CSE303 Report - Succint non-interactive proofs for verifiable computing using STARK systems

Amine Abdeljaoued under the supervision of Sarah Bordage

December 2020

Contents

1 Motivation

Blockchain technologies rely on several principles. Among the most important ones, there are the scalability and the privacy. Here, scalability refers to the ability to adapt to a larger throughput in order to assess the computational integrity of transaction histories. Privacy is about managing to do that while not giving everyone full access to transactions details. In most fields, rigorously ensuring the computational integrity of systems to the user/client is not considered essential. We **trust** our banks and financial systems. This is where Blockchains offer a revolutionary alternative.

In Blockchains, computations are verified for every transaction. A naive verification would yield privacy problems through an access to all elements of the computation, and scalability problems through a lack of efficiency. STARKs (Succint transparent arguments of knowledge) offer a highly efficient solution to these challenges.

2 Succint transparent arguments of knowledge (STARKs)

2.1 What are STARKs

The succint transparent arguments of knowledge (STARKs) are interactive proofs and consist of an argument of knowledge between a Prover and Verifier, ensuring two fundamental properties:
1) Completeness: The Prover can prove anything that's right and convince the Verifier.
2)"Quasi"-Soundness: If the Prover doesn't know one step in the proof it constructs but still emits it, the Verifier will notice the inconsistency. "Quasi" because the Verifier can accept a wrong proof, but with a tiny probability [2].

The Prover's running time scales "quasi-linearly" with respect to the naive verification where we would just redo the computation. The Verifier's running time is polynomial in the logarithm of the naive method's time. It means that we can add a proof to all transactions in a linear time to just redoing the computation, but most importantly with a Verifier's running time decreasing exponentially in its verification time, ensuring a very good scalability for an increasing throughput.
In STARK systems, nothing is trusted to be true, the Verifier being suspicious and trying to fail the Prover's proof by rigorously analyzing each detail, allowing the second fundamental property stated above to be held true. They ensure zero knowledge through making private inputs not accessible.

2.2 How do they work

The central idea of STARK proofs is to reduce a problem to polynomial constraints. To illustrate the functioning of the method, we will take the example of Lucas sequences. A Lucas sequence $U(P,Q)$ is defined as follow:

$$\begin{cases} U_0(P,Q) = 0 \\ U_1(P,Q) = 1 \\ U_n(P,Q) = PU_{n-1}(P,Q) - QU_{n-2}(P,Q) \quad \forall n \geq 2 \end{cases} . \tag{1}$$

Let's imagine that we want to verify the following computational statement: *There exists elements P, Q such that the 15th element (U_{14}) of the sequence is 409593865.* The first step, called arithmetization [4] [5], is to translate this problem into verifying that a polynomial is of low-degree.

We start by defining the execution trace, which is the trace of all the computations executed in order to get (U_{14}). Here, the trace corresponds to all previous elements of the sequence: $U_0, U_1, ..., U_{14}$. We will denote by a_i the i-th element of the trace. In other cases, the trace might be less implicit and one needs to figure out all the steps achieved in order to get the computations.

Once the execution trace is well defined, one should establish the constraints associated to these computations. For our example, the constraints are inherent to the definition of our sequence and could therefore be formulated as follow:

1. $a_0 = 0$
2. $a_1 = 1$
3. $a_n = P \cdot a_{n-1} - Q \cdot a_{n-2}, \forall\ 2 \leq n \leq 14$

Once this has been established, we have to translate these constraints into a polynomial. For that, we need to see our execution trace as the evaluation of polynomials on a subgroup G of a field $\mathbf{Z}/p\mathbf{Z}$ for a sufficiently large integer prime p. The group G should have the size of the execution trace and we consider a generator g of this group. We now translate our execution trace to polynomials as follow: each element of the trace is an evaluation of a polynomial f of degree less than $|G| = 15$ such that a_i corresponds to $f(g^i)$. This gives the following constraints:

1. $f(x)$ evaluates to 0 for $x = g^0 = 1$
2. $f(x) - 1$ evaluates to 0 for $x = g^1$
3. $f(x) - 409593865$ evaluates to 0 for $x = g^{14}$
4. $f(g^2 \cdot x) - P \cdot f(g \cdot x) + Q \cdot f(x)$ evaluates to 0 for $x \in G \backslash \{g^{15}, g^{14}, g^{13}\}$

The next step is to consider a fundamental property of polynomials, which is that for a polynomial $p(x)$, $p(x_0) - y = 0 \iff x_0$ divides $p(x) - y \iff \frac{p(x)-y}{x-x_0}$ is a polynomial. This is how we get the following constraints:

1. $p_1(x) = \dfrac{f(x)}{x-1}$ is a polynomial
2. $p_2(x) = \dfrac{f(x)-1}{x-g}$ is a polynomial
3. $p_3(x) = \dfrac{f(x)-409593865}{x-g^{14}}$ is a polynomial
4. $$p_4(x) = \frac{f(g^2 \cdot x) - P \cdot f(g \cdot x) + Q \cdot f(x)}{\prod_{i=0}^{12}(x-g^i)} = \frac{f(g^2 \cdot x) - P \cdot f(g \cdot x) + Q \cdot f(x)}{\frac{x^{16}-1}{(x-g^{15})(x-g^{14})(x-g^{13})}}$$
   is a polynomial

At this point, one knows that checking the above constraints is analogous to checking the computational integrity of the execution trace and therefore of the computation. To that end, the Prover creates what's called a composition polynomial, using random values $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ provided by the Verifier. The composition polynomial is then defined as follow: $p = \alpha_2 p_1 + \alpha_2 p_2 + \alpha_3 p_3 + \alpha_4 p_4$ and if p is a polynomial of degree < 16 ,we know that there is a high probability that our 4 above constraints will be verified. The Prover then evaluates this composition polynomial on the evaluation domain and commits the root R of the Merkle Tree [1] this data generates.

The protocol to verify that p is a polynomial of low-degree is called the Fast Reed-Solomon Interacle Proof of Proximity (FRI). Further development can be find in [6] but an important part of it lies in the interaction between the Prover and the Verifier. In this protocol, the Verifier will ask for the

[1]Merkle Tree: "A tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes", source Wikipedia

evaluation of p on random points of the evaluation domain. The Prover will provide this value, with something called a Merkle proof, that will allow the Verifier to be sure that the value is coherent with the root R previously committed. A detailed explanation of these Merkle Trees and Merkle proofs can be found in the next section.

3 Implementation

My implementation can be seen at `https://github.com/amine-abdeljaoued/CSE303` and has been inspired by the online tutorial [3].
In particular, the jupyter notebok containing the pipeline of the proof at:
`https://github.com/amine-abdeljaoued/CSE303/blob/main/implementation.ipynb`.
The MerkleTree implementation at:
`https://github.com/amine-abdeljaoued/CSE303/blob/main/Merkle.py`.

3.1 Jupyter notebook

For this implementation, I used SageMath to manipulate fields, groups, generators and polynomial rings. It also provided a useful function for Lagrange interpolation on a defined polynomial ring.

For the last part, I did not implement the FRI protocol and used a simpler method to check that the composition polynomial was of low degree. To prove that p is a polynomial of degree less than 16, we evaluate it on 16 random points and interpolate these values to get p_interp. Now, we know that if p was indeed a polynomial of degree less than 16, it would evaluate to p_interp for any random point. This is why we compare $p(x)$ and $p\_interp(x)$ at a random point x.

As explained previously, it is the Verifier that asks for the evaluation of p on certain values. The Prover then provides these values with their Merkle proofs, allowing the Verifier to validate the points or not.

3.2 MerkleTree

For the MerkleTree implementation, I chose a typical class node with left and right children, a value (hashed) and a label. The class constructor takes as input the data as a list. The initial Nodes are created for each element l of the list, without left or right children, with a label being $str(l)$. The role of the label is just to simplify the searching algorithms in the Tree, also allowing an easier debugging. The value of each node is the $sha256$ encoding of l, and we use the function from the hashlib library. If we don't have a list of length a power of 2, we just append repeatedly the last element until we reach it. The tree is then constructed recursively. For each two consecutive nodes, we create their parent by:

1. Hashing the sum of their hashed values and giving the parent node this new hashed value.
2. Appending their string labels and making it the new parent's label.
3. Making these two nodes the left and right children of our new node.

We do this until there are only two parent nodes, from which we form the root of our tree, using the same method.

For example, the list $[42, 33, 62, 20]$ would induce the Merkle Tree represented in Figure 1.

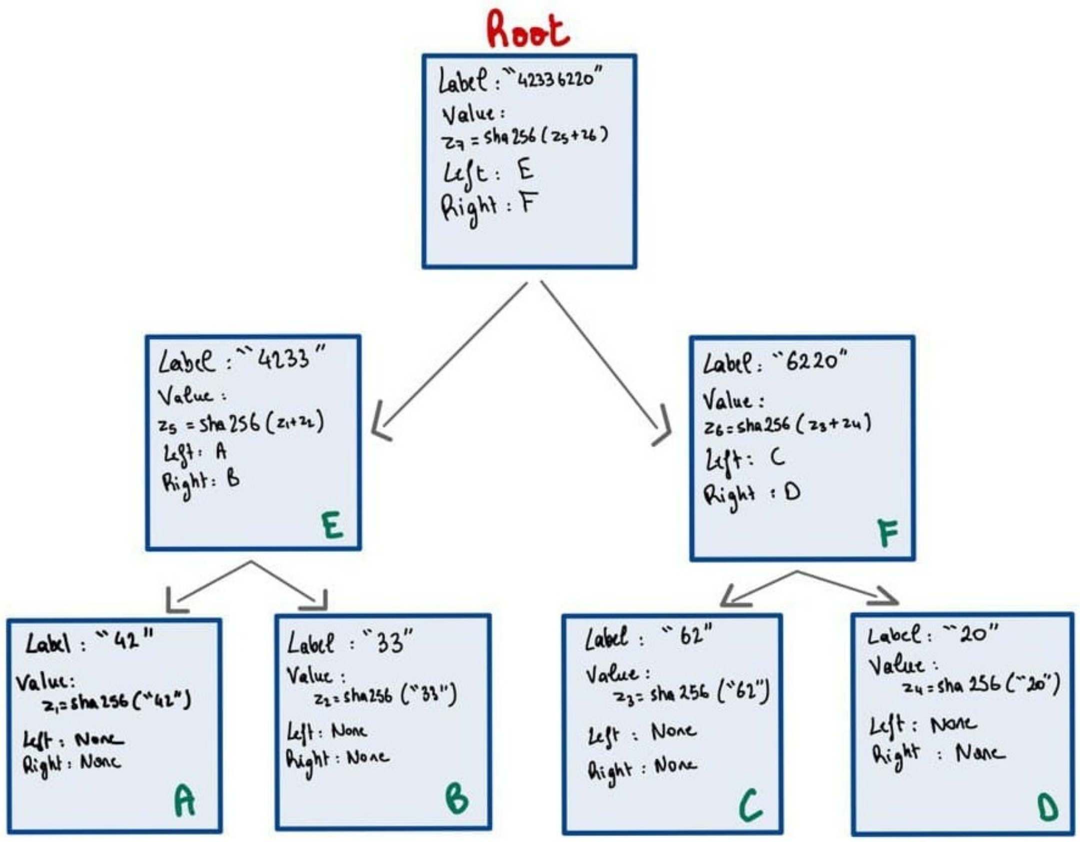

Figure 1: Illustration of a Merkle Tree

This allows to create a tree Mk from the evaluation of our composition polynomial on the points forming the evaluation domain. The root is then commited by the Prover. When the Verifier then asks for the evaluation of p on a point i.e. $p(x)$, the Prover provides it with a proof constructed as follow:

1. We search in Mk the path from the root to $p(x)$. For this, we use the function $binaryTreePaths$ that allows to get all paths from root to leafs in a Merkle Tree.

2. We provide a list $path$ that is formed by the Nodes of this path.

3. We iteratively traverse the Tree and provide $proof$, formed by the respective Node neighbors of $path$, i.e. for each Node n_0 in $path$, the neighbor n_1 of n_0 such that the parent of n_0 is formed by the hash of the sum of n_0 and n_1's values.

It is then easy for the Verifier to verify the authenticity of $p(x)$. The Verifier does so by going through the $path$ and $proof$ lists. Starting at the leaf of $path$, it hashes the sum of its value and the value of its neighbor in $proof$. It then checks that the value corresponds to the value of the next element in path, and does so iteratively until reaching the root. The last step consists in checking that the root computed by the Verifier corresponds to the one previously commited by the Prover, that ensures the authenticity of $p(x)$.

References

[1] 2020. URL: https://starkware.co/.

[2] Eli Ben-sasson et al. "Scalable Zero Knowledge with No Trusted Setup". In: Aug. 2019, pp. 701–732. ISBN: 978-3-030-26953-1. DOI: 10.1007/978-3-030-26954-8_23.

[3] *STARK 101.* Aug. 2020. URL: https://starkware.co/developers-community/stark101-onlinecourse/.

[4] StarkWare. *Arithmetization I.* Feb. 2019. URL: https://medium.com/starkware/arithmetization-i-15c046390862.

[5] StarkWare. *Arithmetization II.* Sept. 2020. URL: https://medium.com/starkware/arithmetization-ii-403c3b3f4355.

[6] StarkWare. *Low Degree Testing.* June 2019. URL: https://medium.com/starkware/low-degree-testing-f7614f5172db.

[7] StarkWare. *STARK Math: The Journey Begins.* June 2020. URL: https://medium.com/starkware/stark-math-the-journey-begins-51bd2b063c71.