the remarkable property that if $t \leq (n-k)/2$, then the correct information can

be computed from this faulty codeword.
Furthermore, if s of the received symbols are erased (i.e, tagged as probably being faulty) and another t symbols are received in error, the correct information can be computed from the faulty code word provided that $s + 2t \leq n - k$.
The device that reconstructs the information from the received vector is called a decoder

The `rate` is k/n

Use of polynomials

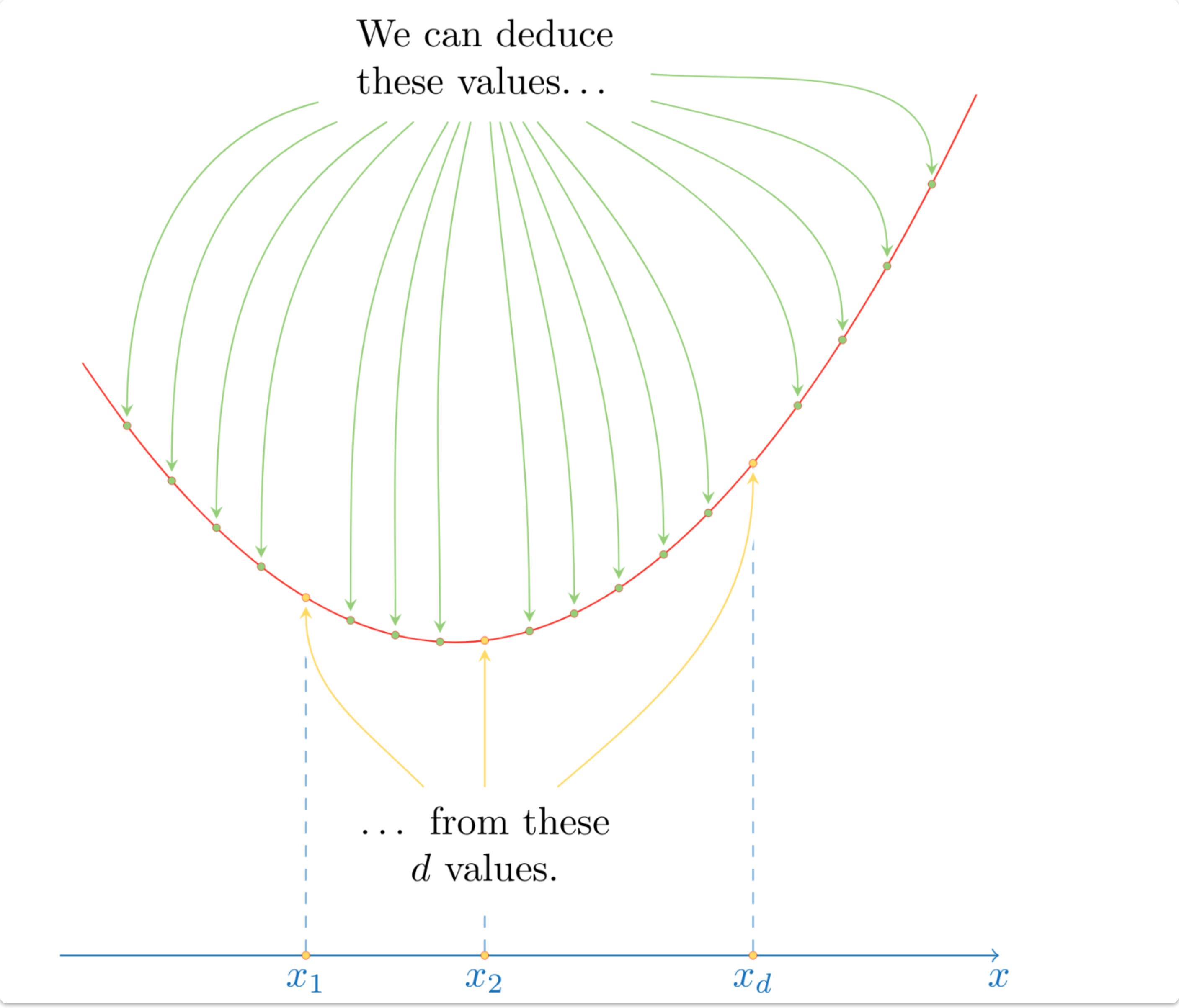
This extra data is calculated using polynomial arithmetic over a finite field.

Thus we have our message as coefficients in a polynomial of degree d
A = $a_0, a_1, . . a_{d-1}$

and we add redundancy by producing the evaluations of that polynomial at certain points $A(1), A(w), A(w^2)...A(w^{n-1})$ where $n >> d$

With FFTs we can quickly compute evaluations of polynomials

From https://hackmd.io/@olivierbbb/By1dILSV8



Computational Integrity

One of the (remarkable) features of zero knowledge proof systems is that they can be used to prove that some computation has been done correctly.
For example if we have a cairo program that is checking that a prover knows the square root of 25, they can run the program to test this, but the verifier needs to know that the computation was done correctly.

The issue of succinctness is important here, we want the time taken to verify the computation to be substantially less than the time taken to execute the computation, otherwise the verifier would just repeat the computation.

With the Starknet L2 we are primarily concerned that a batch of transactions has executed correctly giving a valid state change.
Participants on the L1, wish to verify this, without the need to execute all the transactions themselves.

In the context of Starknet, computational integrity is more important than zero

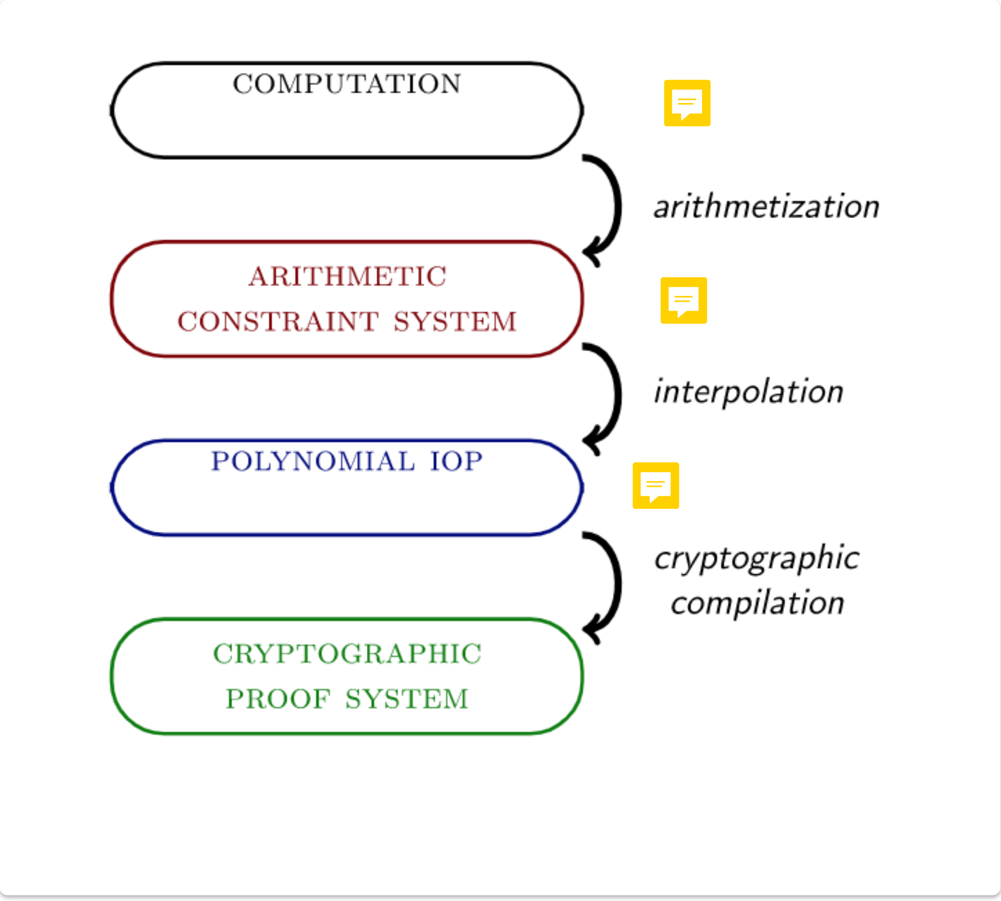
> knowledge, all data on Starknet is public.

Overview of the Stark process

Setup

Starks are transparent and therefore do not require an MPC ceremony to hide toxic waste.

The underlying cryptographic assumption is that we can use a collision resistant hash function.
This means that STARKS are quantum secure, see this answer from Stack Exchange for further details


COMPUTATION
arithmetization
ARITHMETIC CONSTRAINT SYSTEM
interpolation
POLYNOMIAL IOP
cryptographic compilation
CRYPTOGRAPHIC PROOF SYSTEM

We are interested in Computational Integrity (CI), for example knowing that the Cairo program you wrote was computed correctly.

We need to go through a number of transformations from the *trace* of our program, to the proof.
The first part of this is called arithmetisation, it involves taking our trace and turning it into a set of polynomials.

Our problem then becomes one where the prover that attempts to convince a verifier that the polynomial is of low degree.

The verifier is convinced that the polynomial is of low degree if and only if the original computation is correct (except for an infinitesimally small probability).

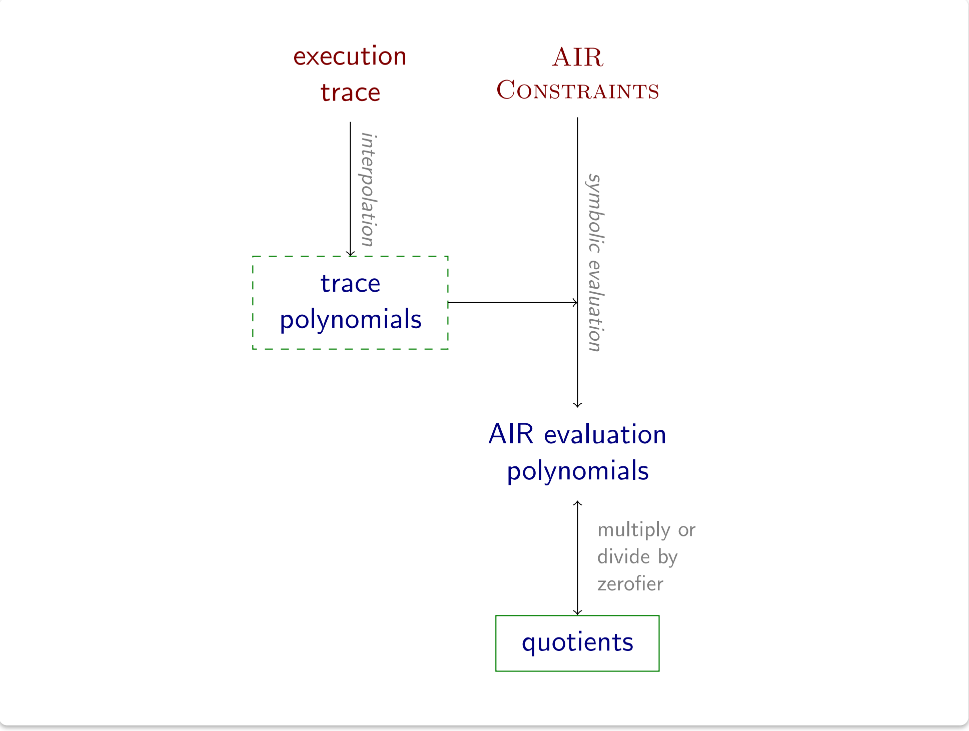
Compared to SNARK arithmetisation, in a STARK our model is closer to a virtual machine.

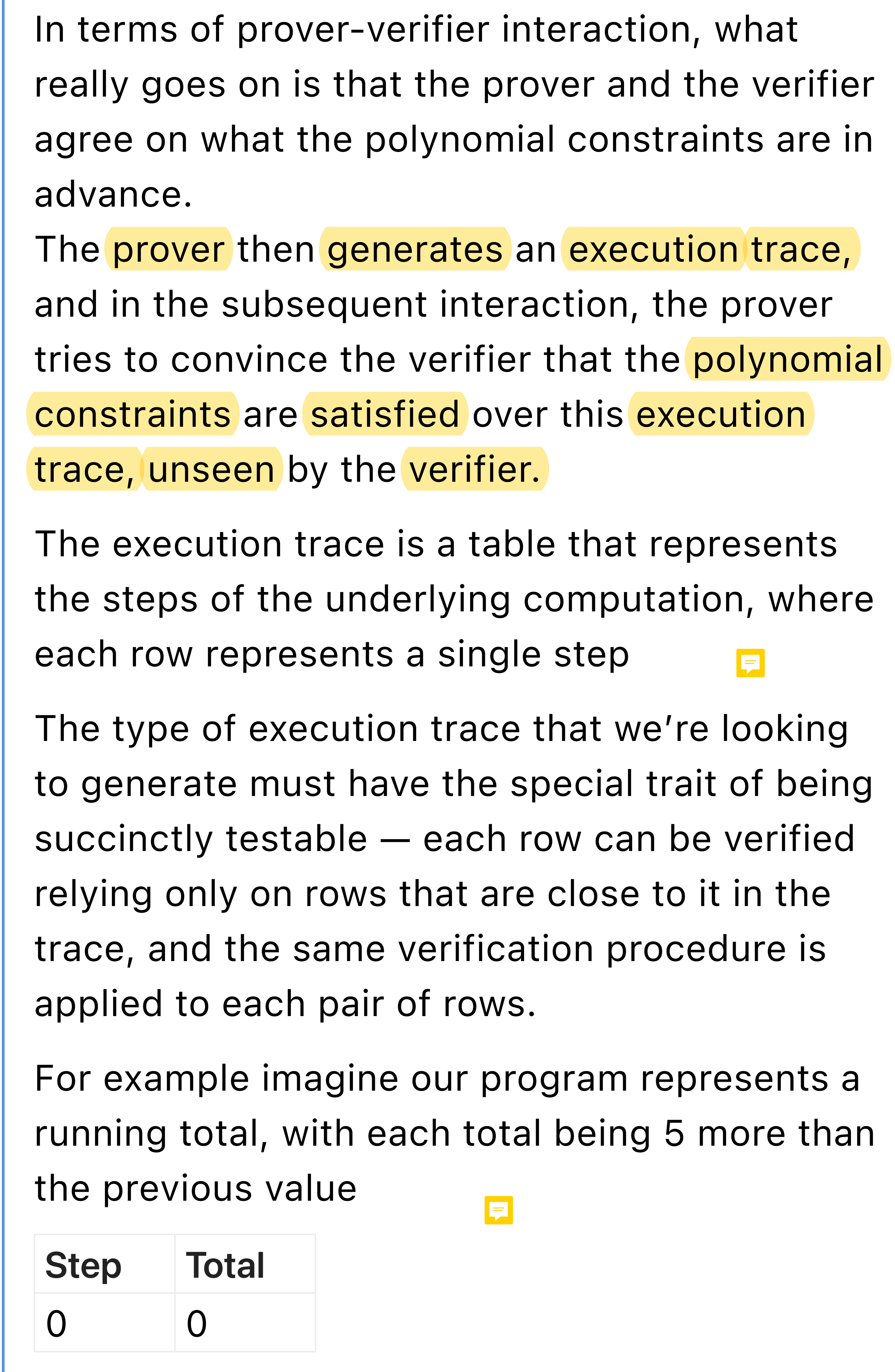
Arithmetisation

There are two steps

1. Generating an execution trace and polynomial constraints
2. Transforming these two objects into a single low-degree polynomial.

The *arithmetic intermediate representation (AIR)* (also, arithmetic *internal* representation) is a way of describing a computation in terms of an execution trace that satisfies a number of constraints induced by the correct evolution of the state.





In terms of prover-verifier interaction, what really goes on is that the prover and the verifier agree on what the polynomial constraints are in advance.
The prover then generates an execution trace, and in the subsequent interaction, the prover tries to convince the verifier that the polynomial constraints are satisfied over this execution trace, unseen by the verifier.

The execution trace is a table that represents the steps of the underlying computation, where each row represents a single step

The type of execution trace that we're looking to generate must have the special trait of being succinctly testable — each row can be verified relying only on rows that are close to it in the trace, and the same verification procedure is applied to each pair of rows.

For example imagine our program represents a running total, with each total being 5 more than the previous value

Step	Total
0	0

Step	Total
1	5
2	10
3	15
4	20
5	25

If we represent the row as i , and the column as j , and the values as $A_{i,j}$
We could write some contraints about this as follows

$$A_{0,2} = 0$$

$$\forall 1 <= i <= 5 : A_{i,1} - A_{i-1,1} - 5 = 0$$

$$A_{5,1} = 25$$

These are linear polynomial constraints in $A_{i,j}$
Note that we are getting some succinctness here because we could represent a much larger number of rows with just these 3 constraints.

We want a verifier to ask a prover a very small number of questions, and decide whether to accept or reject the proof with a guaranteed high level of accuracy.
Ideally, the verifier would like to ask the prover

to provide the values in a few (random) places in the execution trace, and check that the polynomial constraints hold for these places.

A correct execution trace will naturally pass this test.

However, it is not hard to construct a completely wrong execution trace (especially if we knew beforehand which points would be tested) , that violates the constraints only at a single place, and, doing so, reach a completely far and different outcome.

Identifying this fault via a small number of random queries is highly improbable.

But polynomials have some useful properties here

Two (different) polynomials of degree d evaluated on a domain that is considerably larger than d are different almost everywhere.

So if we have a dishonest prover, that creates a polynomial of low degree representing their trace (which is incorrect at some point) and evaluate it in a large domain, it will be easy to see that this is different to the *correct* polynomial.

Our plan is therefore to

1. Rephrase the execution trace as a polynomial
2. extend it to a large domain, and
3. transform that, using the polynomial constraints, into yet another polynomial that is guaranteed to be of low degree if and only if the execution trace is valid.

In general if our computation involves N steps, the execution trace will be represented by polynomials of degree less than N

$$f(X) = c_0 + c_1X + c_2X^2 + \cdots + c_{N-1}X^{N-1}$$

"The coefficients c_i are in the field F and the bound N on the degree is typically large, maybe of the order of a few million. Despite this, such polynomials are referred to as low degree. This is because the point of comparison is the size of the field.

By interpolation, every function on $\mathbb{F}$ can be represented by a polynomial.

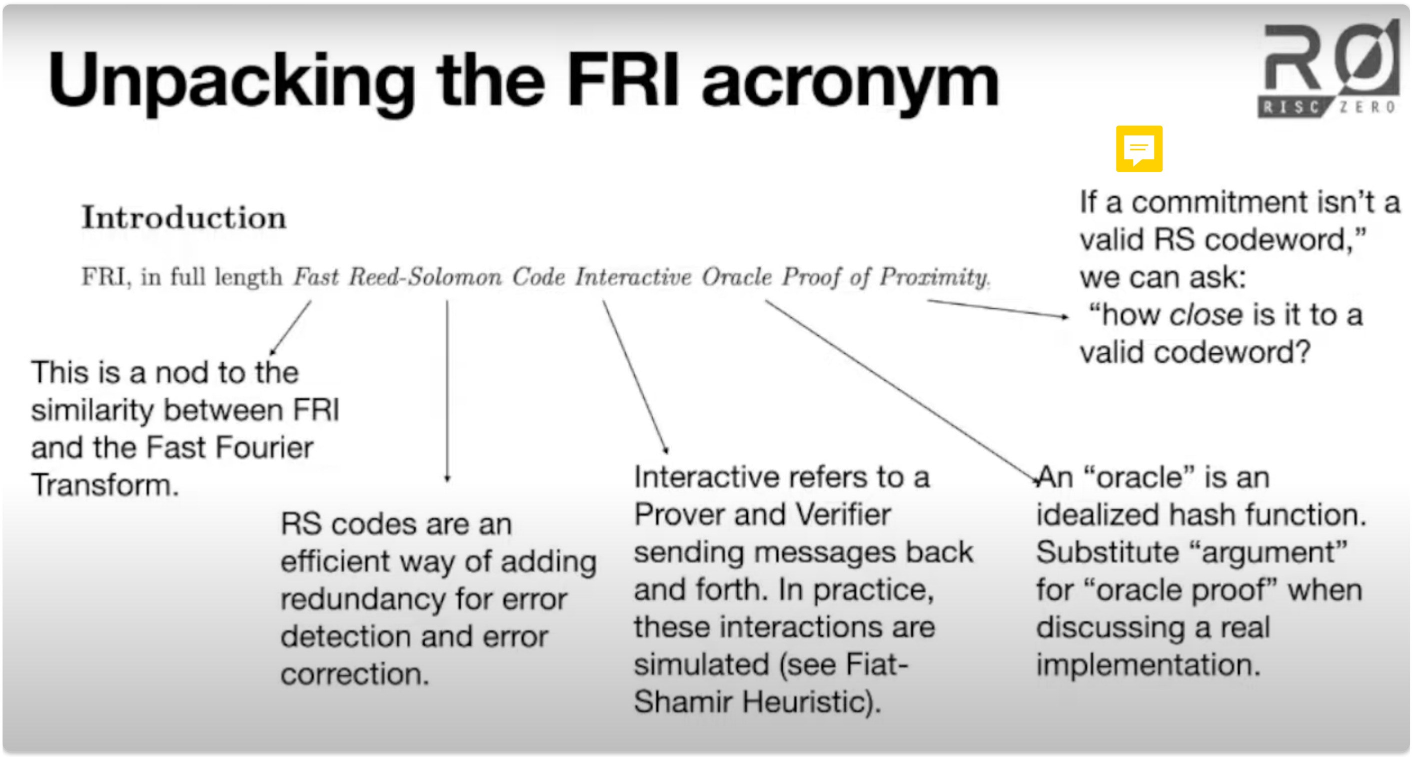
Most of these will have degree equal to the full

> size of the field so, compared to this, N is indeed low.

FRI

FRI stands for *Fast Reed-Solomon IOP of Proximity*, it is a protocol that establishes that a committed polynomial has a bounded degree.

From Risc Zero video



Unpacking the FRI acronym

Introduction

FRI, in full length *Fast Reed-Solomon Code Interactive Oracle Proof of Proximity*.

If a commitment isn't a valid RS codeword," we can ask: "how *close* is it to a valid codeword?

This is a nod to the similarity between FRI and the Fast Fourier Transform.

RS codes are an efficient way of adding redundancy for error detection and error correction.

Interactive refers to a Prover and Verifier sending messages back and forth. In practice, these interactions are simulated (see Fiat-Shamir Heuristic).

An "oracle" is an idealized hash function. Substitute "argument" for "oracle proof" when discussing a real implementation.

See also their overview

FRI tries to show that a vector commitment is close to a Reed-Solomon codeword
Relating this to the arithmetisation and to the trace we are trying to prove, in the arithmetisation we

- set up columns, each representing an item in a VM, for example a register

    - each row represents a clock cycle
- encode each column as a Reed-Solomon codeword. (called a trace block)

FRI is complex and much of the processing that makes it up is designed to make the testing feasible and succinct.
There is also much processing involved with guarding against various types of attacks that could be made by the prover, and ensuring that everything is carried out in zero knowledge.
It aims to find if a set of points are mostly on a polynomial of low degree and can achieve linear proof complexity and logarithmic verification complexity.

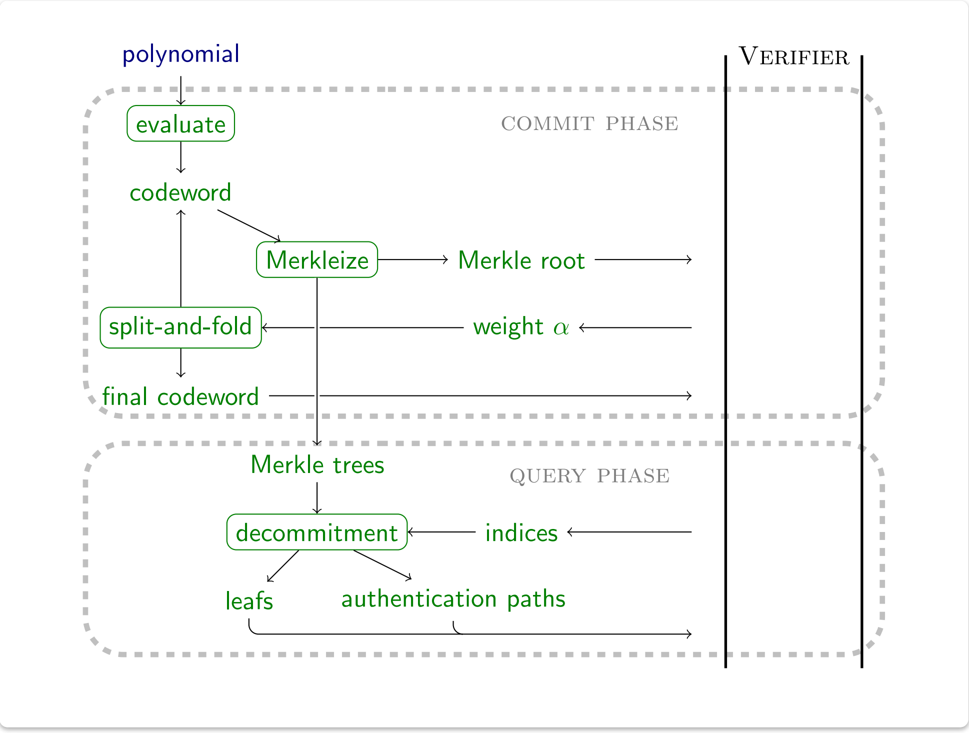
FRI is a protocol that establishes that a committed polynomial has a bounded degree. FRI is presented in the language of codewords: the prover sends codewords to the verifier who does not read them whole but who makes oracle-queries to read them in select locations. The codewords in this protocol are *Reed-Solomon codewords*, meaning that their values correspond to the evaluation of some low-

degree polynomial in a list of points called the domain
The length of this list is larger than the number of possibly nonzero coefficients in the polynomial by a factor called the *expansion factor* (also *blowup factor*), which is the reciprocal of the code’s *rate* ρ.

Since the codewords represent low-degree polynomials, and since the codewords are hidden behind Merkle trees in any real-world deployment, it is arguably more natural to present FRI from the point of view of a polynomial commitment scheme, with some caveats. There is scientific merit in separating the type of codewords from the IOP, and those two from the Merkle tree that simulates the oracles. However, from an accessibility point of view, it is beneficial to consider them as three components of one basic primitive that relates to polynomial commitment schemes. For the remainder of this tutorial, we will use the term FRI in this sense

FRI is a protocol between a prover and a verifier, which establishes that a given codeword belongs to a polynomial of low degree – low meaning at most ρ times the length of the codeword.

Here's a high-level overview of how the FRI protocol works:



1. **Commitment:** The prover, who wants to show that their points come from a low-degree polynomial, sends a "commitment" to the verifier. This commitment is a Merkle root of the polynomial's evaluations.
2. **Decomposition:** The prover then decomposes the polynomial into two polynomials of half the degree, evaluated at twice the spacing. This process is

repeated until a polynomial of constant degree is reached.

3. **Proof Generation:** The prover generates Merkle proofs for each level of decomposition. Each proof includes a random subset of points from the original polynomial and the corresponding points in the decomposed polynomials.
4. **Verification:** The verifier checks the Merkle proofs and uses them to ensure that the decomposed polynomials are consistent with the original polynomial. This process is repeated for each level of decomposition.

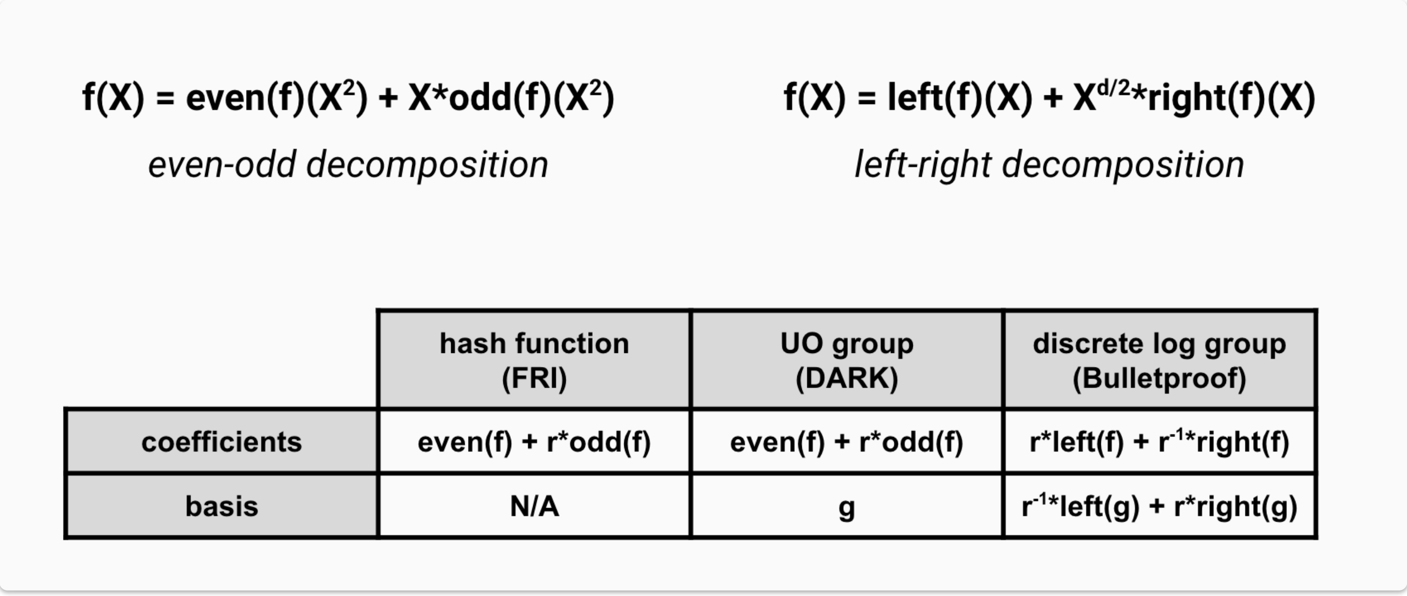
The main advantage of the FRI protocol is its efficiency. It allows for the verification of polynomial commitments with a logarithmic number of queries, making it very scalable.

It is explained in further detail in this article

Split and fold techniques

"One of the great ideas for proof systems in recent years was split-and-fold technique. The idea is to reduce a claim to two claims of half the size. Then both claims are merged into one

> using random weights supplied by the verifier. After many steps the claim has been reduced to one of a trivial size which is true if and only if (modulo some negligible security degradation) the original claim was true."
>
> The verifier inspects the Merkle trees (specifically: asks the prover to provide the indicated leafs with their authentication paths) in consecutive rounds to test a simple linear relation.
> For honest provers, the degree of the represented polynomials likewise halves in each round, and is thus much smaller than the length of the codeword.
>
> However for malicious provers this degree is one less than the length of the codeword. In the last step, the prover sends a non-trivial codeword corresponding to a constant polynomial.
>
> The splitting of the polynomial is done by splitting into even and odd terms
>
> Slide from ZK Study club



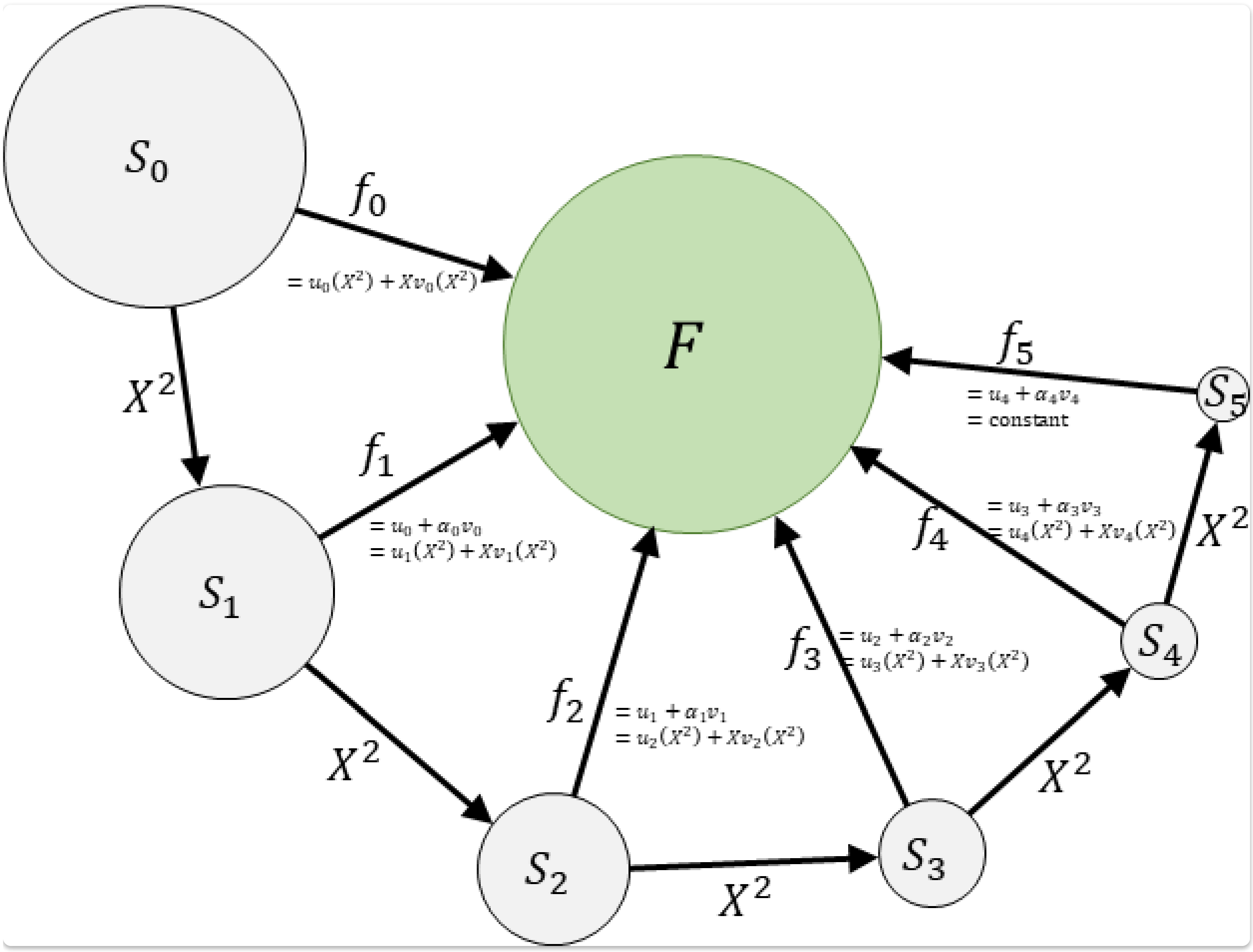
$f(X) = even(f)(X^2) + X*odd(f)(X^2)$

even-odd decomposition

$f(X) = left(f)(X) + X^{d/2}*right(f)(X)$

left-right decomposition

	hash function (FRI)	UO group (DARK)	discrete log group (Bulletproof)
coefficients	even(f) + r*odd(f)	even(f) + r*odd(f)	r*left(f) + r^{-1}*right(f)
basis	N/A	g	r^{-1}*left(g) + r*right(g)



For much more detail of the process and a python implementation see this article

Fiat Shamir Heuristic

The final part of the process is to make our proof non interactive, to understand the process I find thinking in terms of interaction helps, and the final part is just squashing this interaction into one step.
For this we use the ubiquitous Fiat Shamir Heuristic.

Public coin protocols

This terminology simply means that the verifier is using randomness when sending queries to the prover , i.e. like flipping a coin

Random oracle model

Both parties are given blackbox access to a random function

Fiat Shamir Heuristic

In an interactive process, the prover and a verifier engage in a multiple interactions, such as in the billiard ball example, to verify a proof. However, this interaction may not be convenient or efficient, with blockchains, we would like only a single message sent to a verifier smart contract to be sufficient.

The Fiat-Shamir heuristic addresses this limitation by converting an interactive scheme into a non-interactive one, where the prover can produce a convincing proof without the need for interactive communication.
This is achieved by using a cryptographic hash function.

The general process is

1. Setup: The verifier generates a public key and a secret key. The public key is made public, while the secret key remains private.
2. Commitment: The verifier commits to a randomly chosen challenge, typically by hashing it together with some additional data. The commitment is sent to the prover.
3. Response: The prover uses the commitment received from the verifier and its secret key to compute a response. The response is generated by applying the cryptographic hash function to the commitment and the prover's secret key.

> 4. Verification: The verifier takes the prover's response and checks if it satisfies certain conditions. These conditions are typically defined by the original scheme.
>
> The Fiat-Shamir heuristic is secure under the assumption that the underlying identification scheme is secure and the cryptographic hash function used is collision-resistant. It provides a way to convert interactive protocols into more efficient and practical non-interactive ones without compromising security.
>
> When the prover and verifier are interacting, there are a small number of queries from the verifier that would allow the prover to cheat, because we don't have complete soundness.
>
> In the Fiat Shamir heuristic we assume that the prover is computationally bounded (i.e this is an argument of knowledge as opposed to a proof).