STARK

Notes by Linus Tang.

These notes have not been thoroughly reviewed. Any errors below are my own responsibility.

Sources

- Vitalik Buterin's blog:
  - https://vitalik.eth.limo/general/2017/11/09/starks_part_1.html
  - https://vitalik.eth.limo/general/2017/11/22/starks_part_2.html
- aszepieniec's blog:
  - https://aszepieniec.github.io/stark-anatomy/stark
- Tiago Martins, João Farinha
  - https://eprint.iacr.org/2023/661.pdf
- Bobbin Threadbare
  - https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-airAssembly.pdf

Each of the two blogs has more parts than are listed above!

Prerequisite knowledge

Some necessary background knowledge:

- Interactive proofs
- Merkle trees
  - https://en.wikipedia.org/wiki/Merkle_tree
- Polynomial interpolation and some properties of polynomials in one variable
  - https://en.wikipedia.org/wiki/Polynomial_interpolation
- Modular arithmetic and Fermat's little theorem
  - https://en.wikipedia.org/wiki/Proofs_of_Fermat%27s_little_theorem

Contents

1. What is a STARK?

(zk)STARK: like (zk)SNARK, except

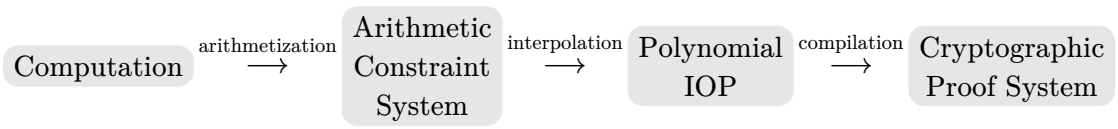
- it doesn't need a trusted setup
- the only security assumption is a secure hash is used
- the proof is not quite as succinct (short and fast to verify)

The T stands for "transparent" and means that no trusted setup is required.

Situation: the prover is about to perform a computation f on input x (where f is publicly known). When the prover finishes the computation and finds $f(x) = y$, the prover will want to convince the verifier that $f(x) = y$ (or in the zero knowledge setting, that the prover knows a secret x such that $f(x) = y$).

Below, all arithmetic is done over a finite field. It is convenient if the computation f is natively expressed in terms of operations over the same finite field. We use a prime field $\mathbb{F}_p$ for simplicity. (I have read that there are some advantages to working over a field $\mathbb{F}_{2^k}$ of characteristic 2 but not learned the details of this.)

2. Outline



The protocol can be broken down into three parts as in the diagram above. Here's a brief informal description of each:

Arithmetization:

- Think of the computation as being performed on a set of w registers, over T time steps. (Here, w and T will depend on the function f and on the details of its implementation as a circuit. Assume w is small.)
- Thus, the computation has a *trace*, which is a $w \times T$ array W consisting of the values of all registers over all time steps.
- Each time step, the registers should update in a simple way.
- The goal of arithmetization is to capture this simple update by some polynomial relations that should hold over register values over every pair of consecutive time steps (consecutive columns of W).
- These polynomial relations depend only on the publicly known function f being computed.

Interpolation

- The prover can perform a computation and obtain its trace W by recording the value of every register at every time step.
- The prover wants to be able to prove that W satisfies the polynomial constraints (which implies that the computation was done correctly), then reveal the particular index of the trace which holds the output y.
- For each register, if its value over the time steps are $a_0, ..., a_{T-1}$, it turns out to be helpful for the prover to do the following:[1]
  - Find the unique polynomial p of degree at most $T-1$ such that $P(i) = a_i$ for $i = 0, ..., T-1$.
  - Extend the sequence w according to this polynomial. That is, choose some $N \gg T$ and define $a_i = P(i)$ for $i = T, ..., N-1$.
- (The reason why this is helpful might not be obvious now but will become clear later!)

[1] For simplicity, we lie here about what inputs you interpolate over and will correct this lie later.

- In this way, the $w \times T$ array W of register values extends to a $w \times N$ array W' in which each row consists of the evaluations of a polynomial of degree at most $T - 1$.

Compilation

- The prover can commit to every value in W' by sending the verifier the root of a Merkle tree.
  - You can learn about Merkle trees here (https://en.wikipedia.org/wiki/Merkle_tree).
  - Alternatively, you can leave this as a black box, and assume the following:
    - The size of the commitment is constant
    - The verifier can query any index of the array, and the prover can send a $\log(N)$-size proof of the value at that index, which can be verified in $\log(N)$ hash evaluations.
- The prover wants to prove, by answering the verifier's queries, that the values in W' satisfy two properties:
  - Recall that consecutive columns of W satisfy certain polynomial constraints. These constraints translate to a property that W' must satisfy. Details of this are delayed until later.
  - The second property is that the N elements of each row of W' are the evaluations of some polynomial of degree at most $T - 1$ (as they should if interpolation was done correctly).
- In fact, we want the verifier to be able to do this with much fewer than T queries. After all, if the verifier had the bandwidth to send T queries, the verifier could probably just compute $f(x)$ directly (at least in the non-zk setting where x is public), since by definition that can be done in T time steps.
- It should be surprising that the verifier can be convinced of the second property in much fewer than T queries, because any T evaluations are consistent with some polynomial of degree $\leq T - 1$.

Roadmap

- In section 3, we look at a toy subproblem of compilation that motivates interpolation.
- In section 4, we learn how the prover can efficiently prove a degree bound (the "second property" mentioned in Compilation).
- In section 5, we discuss some aspects of arithmetization and establish what the outputs (polynomial constraints) of arithmetization look like.
- In section 6, we generalize the toy subproblem to accommodate said polynomial constraints.

3. A Motivating Subproblem (Vitalik Part 1)

Recall that in compilation, the prover sends a commitment encompassing every value in W'. The verifier can query for $\ll T$ entries of W' and should end up convinced of properties (1) and (2) of W':

- Property (1) depends on the polynomial constraints determined in arithmetization
- Property (2) is that the degree of each row is at most $T - 1$.

We neglect property (2) until the next section. For now, it may be helpful for us to pretend that the prover can efficiently prove a degree-check. That is, after the prover has committed to a sequence of numbers $a_0, ..., a_{N-1}$, if these numbers agree with some degree $\leq d$ polynomial, then the prover can efficiently convince the verifier that it is so.

(The degree-check proof as just described is not actually possible, but we will see that a weaker version suffices for our purposes, and we will demonstrate how to achieve this weaker degree-check in the next section.)

Consider the following toy problem which is used to motivate interpolation and show how constraints on W translate to constraints on W'. In this toy problem, W is a one-dimensional array instead of two-dimensional.

The Problem

Suppose that a prover has sent a Merkle root for numbers $W = (W_1, ..., W_{10^6})$ and wants to prove that it satisfies $W_x \in \{0, 1, ..., 9\}$ for all $x = 1, 2, ..., 10^6$. One way to do this is by letting the verifier query all 10^6 values W_x. How can we decrease the verifier complexity way below 10^6?

The Solution: Interpolation!

- Compute the degree $\leq 10^6 - 1$ polynomial P which agrees with W in the sense that $P(x) = W_x$ for $x = 1, ..., 10^6$. (We will ignore the "-1" in future degree analyses for simplicity.)
- Use P to extend W to $W' = (W_1, ..., W_{10^9}) = (P(1), ..., P(10^9))$ and send a commitment to the new values of W'.
- Let

$$C(z) = z(z-1)\cdots(z-9),$$

  which encodes the condition $z \in \{0, ..., 9\}$ to $C(z) = 0$. The prover now wants to show that $C(W_x) = C(P(x)) = 0$ for $x = 1, ..., 10^6$.
- Note that $C(P(x)) = 0$ is a polynomial constraint on x with degree $\leq 10^7$. We know that $x = 1, ..., 10^6$ are roots, so we let

$$Z(x) = (x-1)(x-2)\cdots(x-10^6)$$

  and have that $C(W_x) = C(P(x)) = D(x)Z(x)$ for some degree $\leq 9 \cdot 10^6$ polynomial D.
- Conversely, the existence of such a polynomial D convinces the verifier that the values W_i really do lie in $\{0, ..., 9\}$.
- The prover commits (via Merkle tree) to the evaluations of $D(x)$ on all inputs $x = 1, 2, ..., 10^9$.
- The verifier makes 20 random queries q, and for each, asks the prover to reveal the committed values W_q and $D(q)$. The verifier confirms that

$$C(W_q) = Z(q)D(q).$$

  - Objection: Since we want low verifier complexity, is it bad that the verifier needs the evaluations of $Z(q)$, where Z is defined as a product of 10^6 numbers?
  - Response: We assume that the verifier knows the evaluations of Z, which is a sort of "public verification key" for this scheme. The verifier either stores all evaluations of Z or stores the root of a Merkle tree and queries for the evaluations of Z at proof time.
- An honest prover will always pass every query (completeness). But why is the scheme sound?
  - Recall that we're assuming that the prover can quickly prove degree-checks. In this case, the prover can convince the verifier that the values W_x and $D(x)$ that were committed to in fact to belong to polynomials of degree $\leq 10^6$ and $9 \cdot 10^6$, respectively.
  - Thus, the verifier can trust that $C(W_x) - Z(x)D(x)$ is a polynomial of degree $\leq 10^7$ over x. In particular, either this polynomial is uniformly 0 (and the prover is honest) or the polynomial has at most 10^7 roots (and the prover gets caught whenever q is chosen to be any non-root). So, the probability of a dishonest prover surviving 20 rounds is a negligible $\left(\frac{10^7}{10^9}\right)^{20} = 10^{-40}$.

As mentioned earlier, a weaker degree-check is sufficient for our purposes. The specification of this weakening is as follows.

Proximity degree-check:

Let $N \gg d$.

- Completeness: If the prover commits to N values which are all consistent with some degree $\leq d$ polynomial, then the proof always verifies.
- Soundness: If the prover commits to N values such that no degree $\leq d$ polynomial is consistent with $> 90\%$ of them, then the proof verifies with negligible probability.

Why is this enough? Consider a dishonest prover who got away with committing to $\mathfrak{W}_x$ and $\mathfrak{D}(x)$ which differ from W_x and $D(x)$ (of appropriate degree) on up to 10% of inputs x. Then the prover survives query q if q is among those $\leq 10\%$ of disagreements between $\mathfrak{W}$ and W or between $\mathfrak{D}$ and D, or if q is one of the 10^7 aforementioned roots. We can get that the probability of a dishonest prover surviving 20 queries is at most $\left(10\% + 10\% + \frac{10^7}{10^9}\right)^{-20}$, still small.

4. Degree-check, or FRI (Vitalik Part 2)

Here, we describe exactly how to perform the above degree-check proof efficiently, using the FRI algorithm.

- FRI: Fast Reed-Solomon IOP of Proximity
- IOP: Interactive Oracle Proof.
- "Oracle": just means we assume that the verifier can access any of the N values that have been chosen by the prover. The Merkle tree mentioned earlier is one way to achieve such an oracle, though with a logarithmic overhead on each oracle access.
- "Proximity": refers to the fact that the verifier is only convinced that the N chosen values *mostly* agree with the evaluations of some low-degree polynomial.
- "Fast": The verifier complexity is polylogarithmic in d.

We again assign specific numbers to gain intuition more easily. Consider $d = 10^6 - 1$ and $N = 10^9$.

Suppose the prover has commited to 10^9 evaluations $(a) = (a_1, ..., a_{10^9})$ of a degree $\leq 10^6 - 1$ polynomial P, and wants to convince the verifier that there exists a degree $\leq 10^6 - 1$ polynomial which agrees with (a) at $\geq 90\%$ of the values in $\{1, ..., 10^9\}$.

Attempt 1

Since P has degree at most $10^6 - 1$, we can write $P(x)$ as $g(x, x^{1000})$, where g has degree at most 999 in each argument. Now, the prover hashes all 10^{18} evaluations of $g(x, y)$ for $x \in \{1, ..., 10^9\}$ and $y \in \left\{1^{1000}, ..., (10^9)^{1000}\right\}$ in a Merkle tree. We think of this as a square table of evaluations:

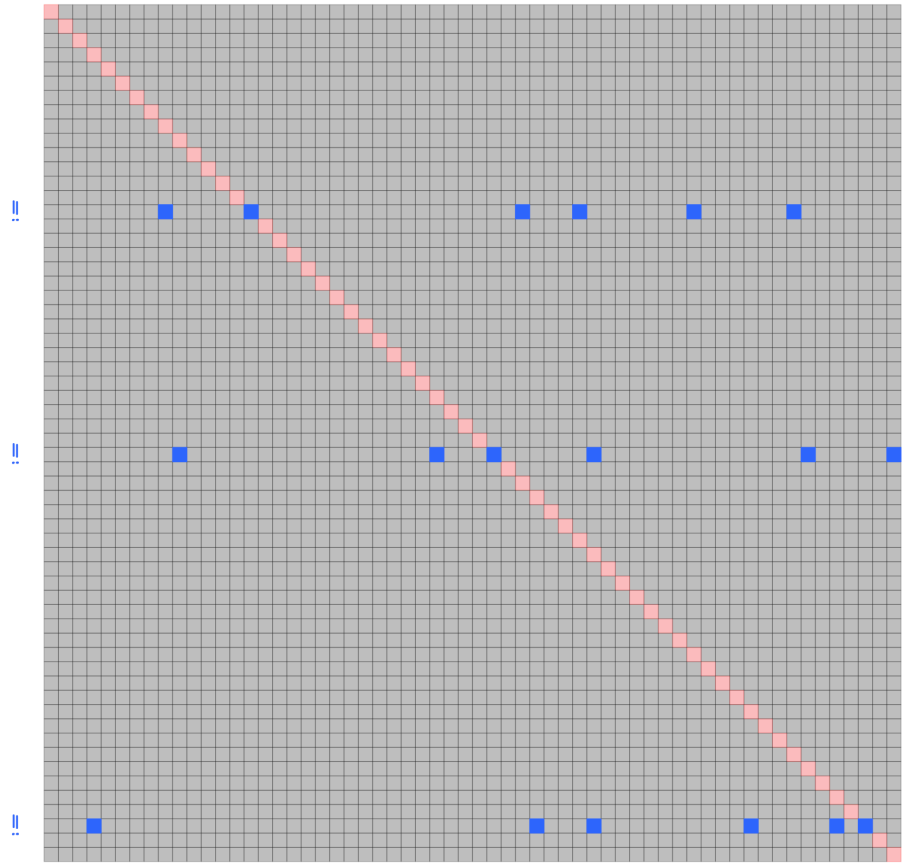
$g(1, 1^{1000})$	$g(2, 1^{1000})$	$g(3, 1^{1000})$	...	$g(10^9, 1^{1000})$
$g(1, 2^{1000})$	$g(2, 2^{1000})$	$g(3, 2^{1000})$	...	$g(10^9, 2^{1000})$
$g(1, 3^{1000})$	$g(2, 3^{1000})$	$g(3, 3^{1000})$	...	$g(10^9, 3^{1000})$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$g\left(1, (10^9)^{1000}\right)$	$g\left(2, (10^9)^{1000}\right)$	$g\left(3, (10^9)^{1000}\right)$	...	$g\left(10^9, (10^9)^{1000}\right)$

Note that the entries along the diagonal are $g(x, x^{1000}) = P(x)$, i.e. the original values which the the prover is performing the degree-check on.

The verifier can check that a row $g(\cdot, j^{1000})$ is likely almost-entirely degree ≤ 999 by querying for 1001 entries from that row and checking that they lie on a polynomial of degree ≤ 999. Similarly for a column $g(i, (\cdot)^{1000})$.

So, the verifier picks 30 random rows $g(\cdot, j_1^{1000}), ..., g(\cdot, j_{30}^{1000})$ and queries for 1001 entries from each, making sure that points in the same row lie on a degree ≤ 999 polynomial. The verifier also needs to check that the diagonal entries $g(j_1, j_1^{1000}), ..., g(j_{30}, j_{30}^{1000})$ lie on the respective polynomials, since the diagonal entries are the ones that the prover is trying demonstrate a degree-bound on.

The verifier then does the same for 30 random columns.



Above is a diagram of this protocol (except with smaller numbers, of course). The red cells hold the values $a_1, ..., a_{10^9}$, which the prover wants to show mostly lie on a degree $\leq 10^6 - 1$ polynomial. The gray cells are the other values which the prover commits to. The blue cells are the ones which the verifier queries for while performing 30 row checks. Column checks are not shown here for readability.

Informal soundness analysis: So far, we've described the scenario in terms of an honest prover, i.e. the values committed to are indeed evaluations $\left(g(i, j^{1000})\right)_{1 \leq i,j \leq 10^9}$ of a polynomial g with degree ≤ 999 in each variable. We consider a dishonest prover who instead commits to 10^{18} arbitrary values $\left(a_{i,j^{1000}}\right)_{1 \leq i,j \leq 10^9}$ such that $a_{1,1^{1000}}, a_{2,2^{1000}} ..., a_{10^9,10^{9000}}$ are not a $\geq 90\%$ match with the evaluations of any degree $\leq 10^6 - 1$ polynomial on $1, ..., 10^9$. Can such a dishonest prover trick the verifier with nonnegligible probability?

- For almost all rows $a_{\cdot,j^{1000}}$, there should be a polynomial row_j of degree ≤ 999 such that for almost all $1 \leq i \leq 10^9$, $\text{row}_j(i) = a_{i,j^{1000}}$, otherwise the prover will very likely fail the row check. Furthermore, $i = j$ must be among these "almost all", since the verifier checks that the diagonal entries lie on their respective row polynomials.
- Similarly, for almost all columns $a_{i,(\cdot)^{1000}}$, there should be a polynomial col_i of degree ≤ 999 such that for almost all $1 \leq j \leq 10^9$, necessarily including $j = i$, we have $\text{col}_i(j^{1000}) = a_{i,j^{1000}}$.
- It turns out (mathematical exercise!) that the existence of almost-correct row and column polynomials row_j and col_i of degree ≤ 999 implies the existence of a almost-correct table polynomial, i.e. a

two-variable polynomial t of degree ≤ 999 in each variable such that $t(i, j^{1000}) = a_{i,j^{1000}}$ for almost all i, j. Furthermore, these include almost all i, j with $i = j$ since the almost all row and column polynomials are correct on their diagonal elements.

- Thus, $a_{i,i^{1000}} = t(i, i^{1000})$ is close to a polynomial in i of degree $\leq 10^6 - 1$.
- We conclude that any prover who can pass verification with nontrivial probability must have committed to diagonal elements that are close to a degree $\leq 10^6 - 1$ polynomial, as desired.

Efficiency analysis This strategy achieves a sublinear number $O\left(\sqrt{d}\right)$ of queries from the verifier (about 60000 in this case), but has huge overhead for the prover (hashing $N^2 = 10^{18}$ polynomial evaluations). We show two tricks that reduce the verifier complexity to polylogarithmic in d and the prover overhead to linear in N. The first trick is a carefully-chosen field, and the second is recursion.

Improvement 1

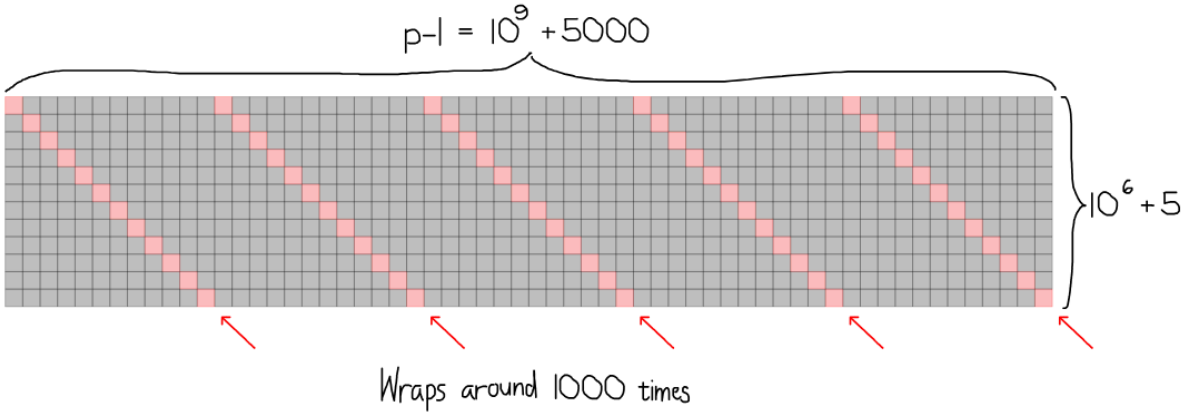
Continue working with $d = 10^6 - 1$ and $N \approx 10^9$. Work in the prime field $\mathbb{F}_p$, where $p = 10^9 + 5001$ is prime, and for convenient let $N = p - 1$. The key fact is that x^{1000} takes only $10^6 + 5$ distinct values over nonzero $x \in \mathbb{F}_p$, and each is achieved by exactly 1000 values of x, by Fermat's little theorem.

As a result, while the table of values the prover needs to commit is a $p - 1$ by $p - 1$ table (p close to 10^9), it only has $10^6 + 5$ distinct rows.

We reorder the rows and columns to be indexed by $s^0, s^1, \ldots, s^{p-2}$ instead of $1, 2, \ldots, p - 1$, where s is a generator of $\mathbb{F}_p^\times$:

$g\left(s^0, (s^0)^{1000}\right)$	$g\left(s^1, (s^0)^{1000}\right)$	$g\left(3, (s^0)^{1000}\right)$	$\ldots$	$g\left(s^{p-2}, (s^0)^{1000}\right)$
$g\left(s^0, (s^1)^{1000}\right)$	$g\left(s^1, (s^1)^{1000}\right)$	$g\left(s^2, (s^1)^{1000}\right)$	$\ldots$	$g\left(s^{p-2}, (s^1)^{1000}\right)$
$g\left(s^0, (s^2)^{1000}\right)$	$g\left(s^1, (s^2)^{1000}\right)$	$g\left(s^2, (s^2)^{1000}\right)$	$\ldots$	$g\left(s^{p-2}, (s^2)^{1000}\right)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$g\left(s^0, (s^{p-2})^{1000}\right)$	$g\left(s^1, (s^{p-2})^{1000}\right)$	$g\left(s^2, (s^{p-2})^{1000}\right)$	$\ldots$	$g\left(s^{p-2}, (s^{p-2})^{1000}\right)$

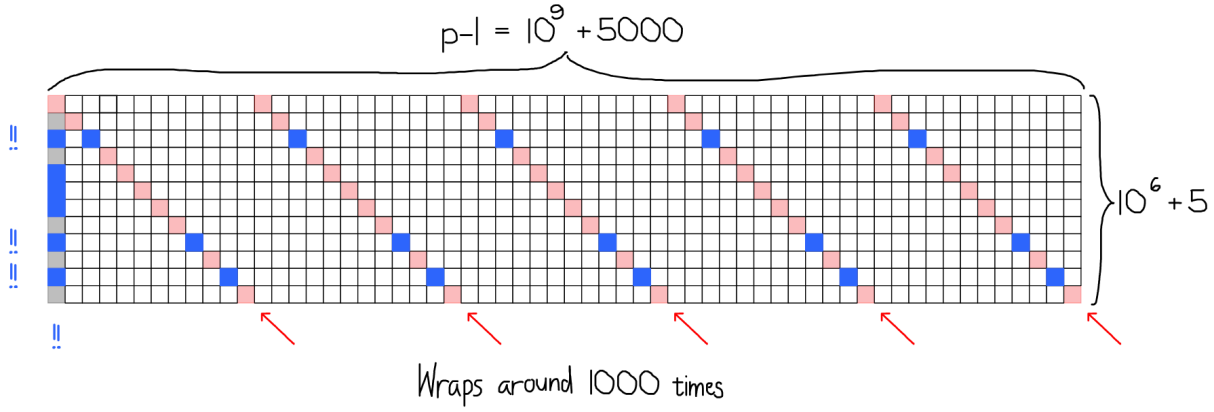
This is good for geometric intuition, because we can check that the reordered table repeats itself every $10^6 + 5$ rows (since $\left(s^{10^6+5}\right)^{1000} = 1$ in $\mathbb{F}_p$). In particular, the prover only needs to commit to the first $10^6 + 5$ rows of the table. The new diagram (before the verifier sends queries) looks like this:



If we do everything the same as before, we achieve similar verifier complexity but the prover only has to hash 10^{15} polynomial evaluations instead of 10^{18}. But we can reduce verifier complexity even further.

Note that each row in the table of committed values intersects the red "diagonal" 1000 times. Therefore, when performing a row check, the verifier can query for the values at these 1000 intersections, plus one more. In fact, this means that the prover only needs to commit to the diagonal and one additional column of the table! The protocol is as follows:

- The prover commits to polynomial evaluations on the diagonal and one column
- The verifier chooses a few random rows. For each:
  - The verifier queries the evaluations of the 1000 intersections of the row and diagonal and computes the unique degree ≤ 999 polynomial with these evaluations.
  - The verifier queries for the intersection of the row and column and verifies that
- The verifier makes 1001 random queries on the column to check that the column has degree 999.



We gloss over the soundness analysis here but claim that any prover that passes these tests with non-negligible probability must have assigned diagonal values that are almost all in alignment with some degree $10^6 - 1$ polynomial.

Efficiency analysis: Since the prover only commited to $\approx 10^9 + 10^6$ values, the overhead is linear in $N \approx 10^9$ and pretty much as small is it can be. The verifier complxity is lower than before but still $O\left(\sqrt{d}\right)$. We can make verifier complexity logarithmic by making the algorithm recursive.

Improvement 2

Continue working with $d = 10^6 - 1$ and $N = p - 1 \approx 10^9$.

Instead of writing $p(x) = g(x, x^{1000})$ with degree 999 in each argument, write $p(x) = g(x, x^2)$ linear in the first argument and degree $\frac{d-1}{2} = 499999$ in the second argument.

As before, the prover sends commitments to the diagonal and one column $g(1, (\cdot)^2)$ of the table.

Since each row should be linear, the verifier only needs to check 3 points from each of several randomly chosen rows. Two of these three points will come from the diagonal, and the third will come from a column randomly chosen by the verifier. The tradeoff is that the degree-check for the column now has degree $\frac{10^6}{2} - 1$ instead of 999.

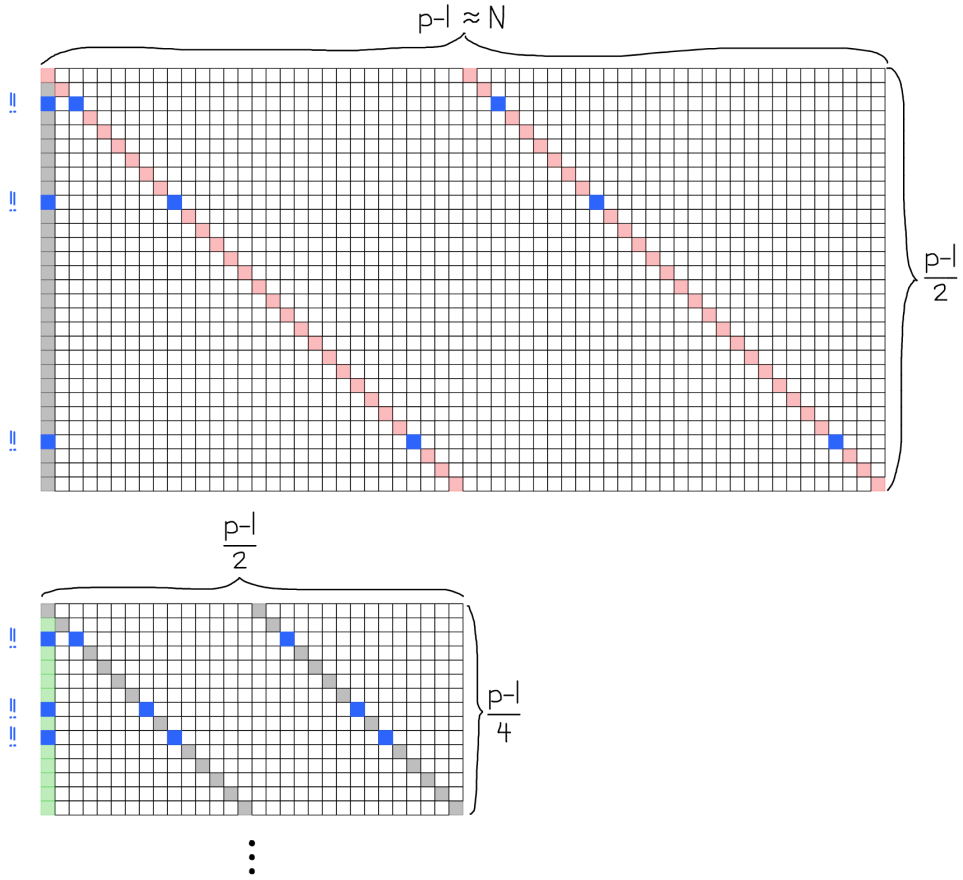
This is where recursion comes in: The prover now wants to show that $g(1, x^2)$ has degree $\leq \frac{d-1}{2}$ over its second input, which is restricted to the quadratic residues in $\mathbb{F}_p$. This is simply another instance of the degree-check, so in a similar manner, the prover determines a polynomial h of degree $\leq \frac{d-3}{4}$ such that $g(1, x^2) = h(x^2, x^4)$, and uses it to construct a smaller table, where the column of the original table becomes the diagonal of the smaller table. And so on!

To clarify, the protocol goes like this:

- The prover commits to the diagonal entries $g\left(s^i, (s^i)^2\right)$.
- The verifier picks a random column indexed by c, and the prover commits to the entries $g\left(s^c, (s^i)^2\right)$.

- The verifier picks a few random rows, indexed by r, and for each, ensures that the values $g\left(s^r, (s^r)^2\right)$, $g\left(-s^r, (-s^r)^2\right)$, and $g\left(s^c, (s^r)^2\right)$ are consistent with the claim that g is linear in its first argument.
- Recursively, the prover demonstrates that the column $g\left(s^c, (s^i)^2\right)$ is consistent with the claim that g has degree $< \frac{d-1}{2}$ in its second argument.

The diagram below is slightly flawed in that it suggests that the first column of the table is always the one audited. The truth is that the audited column is chosen randomly by the verifier after the red values have been commited to.



Some details:

- We now want to choose a prime p such that $p-1$ is divisible by a large power of 2 so that we can do many layers of recursion.
- For similar reasons we also want $d+1$ to be divisible by a large power of 2. If it is not, the verifier can just round the target degree d up to the nearest number of the form $d' = 2^k - 1$ and has a strategy to "pad" the degree from d to d' by multiplying entries by $x^{d'-d}$ (further details left as an exercise).

One issue with the above protocol is that it forced us to choose $N = p - 1$. This means that if we want to support native arithmetic over $\mathbb{F}_p$ for a prime $p \approx 2^{64}$, for exmple, then we'd have $N \approx 2^{64}$, which is unacceptably large. There is a simple modification to the above that helps us choose any $N|p-1$ so that we can support native arithmetic over large fields without having N comparably large. Namely, instead of making s a generator of $\mathbb{F}_p$, choose s to be an element of order N.

The prover overhead (measured by polynomial evaluations hashed) is still linear in N, and verifier complexity is now polylogarithmic! Yay!

5. Arithmetization (Aszepieniec Part 4)

We now discuss arithmetization. It may be helpful to reread the brief overview of arithmetization in section 2.

Think of the computation of a function of execution of a program as the evolution of a state over T time steps (cycles). The state consists of w registers, each of which holds a number in $\mathbb{F}_p$. The evolution is dictated by the state transition function $f : \mathbb{F}_p^w \to \mathbb{F}_p^w$ (i.e. if the state at one cycle is x, then the state at the next cycle is $f(x)$). A list of boundary conditions $\mathcal{B} \in \mathbb{Z}_T \times \mathbb{Z}_W \times \mathbb{F}_p$ enforces correct values of some registers of the program at some cycles, especially the first cycle.

A witness to the computation is a $W \in \mathbb{F}_p^{T \times w}$ that is consistent with the state transition function and boundary conditions, i.e.

- For all $i \in \{0, 1, ..., T-2\}$ we have $f\left(W_{[i,:]}\right) = W_{[i+1,:]}$.
- For all $(i, v, e) \in \mathcal{B}$ we have $W_{[i,v]} = e$.

The state function hides a lot of complexity. We would be satisfied if we could write it as a low degree polynomial that does not depend on the cycle number. However, as a toy example, even an unrealistically simple state transition rule $\mathbb{F}_p \to \mathbb{F}_p$ given by

$$x \mapsto \begin{cases} \frac{1}{x} \text{ if } x \neq 0 \\ 0 \text{ if } x = 0 \end{cases}$$

must be written as a degree $p-2$ polynomial $x \mapsto x^{p-2}$.

Instead of directly enforcing the transition rule $f\left(W_{[i,:]}\right) = W_{[i+1,:]}$, we can encode it using low degree constraints. For example, the above inversion can be handled by enforcing that

$$\boldsymbol{p}(x, y) = (p_1(x, y), p_2(x, y)) = (x(xy-1), y(xy-1)) = (0, 0),$$

where x is the state before the transition and y is the state after the transition.

Moving out of the toy example and back to the main problem, our goal is to find a state-transition function $f : \mathbb{F}_p^w \to \mathbb{F}_p^w$ that governs correct computation, and a not-too-large set of low-degree polynomials $p_1, ..., p_k$ such that

$$f(x) = y \Leftrightarrow \boldsymbol{p}(x, y) = \boldsymbol{0},$$

where we use $\boldsymbol{p}$ as a shorthand for the collection of polynomials $(p_1, ..., p_k)$. Then we can check that a witness satisfies $\boldsymbol{p}\left(W_{[i,:]}, W_{[i+1,:]}\right) = \boldsymbol{0}$ for each i.

For more details on arithmetization and on how to construct the polynomial constraints $\boldsymbol{p}$ for a broad class of computations:

- https://eprint.iacr.org/2023/661.pdf (mathematical details)
  - Read section 3
- https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-airAssembly.pdf (low-level language implementation details)
  - Read sections 2 and 3

To summarize:

- The trace of a computation is the array $W \in \mathbb{F}_p^{T \times w}$ consisting of the values of w registers over T time steps.
- Boundary constraints

$$(\forall (i,v,e)\in\mathcal{B})\Big(W_{[i:v]}=e\Big)$$

and low-degree polynomial constraints

$$\boldsymbol{p}\Big(W_{[i,:]},W_{[i+1,:]}\Big)=\mathbf{0}$$

enforce that the computation was done correctly (and that its trace was generated correctly).

- In particular, $\boldsymbol{p}$ is a small collection of polynomials $p_1,\ldots,p_s$ in $2w$ variables. The trace is correct if and only if it satisfies the boundary constraints and $p_1,\ldots,p_s$ all vanish when fed the $2w$ values of registers over every pair of consecutive time steps.

6. Putting it together

Through arithemization, we've reduced the problem of checking a computation $f(x)=y$ to verifying that a witness $W\in\mathbb{F}_p^{T\times w}$ satisfies $\boldsymbol{p}\Big(W_{[i,:]},W_{[i+1,:]}\Big)$ for each $i\in\{0,1,\ldots,T-2\}$.

- The constraints that the input is x and the output is y can be encoded by boundary conditions $(\forall (i,v,e)\in\mathcal{B})\Big(W_{[i:v]}=e\Big)$, and these boundary conditions are easy to prove as we will see later.
- If the prover wants to hide x and maintain some zero-knowledge properties, they simply relax the boundary conditions.

Now suppose that a prover has run a computation and honestly generated a witness W. In order to prove to the verifier that the computation has run correctly, the prover does the following:

- Choose some $N\gg T$ such that $N|p-1$ and N is divisible by a large power of 2.
- Let $\{s^0,\ldots,s^{N-1}\}$ be a subgroup of $\mathbb{F}_p^\times$.
- Use polynomial interpolation to find degree $\le T-1$ polynomials $P_1,\ldots,P_w$ such that $P_j(s^i)=W_{i,j}$ for $i=0,\ldots,T-1$.[2]
- Compute the wN evaluations $W'_{i,j}=P_j(s^i)$ for $i=\{0,\ldots,N-1\}$.
- Hash the entries of W' in a Merkle tree, and send the verifier the root.
- Since W is a witness of the computation, it ought to satisfy the boundary constraints

$$\boldsymbol{p}(\boldsymbol{P}(s^i),\boldsymbol{P}(s^{i+1}))=\boldsymbol{p}\Big(W_{[i,:]},W_{[i+1,:]}\Big)=\mathbf{0}$$

  for all $i\in\{0,1,\ldots,T-2\}$. Letting $t=s^i$, the left side is a collection polynomials in t, so of these polynomials ought to be divisible by $Z(t)=(t-s^0)(t-s^1)\cdots(t-s^{T-2})$.
- So, the prover hashes the evaluations (on $t\in\{s^0,\ldots,s^{N-1}\}$) of

$$\boldsymbol{D}(t)=\frac{\boldsymbol{p}(\boldsymbol{P}(t),\boldsymbol{P}(st))}{Z(t)}$$

  in a Merkle tree and sends the root to the verifier. Now each $D_j(i)$ should be a polynomial of degree $\le\big(\deg(p_j)-1\big)(T-1)$.

Remember that $\boldsymbol{p}$ and Z are known to the verifier, and assume that the verifier has oracle access to the evaluations of both.

Now, the verifier can do the following to be convinced of the correctness of the computation and trace:

- Ask the prover to open certain evaluations to verify the boundary constraints.
- Choose a few random $t\in\{0,\ldots,N-1\}$, and for each, ask the prover to open $P_j(t)$, $P_j(st)$, and $D_j(t)$ for each j.
  - If the prover wants zk properties, they may forbid the verifier from choosing $t\in\{s^0,\ldots,s^{T-1}\}$.

[2] Finally correcting the earlier lie: We interpolate over inputs $s^0,\ldots,s^{T-1}$ instead of $0,\ldots,T-1$.

- For each, the verifier should check that

$$\boldsymbol{p}(\boldsymbol{P}(t), \boldsymbol{P}(st)) = \boldsymbol{D}(t)Z(t).$$

- Then the verifier should check that the supposed evaluations of $\boldsymbol{P}$ and $\boldsymbol{D}$ are indeed consistent with polynomials of degree $T-1$ and $\big(\deg\big(p_j\big)-1\big)(T-1)$, respectively, using the protocol developed section 4.

Thus, the verifier can be convinced that W is a correct receipt for the computation, in complexity polylogarithmic in T, the number of time steps that it took the prover to run the computation.

Making the interactive proof non-interactive

Instead of waiting for a verifier's random challenges, the prover can use a hash of the most recent commitments as a source of (pseudo)randomness. The prover can run the entire proof, then publish it for verifiers to check at their leisure.

This is called the Fiat-Shamir heuristic and is a general tool for turning interactive proofs into non-interactive proofs.

For a more detailed overview: https://en.wikipedia.org/wiki/Fiat%E2%80%93Shamir_heuristic

Applying the Fiat-Shamir to the above interactive protocol, we have achieved (zk)STARK!