polygon zkEVM

Wiki Document

Prover - eSTARK

An Introduction to the Polygon zkEVM Proving Mechanism using `pil-stark` v.1.0

November 13, 2023

The setting for STARK proofs are machine-like computations from which we can derive a certain arithmetization, giving us a set of constraints describing its correct execution. The polynomial interpolating the constrained values that arise from a certain arithmetization can, in fact, depend on the inputs of the computation itself giving what we call **committed polynomials** and, by definition, should be computed once per proof. However, a polynomial that is completely independent of the input values so it is kept constant among several executions of the same computation should be computed only once per arithmetization. The former kind of polynomials are called **constant polynomials** and, together with the constraints, represent the computation that is being executed, so that they are publicly available for both the prover and the verifier, unlike the committed ones, which are not directly available to the verifier.

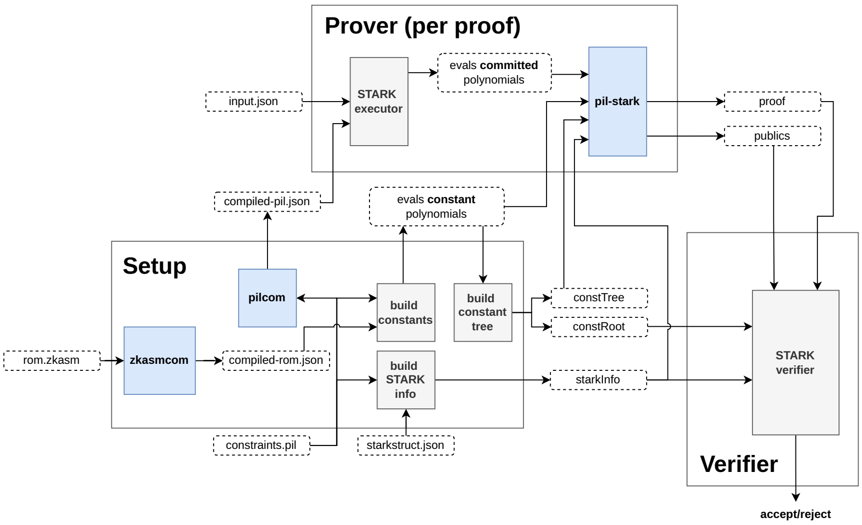

Figure 1: Proving Architecture Scheme in zkEVM.

Due to the fact that all the constant polynomials do not depend on the particular inputs of a certain computation, we can split the processes that make up the generation of a proof in two: the *setup phase* and the *proving phase*. This allows to execute both processes separately, which is highly important in order not to execute the constants' computation per proof, drastically increasing the proving time.

The Setup Phase The *setup phase* performs all the pre-processing. The setup is only done once per state machine definition, allowing to reuse it whether the definition does not change. The definition of a machine-like computation with `pil-stark` has three parts:

- **The program:** A `rom.json` file to build the computation. This file serves as the "guide" for constructing the computation, detailing the specific steps and operations that define the logic of the state machine. In the zkEVM, this file is obtained by compiling (using the `zkasmcom` compiler) a `.zkasm` file wrote in zkASM language, an assembly language specific for the zkEVM.

- **The constraints:** The PIL description of the STARK state machine, that is to say, the constraints that define the correctness of the execution trace. In the zkEVM, the constraints are wrote into a `.pil` file using PIL, standing for Polynomial Identity Language. The `pilcom` parser is written with https://github.com/zaach/jison, which generates bottom-up parsers in JavaScript. This compiled constraints file will be used by the prover to compute all the polynomials involved in the proving procedure.

- **The configuration:** The configuration of the FRI used in the STARK proof, inside a `starkStruct` file, specifying, among other, the blowup factor, the size of the trace and its LDE (Low Degree Extension), the number of queries to be performed and the willing-to-use step configuration.

Let's delve into each block service within the setup zone of the Figure 1.

- **Build Constants**: This service is in charge of generating a binary for the constant polynomials' evaluations from a given compiled program `rom.json` and a set of constraints `file.pil`. The compiled program is directly used to accurately populate the polynomial values based on the its rules, while the PIL file is only used to extract the polynomial degree, also known as the length of the execution trace. This service is run using the script `main_buildconstants` from the `zkevm-proverjs` repository.
- **Build Stark Info**: This block, managed by the `main_genstarkinfo` script, plays a pivotal role in the setup phase. This service involves the generation of a `starkInfo` file, containing essential information required for the STARK proof generation process. Beyond the conventional `starkStruct` details, this file incorporates additional information intrinsic to the eSTARK protocol. Notably, it integrates expressions computation helpers tailored for the polynomials constructed in each of the eSTARK phases, which will be explained later on.
- **Build Constant Tree**: Executed through the `main_buildconsttree` script, is tasked with the creation of a structured representation of constant polynomials. This tree-like structure facilitates efficient manipulation and retrieval of constant polynomial values during the proof generation process. It also generates the Merkle root of its corresponding tree for the verifier, which we call constant root (`constRoot`).

The Proving Phase In the *proving phase*, the Prover executes all the processes that, given an input, generate a proof for the computation. The STARK executor process computes the evaluations of the polynomials that are going to be committed. To do so, it takes the names and descriptions of the polynomials from the parsed PIL and the provided inputs. Observe that, since the values of the committed polynomials are strongly dependent on the inputs, this procedure should be executed once per proof, unlike the setup phase.

Finally, the `pil-stark` eSTARK prover process takes the evaluations of both the constant and the committed polynomials of the previous steps and all the information stored in the `starkInfo` object in order to generate the corresponding eSTARK proof and the associated public values for which the proof is valid. We use the eSTARK protocol, which is specially designed to proof PIL statements. A complete description of the `eSTARK` protocol can be found in [MAGABMMT23].

As a summary, the eSTARK protocol is composed on two main stages:

- *Low-Degree Reduction phase*: Following [Sta21], first of all, we obtain a polynomial called FRI which codifies the validity of the values of the trace according to the PIL into the fact that it has low degree. This polynomial is committed to the verifier, as well as several previous polynomials that are used to provide consistency checks between them. This phase, however, differs from the one described in [Sta21] because PIL also accepts, apart from polynomial equalities, arguments such lookups, permutations or even copy-constraints (called connection arguments). Hence, this phase needs to be adapted in order to proof the correctness of each of the enumerated arguments. This serves as a motivation to call this protocol eSTARK, standing for *extended STARK*.

- *FRI phase*: After obtaining the so called FRI polynomial, the prover and the verifier are involved into a FRI Protocol, aiming for proving that the committed polynomial has low degree (more concretely, it proves that the committed values of the polynomials raise a function that is close enough to a polynomial of low degree.

The first stage can, in turn, be divided into several rounds. Below, we describe in a high level what is each of the rounds aiming.

- *Round 1:* Given the trace column polynomials interpolating the execution trace, the prover commits to them.
- *Round 2:* The prover commits, for each lookup argument, to the h-polynomials of the modified plookup version described in [PFM+22] (see [GW20] for more information about plookup protocol).
- *Round 3:* The prover commits to the grand-product polynomials for each of the arguments appearing in the PIL together with some intermediate polynomials used to reduce the degree of the grand products. This is due to the fact that `pil-stark` imposes a degree bound when committing to a polynomial. See [GWC19, GW20] for the specification of the grand-products of each of the different arguments allowed in PIL.
- *Round 4:* The prover commits to the 2 polynomials Q_1, Q_2 arising from the splitting of the quotient polynomial Q described in the `eSTARK` protocol.
- *Round 5:* The prover provides the verifier with all the necessary evaluations of the polynomials so that he/she can execute the corresponding checks.
- *Round 6:* The prover receives two randomness from the verifier which are used to construct the previously described FRI polynomial. Then, the prover and the verifier are involved into a FRI Protocol, ending with the prover sending the corresponding FRI proof to the verifier.

The `pil-stark` package provides an implementation of a **Prover** instance at the `main_prover.js` script.

Verification Phase After the generation of the proof, it is sent to the designated **Verifier** instance. The **Verifier** is in charge of executing a well-defined procedures stating the correctness of the proof submitted by the **Prover**. This correctness, in turn, signifies the cryptographic correctness of the initial computational statement. The `pil-stark` package provides an implementation of a **Verifier** instance at the `main_verifier.js` script.

References

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[MAGABMMT23] Héctor Masip-Ardevol, Marc Guzmán-Albiol, Jordi Baylina-Melé, and Jose Luis Muñoz-Tapia. estark: Extending starks with arguments. Cryptology ePrint Archive, Paper 2023/474, 2023. https://eprint.iacr.org/2023/474.

[PFM$^+$22] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. Cryptology ePrint Archive, Paper 2022/086, 2022. https://eprint.iacr.org/2022/086.

[Sta21] StarkWare. ethstark documentation. Cryptology ePrint Archive, Paper 2021/582, 2021. https://eprint.iacr.org/2021/582.