polygon zkEVM

Technical Document

Recursion, aggregation and composition of proofs v.1.0

February 10, 2023

Contents

1 Introduction

This document specifies how the polygon zkEVM is proven using recursion, agregation and composition. The constraints of the zkEVM are specified as polynomial identities using the PIL language. Then, an execution trace can be proven using the PIL specification for building a STARK that is proved with the FRI protocol. The problem is that STARKs generate big proofs. This document describes how to use recursion together with composition to shorten the prove size.

In a high level, a basic recursion block transforms a PIL specification into the specification of its STARK verification circuit (written in Circom). The circuit verifies the STARK of the PIL specification. Then, the Circom specification is transformed into a Plonkish PIL specification and the process is iterated.

In addition to recursion, also aggregation is implemented so that provers can aggregate the proofs of multiple transaction batches.

2 Tools

2.1 PIL

Polynomial Identity Language (PIL) is a novel domain-specific language to define the constraints of computation traces of either computations based on the circuit model or computations based on the state machine model. The constraints of the zkEVM execution trace, which is based on a state machine, are specified with PIL.

2.2 Circom

In Figure 1 we show the architecture of Circom. Programmers can use the Circom language to define arithmetic circuits and the compiler generates a file with the set of associated R1CS constraints together with a program (written either in cpp or wasm) that can be run to efficiently compute a valid assignment to all wires of the circuit.

After compiling a circuit, we can calculate all the signals that match the set of constraints of the circuit using the cpp or wasm programs generated by the compiler. To do so, we simply need to provide a file with a set of valid input values, and the program will calculate a set of values for the rest of signals of the circuit. A valid set of input, intermediate and output values is called *witness*.

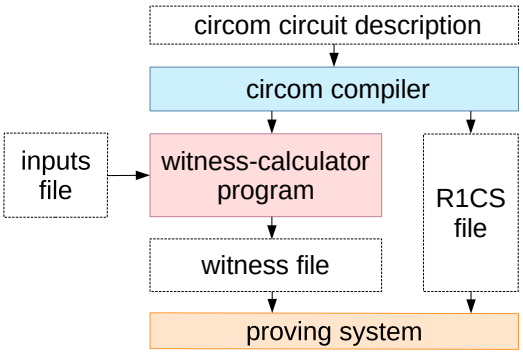

Figure 1: Circom

2.3 Non-recursive STARK

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 building the constraints that arise from a certain arithmetization can, in fact, depend on the inputs of the state machine 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 state machine should be computed only once per arithmetization. The former kind of polynomials are called *constant polynomials* and 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.

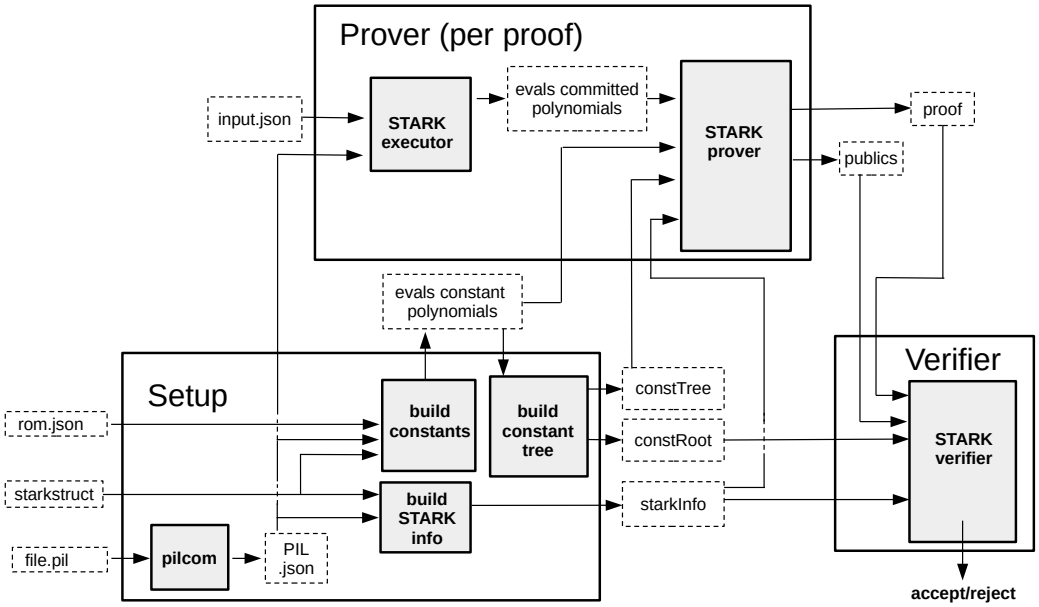

Figure 2: Non recursive STARK.

Due to the fact that all the constant polynomials do not depend on the particular inputs of a certain computation executed by the state machine, 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* 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 state machine with `pil-stark` has three parts:

- The `rom.json` file to build the computation.
- The configuration of the FRI used in the STARK proof, inside a `starkstruct` file, specifying the blowup factor, the size of the trace and its LDE (Low Degree Extension) and the number of queries to be performed.
- The `PIL` description of the STARK state machine, that is to say, the constraints that define the correctness of the execution trace (`file.pil` file).

The PIL file and the `starkstruct` are used by the `build STARK info` process to write a file called `starkInfo` containing, apart from all the FRI-related parameters, several useful fields related with the PIL and the shape of the constraints. The PIL description

is parsed with a compiler called `pilcom`[1] to obtain a parsed JSON version of the PIL, which will be used by the prover to compute all the polynomials involved in the proving procedure. The `build constants` process, using the parsed PIL, computes the evaluations of the constant polynomials over the evaluation domain determined by the `starkstruct`. Additionally, once this evaluations are computed, the `build constant tree` process generates the Merkle root of its corresponding tree for the verifier, which we call constant root (`constRoot`).

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` STARK 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 STARK 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. The eSTARK protocol is composed on two main stages:

- *Low-Degree Reduction phase*: Following [**?**], 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 [**?**] 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, see [**?**] for more information on FRI Protocol).

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 [**?**] (see [**?**] 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 [**?**, **?**] 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.

[1]The `pilcom` parser is written with https://github.com/zaach/jison, which generates bottom-up parsers in JavaScript.

- *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.

After the proof is generated, it is sent to the Verifier instance so that he/she can start the verification procedure, after what will accept or reject the proof.

3 Composition, Recursion and Aggregation

3.1 Composition

As shown in Section 2.3, the basic verification of a STARK is performed by a verifier entity using the proof, the publics and some other verifier parameters. Composing proofs means using different proving systems together to generate a proof. Generally composition is used to increase the efficiency of some part of the system. In our case, as first proving system we use a STARK and our main idea of composition is to delegate the verification procedure of the STARK proof π_{STARK} to a verification circuit C. In this case, if the prover provides a proof for the correct execution of the verification circuit $\pi_{CIRCUIT}$, then this is enough to verify the original STARK. As shown in Figure 3, in this case, the verifier entity just verifies the proof of the STARK verification circuit $\pi_{CIRCUIT}$. The advantage of this composition is that $\pi_{CIRCUIT}$ is smaller and faster to verify than π_{STARK}.

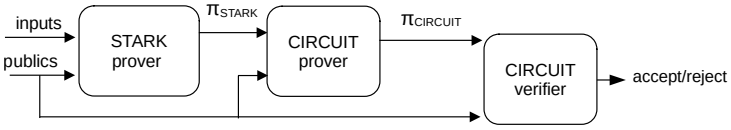

Figure 3: Simple composition.

3.2 Recursion

3.2.1 Setup Phase

Since verifiers are much more efficient than provers, we can use this fact to create a recursive cascade of verifiers in which at each step we achieve a proof that can be more efficiently verified. In our architecture, we create a chain of STARK verifiers using intermediate circuits for the definition of these STARK verifiers as shown in Figure 4. We use circuits because they are suitable for computations with limited branching and a verifier is a computation of this type.

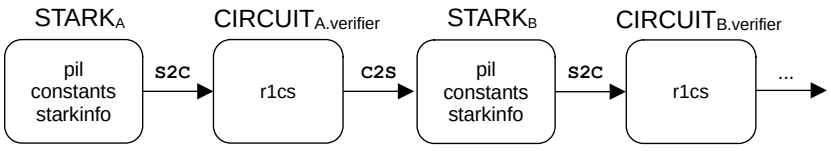

Figure 4: Recursion setup.

Following Figure 4, lets consider that we have the parameters that describe the first STARK (pil, constants and starkinfo). This first STARK that we denote as $\texttt{STARK}_\texttt{A}$, is then

automatically translated into its verifier circuit. The STARK verifier circuit is described with R1CS constraints. This translation, that we call `S2C` (STARK-to-CIRCUIT), is performed in a setup phase. In other words, the R1CS description of the STARK verifier circuit can be pre-processed before the computation of the proof. We use Circom as intermediate representation language for the description of the circuits (see Section 3.4 for more details about `S2C`).

Next, we take the circuit definition (R1CS) and automatically translate it into a new STARK definition, that is to say, a new pil, new constants and a starkinfo. This translation, that we call `C2S` (CIRCUIT-to-STARK), is performed also in a setup phase. Following our example, the new generated STARK is denoted as $\texttt{STARK}_\texttt{B}$ and it is essentially a $\mathcal{P}\texttt{lon}\mathcal{K}$ish arithmetization with some custom gates of the verification circuit of $\texttt{STARK}_\texttt{A}$ (for more details about `S2C` see Section 3.5). It is worth to mention that these recursion steps can be applied as many times as desired taking into account that each step will compress the proof making it more efficient to verify but increasing the prover complexity. Finally, remark that during the setup phase, several artifacts for generating each STARK prover are generated (see Sections 3.4 and 3.5 for more information about these artifacts).

3.2.2 Proving Phase

The first proof is generated by providing the proper inputs and public values to the first STARK prover. Then, the output proof is passed as input to the next STARK prover together with the public inputs and the process is recursively repeated. In Figure 5 we show how in essence a chain of recursive STARK provers work.

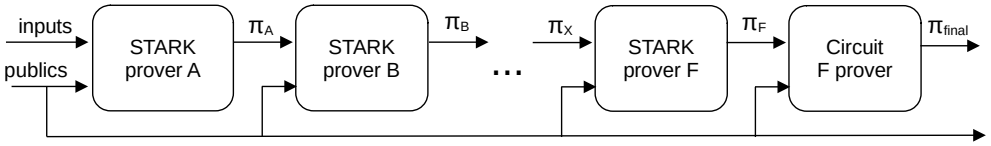

Figure 5: Recursive provers.

Notice that the final proof is actually a circuit-based proof (currently we are using a Groth16 proof). More details about the proving phase can be found in Section 3.6.

3.3 Aggregation

Our architecture also allows aggregation when generating the proofs. Aggregation is a particular type of proof composition in which multiple valid proofs can be all proven to be valid by comprising them all into one proof, called the *aggregated proof*, and only validating the aggregated one. In our architecture, aggregators are defined in intermediate circuits. Figure 6 shows an example of aggregation with binary aggregators.

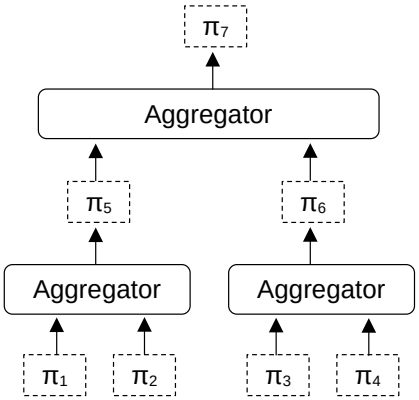

Figure 6: Aggregation example.

3.4 Setup S2C

Recall that we denote S2C to the process of converting a given STARK into its verifier circuit, which is described in Circom and compiled to the corresponding R1CS constraints. The architecture of this generic conversion is depicted in Figure 7, where a $\texttt{STARK}_x$ is converted into a circuit denoted as C_y.

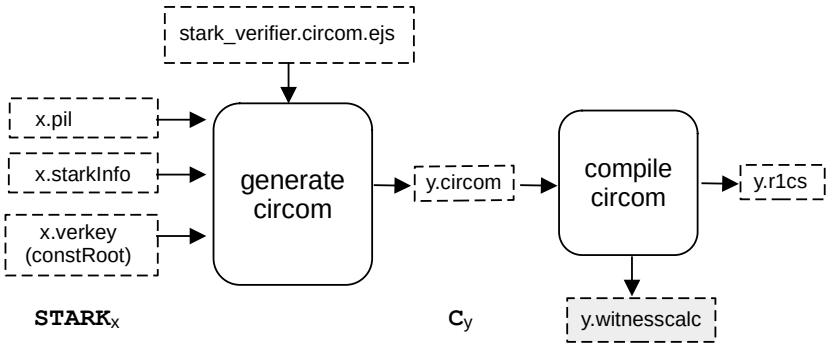

Figure 7: Setup S2C.

The input of the S2C step is all the information needed to set up a circuit verifying the given STARK. In our architecture this is a PIL file specifying the STARK constraints that are going to be validated and the polynomial names, a `starkInfo` file containing the FRI-related parameters (blowup factor, the number of queries to be done, etc.), and the Merkle tree root of the computation constants (`constRoot`). The output of the generate Circom process is a Circom description. The circuit is actually generated by filling an EJS template for the Circom description using the constraints defined by the PIL, the FRI-related parameters included by the `starkInfo` file and the `constRoot`. As shown in Figure 8, the inputs of the generated STARK verifier circuits are divided in two groups: the public inputs and the private inputs.

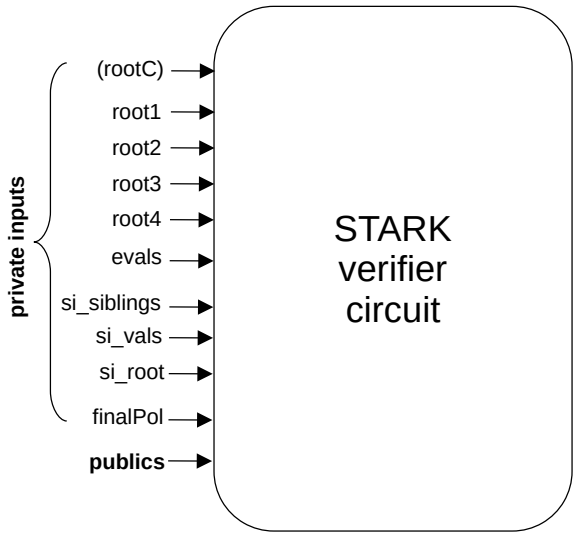

Figure 8: Inputs of the STARK verifier circuits.

The private inputs are the parameters of the previous STARK proof:

- `(rootC)`: Four field elements being the root of the Merkle tree for the evaluations of constant (that is, preprocessed) polynomials of the previous STARK. In some of the intermediate circuits that we generate, `rootC` is an input of the circuit while in other generated circuits `rootC` are internal signals hardcoded to the corresponding values (more information for each particular case is provided later).
- `root1`: Four field elements being the root of the Merkle tree for the evaluations of all the trace column polynomials for the execution trace.
- `root2`: Four field elements being the root of the Merkle tree for the evaluations of the h polynomials appearing in each lookup argument of the previous STARK. This root may be 0 if no lookup argument is provided in the PIL.
- `root3`: Four field elements being the root of the Merkle tree for the evaluations of the grand product polynomials appearing in each argument (that is, lookup, permutation or connection arguments) of the previous STARK and the intermediate polynomials appearing in certain splitting of them. This root may be 0 if no arguments are provided in the PIL.
- `root4`: Four field elements being the root of the Merkle tree for the evaluations of the splitting Q_1 and Q_2 of the Q polynomial of the previous STARK.
- `evals`: Contains all the necessary evaluations for all the polynomials appearing in the FRI verification process at a challenge value z and at gz.
- `si_root`: Four field elements being their root of the Merkle tree for the evaluations of the i-folded FRI polynomial, that is, the polynomial appearing in the i-th step of the FRI verification.
- `si_vals`: The leaves' values of the previous Merkle tree used to check all the queries. The total amount of such values depends on the number of queries and the reduction factor attached to the current step of the FRI.
- `si_siblings`: Merkle proofs for each of the previous evaluations.

- `finalPol`: Contains all the evaluations of the last step's folding polynomial constructed in the FRI verification procedure over the last defined domain which has the same size as the degree of the polynomial.

The `publics` are a set of inputs that will be used by the verifier to check the final prove and also by the intermediate STARKs (more information about publics used in the zkEVM STARK is provided in Section 4.1).

The final process to complete the `S2C` step is to compile the Circom description to obtain a file with the associated R1CS constraints and a witness calculator program capable of compute all the values of the circuit wires for a given set of inputs.

Finally, remark that the particular intermediate circuit generated in a `S2C` step, denoted as C_y in Figure 7, can be just a verifier of the previous STARK ($\texttt{STARK}_\texttt{x}$ in Figure 7) if we are only applying a recursion step, but more generally, other types of circuits that include the verifier but provide more functionality can be used. This latter is the case when we use circuits to verify aggregation of proofs.

3.5 Setup `C2S`

In our proving architecture, we create a chain of STARKs. From the `S2C` step we can obtain a circuit that has to be converted again into a STARK. The step that achieves this conversion is a pre-processing step that we call `C2S`. A picture of a generic `C2S` step can be found in Figure 9, where a circuit denoted as C_y is converted into its corresponding STARK ($\texttt{STARK}_\texttt{y}$).

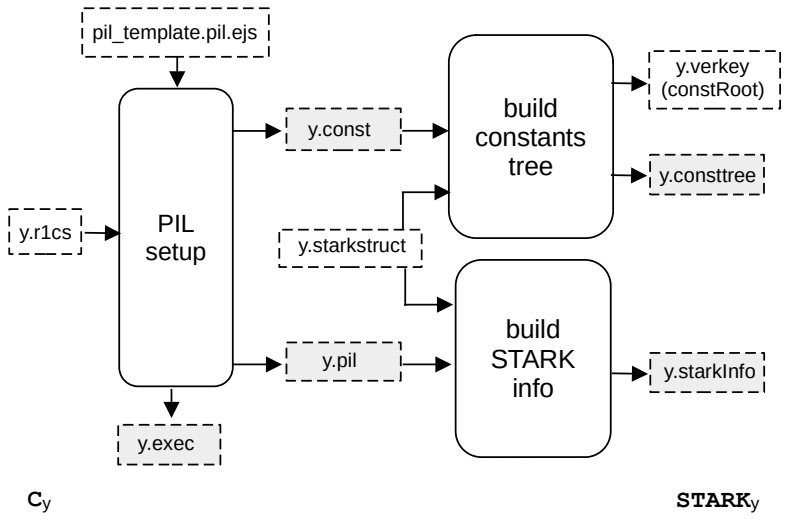

Figure 9: Setup recursion step `C2S`.

In more detail, the STARK arithmetization of the intermediate circuits of our proving architecture is a $\mathcal{P}\texttt{lonK}$ish arithmetization with custom gates and using 12 polynomials for the values of the gate wires of the computation trace. The STARK arithmetization includes several custom gates for doing specific tasks more efficiently. In particular, the custom gates provide the following functionality:

- **Poseidon**: This custom gate is capable of validating a Poseidon hash from a given 8 field elements as inputs, 4 field elements as capacity and a variable number of output elements. More specifically, this circuit implements the MDS matrix and the 7-th power of field elements computations used to execute each of the rounds defined by the Poseidon hash sponge construction.

- **Extended Field Operations**: This custom gate is capable of validating multiplications and additions (or a combination of both) over the extended field $\mathbb{F}_{p^3}$. The inputs are 3 elements $a, b, c \in \mathbb{F}_{p^3}$ and the output is corresponds to the element

$$a \cdot b + c \in \mathbb{F}_{p^3}$$

  where the operations are defined over $\mathbb{F}_{p^3}$. Observe that defining c equal to 0 one can compute pure multiplications. Similarly, setting b equal to 1, pure additions can be computed.

- **FFT**: This custom gate is in charge of computing FFT of a variable size in $\mathbb{F}_p$ or in an extension field.

- **Polynomial Evaluation**: This custom gate is in charge of computing a single evaluation of polynomials in $\mathbb{F}_{p^3}$ using Horner's rule. The input consists on a field element $z \in \mathbb{F}_{p^3}$ and the coefficients of the polynomial p which we are going to evaluate. The output is the evaluation $p(z) \in \mathbb{F}_{p^3}$.

We use selector polynomials to activate the custom gates of our STARK. These polynomials are constant (pre-processed). In particular, we use the selectors `POSEIDON12`, `GATE`, `CMULADD`, `EVPOL4` and `FFT4` are introduced. The selector `GATE` is actually in charge of activating a basic $\mathcal{P}\mathtt{lonK}$ gate. The other selectors are in charge of selecting whichever subcircuit needs to be executed. Moreover, there also exist a special selector called `PARTIAL` which is in charge of distinguishing between partial and full layers in the Poseidon, since this affects the relationships between the values.

At this point we can detail the processes of the `C2S` step. As shown in Figure 9, the first process of the `C2S` step is the PIL setup, which takes the R1CS constraints of a given intermediate circuit as input and produces all the STARK-related artifacts. This includes the associated STARK identity constraints and the computation constants that are respectively stored in a PIL file (`y.pil`) and in a file of constants (`y.const`). In particular, the identity constraints of the $\mathcal{P}\mathtt{lonK}$ish arithmetization are generated by filling an `EJS` template for the associated PIL (for the zkEVM the template used is called `compressor12.pil.ejs`).

The PIL setup also generates an important file with exec extension (`y.exec`) that defines how to rearrange the values produced by the circuit witness calculator into the appropriate values of the STARK execution trace (see Thaler's book for more information about this rearrangement process). Notice that the rearrangement rules and the computation constants only depend on the circuit shape (which is coded in the `.r1cs` file generated by the Circom compiler). In other words, these parameters do not depend on the particular values of the circuit wires computed for a particular input. Nevertheless, we will use the rearrangement rules file together with the witness values for a given input later on to build the STARK execution trace, which in turn is needed to generate the STARK proof.

Finally, we also produce the `starkInfo` file and a Merkle tree with the STARK constants.

3.6 Recursion Step Proof

As we explained in Section 3.2.2, to generate the final proof, we have to compute the proof of each intermediate STARK (see Figure 5). Each intermediate STARK proof is generated using the witness values provided by the execution of the associated circuit witness computation program using as inputs the publics and the values of the previous proof. Then, these values are properly rearranged to build the STARK execution trace using the corresponding `.exec` file. In Figure 10, we provide the scheme of how the proof of an intermediate STARK is generated.

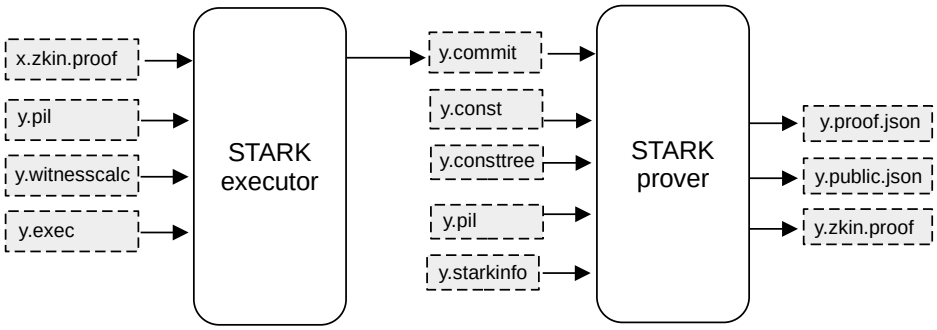

Figure 10: STARK `Proof` of a recursion step.

As it can be observed, the STARK executor process takes the parameters of previous proof and the public inputs (both in the file `x.zkin.proof` which has the proper format for the witness calculator generated by the Circom compiler), the PIL of the current STARK (`y.pil`), the witness calculator program of the associated circuit (`y.witnesscalc`) and the file of rearrangement rules (`y.exec`) to generate the non-preprocessed part of the STARK execution trace (`y.commit`). Next, the STARK prover process takes the execution trace, that is to say, the committed and constant polynomials, the constant tree, the corresponding PIL file and the information provided by the `zkevm.starkinfo.json` file to generate the proof. Finally, when the proof is generated, the STARK prover process generates three files:

- **Proof File** (`y.proof.json`): A json file containing the whole STARK proof in a `.json` file.
- **Publics File** (`y.public.json`): A json file containing only the publics.
- **zkIn File** (`y.zkin.proof.json`): A json file combining both the proof and the publics.

4 Polygon zkEVM

4.1 Architecture

In this section, we provide the concrete blocks and steps used to prove the correct execution of a batch of transactions (or several batches) by our zkEVM using recursion, aggregation and composition. As previously mentioned, generating a proof has two phases. The first phase is a setup executed only once per STARK computation definition. In our case, the STARK computation is the processing of batches by our zkEVM. In the setup phase, the different artifacts needed to generate proofs are preprocessed. The second phase is when actually proofs are generated for given inputs (i.e. batches of transactions). An overview of the overall process can be observed in figure 11.

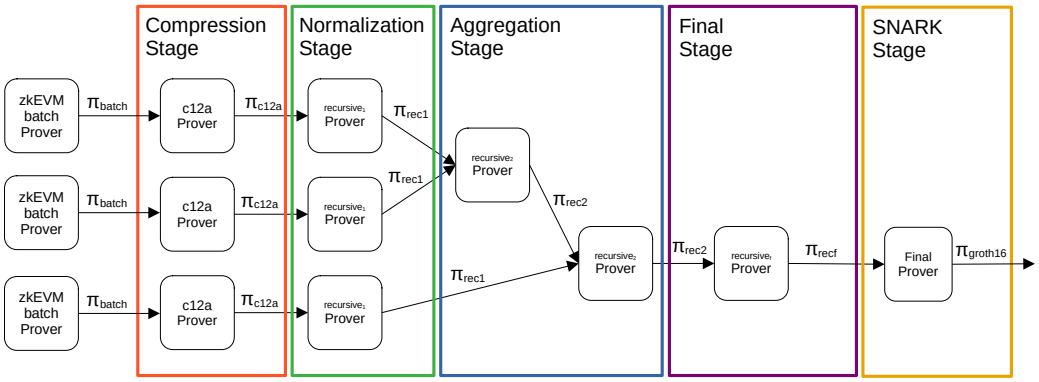

Figure 11: Proving architecture with recursion, aggregation and composition.

Recall that the first STARK generates such a big proof since it has a lot of polynomials so its attached FRI uses a low blowup factor. Henceforth, a first *Compression Stage* its invoked in each batch's proof, aiming to reduce the number of polynomials used, allowing to augment the blowup factor and therefore, reduce the proof size.

Once the compression step has been completed, a proof aggregation stage will be in charge of joining several batches proofs into a single proof proving each of the single proofs all at once. The way of proceeding will be to construct a binary tree of proofs by aggregating two by two each of them. We will call this the *Aggregation Stage*.

However, since the aggregation of two proofs requires the constant root of the previous circuits through a public input coming from the previous circuit, there exists a *Normalization Stage* which is in charge of transforming the obtained verifier circuit verifying the `c12a` proof into a one making the constant root public to the next circuit. This step allows each aggregator verifiers and the normalization verifiers to be exactly the same, permitting successful aggregation via a recursion.

Once the normalization step has been finished, its time for aggregation. In this step we are going to join two batches' proofs together, which will be done many times until only one proof spares. In order to do so, a circuit capable of aggregating two verifiers is created. However, as we can see in the figure below, the inputs of this stage can be proofs of the kind π_{re1} coming from the previous normalization stage, or already aggregated proofs π_{re2}. This allows us to aggregate two π_{re1} proofs, two π_{re2} proofs or a combination of a π_{re1} and a π_{re2} proofs. Henceforth, this aggregation stage has to take into account this fact in its design.

Observe that the *Aggregation Stage* needs to be designed in order to accept either already aggregated proofs or only compressed ones. The *Final Stage* is the very last STARK step among the recursion process, which is in charge of verifying a π_{re2} proof over a completely different finite field, the one defined by the `bn128` elliptic curve. More specifically, the hash in charge of generating the transcript will work over the field of the `bn128` curve. Hence, all the challenges (and so, all polynomials) will belong to this new field. This is done in this way because, in the next step of the process, a SNARK Groth16 proof, which works over elliptic curves, will be generated. In this step is very similar to the others, instantiating a verifier circuit for π_{re2} but, in this case, 2 constants roots should be provided (one for each of the proofs aggregated in the former step).

The last step of the whole process is called *SNARK Stage*, and it purpose is to produce a Groth16 proof π_{groth16} verifying the previous π_{ref} proof. In fact, Groth16 can be replaced with any other SNARK proof. A SNARK is chosen here in order to reduce both verification complexity and proof size, which have constant complexity unlike STARK proofs. The π_{groth16} proof will be sent to the verifier so that he/she can verify it.

As a final remark, one should observe that the whole set of public inputs are being

passed as inputs in each proof of the whole recursion procedure. The set of all public inputs is listed below (see the technical documents about the zkEVM L2 state management and the bridge):

- `oldStateRoot`
- `oldAccInputHash`
- `oldBatchNum`
- `chainId`
- `midStateRoot`
- `midAccInputHash`
- `midBatchNum`
- `newStateRoot`
- `newAccInputHash`
- `localExitRoot`
- `newBatchNum`

Next, let's describe the details of the steps and processes performed in each phase.

4.2 Setup Phase

The setup phase is a pre-processing phase in which all the artifacts for generating proofs are created. This includes the generation of intermediate circuits, which are a finite set of circuits that allow arbitrary combinations of proof recursions and proof aggregations.

4.2.1 Build the zkEVM STARK

As shown in Figure 12, we start by building the ROM of the zkEVM state machine, which is the program containing the instructions for the the executor that will generate the execution trace of the zkEVM. We also build the PIL that validates the execution trace. The ROM will, in fact, generate all the constant values for the execution trace of the zkEVM. Observe that, as said before, committed polynomials are not needed in the setup phase, so we need not run the executor of the zkEVM in order to generate them at this point.

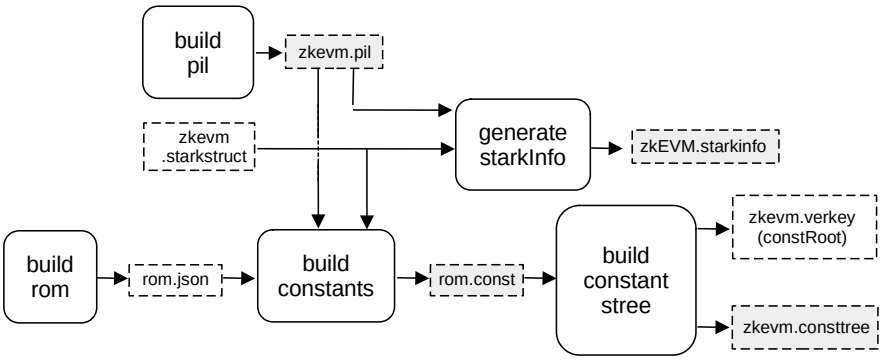

Figure 12: Build the zkEVM STARK

The Merkle root of the tree of constant polynomials' evaluations, which is a hash that serves as cryptographic summary to capture all the fixed parameters of the computation, is stored as a parameter in a file called `zkevm.verkey`. The last piece of data that is generated before building the STARK is the `starkInfo` that is necessary for automatically generating the circuit that will verify the zkEVM STARK. In this case, we use a blowup factor of 2 and 128 queries to generate the proof. The artifacts marked in gray will be used when generating the proof (proof generation is described in more detail in Section 4.3).

4.2.2 Setup S2C for the zkEVM STARK

The next step in the setup is to generate the circuit to verify the zkEVM STARK (see Figure 13).

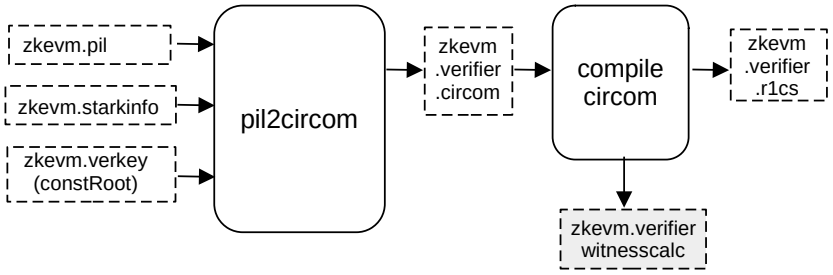

Figure 13: Converting the zkEVM STARK verification into a circuit.

The `pil2circom` process fills an `EJS` template called `stark_verifier.circom.ejs` with all the necessary information needed to validate the zkEVM STARK. Henceforth, we need to add the `zkevm.pil` in order to capture polynomial names, the `zkevm.starkinfo` file which specifies the blowup factor, the number of queries and the steps of the FRI-verification procedure and the `constRoot` in the `zkevm.verkey` file to automatically generates a circuit in Circom. The output Circom file `zkevm.verifier.circom` is then compiled into R1CS constraint system written in a file called `zkevm.verifier.r1cs`. This constraints will be used in the next step to generate the PIL and the constant polynomials for the next proof.

On the other hand, the Circom compilation also outputs a witness calculator program that we call `zkevm.verifier.witnesscalc`. As it can be observed in the picture, the witness calculator program is marked in gray because it is going to be used when generating the proof.

Since our aim at the next proof generation will be compression (that is, proof size reduction) we will use a blowup factor of 4 in this step, with 64 queries. This information is contained in the `c12a.starkstruct` file located in the `proverjs` repository.

4.2.3 Setup C2S c12a

The circuit that verifies the zkEVM STARK is called `zkevm.verifier` (or also `c12a`). This is because the PIL that is going to verify the `c12a` circuit is a $\mathcal{P}\mathfrak{lonK}$ish circuit with custom gates and 12 polynomials aiming at compression.

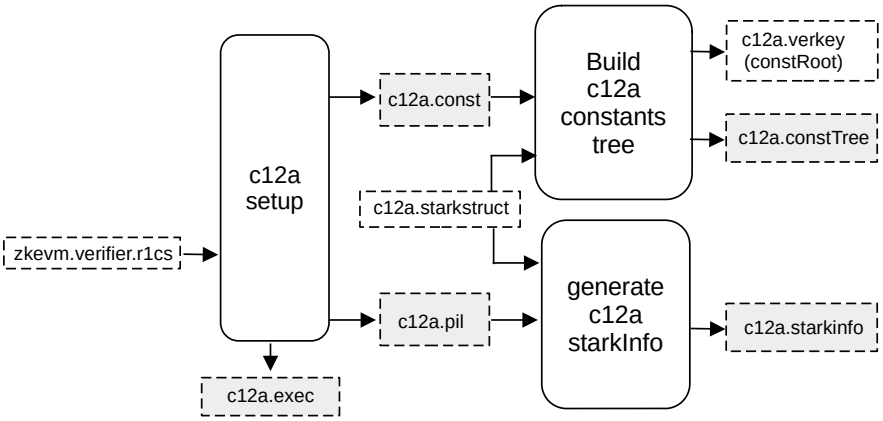

Figure 14: Convert the zkEVM verifier circuit to a STARK called `c12a`.

From the previous R1CS description of the verification circuit we are going to obtain a machine-like construction whose correct execution, which will be described by a PIL file, will be equivalent to the validity of the previous circuit. This process is started through a service called `compressor12_setup`, where the corresponding PIL file for verifying the trace is output, as well as a binary for all the constant polynomials `c12a.const` defined by it. Moreover, a helper file called `c12a.exec` is generated by the same service. This helper file will contain all the necessary rules that will allow us to shuffle all the witness values, which will be computed later on, into the corresponding position of the execution trace. The design of this shuffling, together with the the connections defined in the constants polynomials `c12a.const` will ensure that, for a honest prover, this newly generated trace is valid whenever the previous circuit is.

In order to finish the set up phase, having all the FRI-related parameters willing to be used in this step stored in a `c12a.starkstruct` file (located in the prover repository), we can generate the `c12a.starkinfo` file through the `generate_starkinfo` service and build the constants' tree (and its respectively constant root).

4.2.4 Setup `S2C` for `recursive1`

Up to this point we have a STARK proof π_{c12a} verifying the first big STARK proof π. The idea now is, as before, generate a Circom circuit that verifies π_{c12a} by miming the FRI verification procedure. To do that, we generate a verifier circuit `c12a.verifier.circom` from the previously obtained `c12a.pil` file, the `c12a.starkinfo` file and the constant root `c12a.verkey.constRoot` by filling the `stark_verifier.circom.ejs` template as before.

In this case, in seek of normalization, we need to briefly modify this circuit in order to include the constant root as a public input. Observe that, since we are not still aggregating, the constant root will not be actually used here, though this will be extremely important in the aggregation stage, where all the constants for the computation, which depend on the previous circuit, need to be provided as public inputs. This is done by using `recursive1.circom` file and importing inside the previously generated `c12a.verifier.circom` circuit as a library. The verifier circuit is instantiated inside `recursive1.circom`, connecting all the necessary wires and including the constant root to the set of publics.

The output circom file `recursive1.circom` it is compiled into a R1CS `recursive1.r1cs` file and a witness calculator program `recursive1.witnesscal` which will be both used later on in order to build and fill the next execution trace.

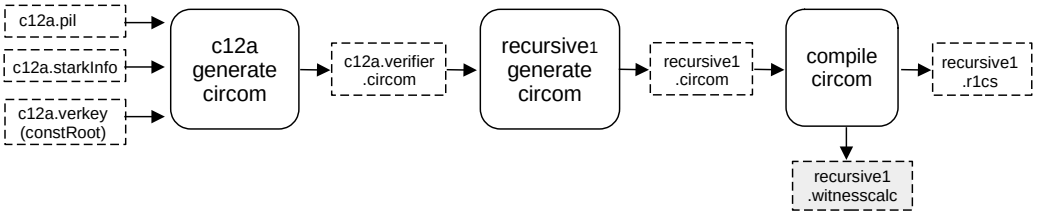

Figure 15: Convert the c12a STARK to a c12a verifier circuit.

4.2.5 Setup `C2S` for `recursive1`

As before, from the previous R1CS description of the verification circuit we are going to obtain a machine-like construction whose correct execution, which will be described by a PIL `recursive1.pil` file, will be equivalent to the validity of the previous circuit. Also, a binary for all the constant polynomials `recursive1.const` defined by it and the helper file providing the witness values allocation into its corresponding position of the execution trace `recursive1.exec` are generated.

In order to finish the set up phase, having all the FRI-related parameters willing to be used in this step stored in a `recursive.starkstruct` file (located in the prover repository), we can generate the `recursive1.starkinfo` file through the `generate_starkinfo` service and build the constants' tree (and its respectively constant root). In this case, we are using a blowup factor of $2^4 = 16$, allowing the number of queries to be 32.

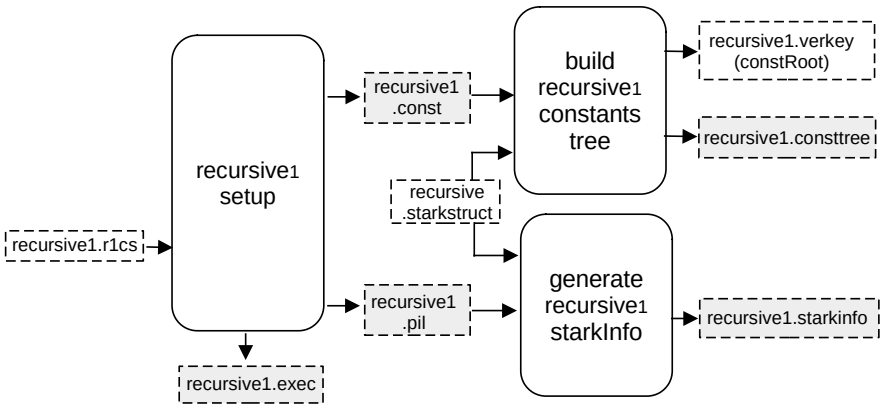

Figure 16: Convert the `recursive1` circuit to its associated STARK.

4.2.6 Setup `S2C` for `recursive2`

Up to this point we have a STARK proof π_{re1} verifying the proof π_{c12a}. As before, we generate a Circom circuit that verifies π_{re1} by miming the FRI verification procedure. To do that, we generate a verifier circuit `recursive1.verifier.circom` from the previously obtained `recursive1.pil` file, the `recursive1.starkinfo` file and the constant root `recursive1.verkey.constRoot` by filling the verifier `stark_verifier.circom.ejs` template.

After the verifier is generated using the template, we will also use a template to create another Circom that aggregates two verifiers. Note that, in the previous step, the constant root is passed harcoded into the circuit from an external file. But this was intended as a normalization, allowing the previous circuit and each ones verifying each of both proofs to have exactly the same form, allowing recursion being possible. Henceforth, this `recursive2.circom` circuit has two verifiers and a two multiplexors that are actually

deciding the form of each of the verifiers: if the proof has π_{re1} form, the hardcoded constant root is input but, if the proof has π_{re2} form, the constant root should be connected as input signal, coming from a previous circuit.

A schema of the `recursive2` circuit generated is as shown in Figure 17. Observe that, since the upper proof has the π_{re2} form, the Multiplexor does not provides the constant root `rootC` to the Verifier A to hardcode it because this verifier should get it trough a public input from the previous circuit. Otherwise, the lower proof has the π_{re1} form, so the Multiplexor let is pass trough the constant root into the Verifier B so that it can be hardcoded it when the corresponding template is filled.

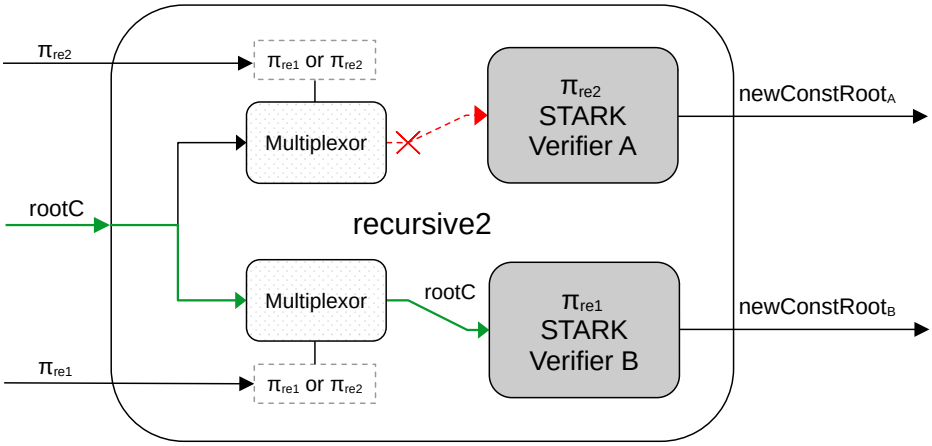

Figure 17: `recursive2` circuit.

The output circom file `recursive2.circom`, obtained running a different script called `genrecursive` it is compiled into a R1CS `recursive2.r1cs` file and a witness calculator program `recursive2.witnesscal` which will be both used later on in order to build and fill the next execution trace.

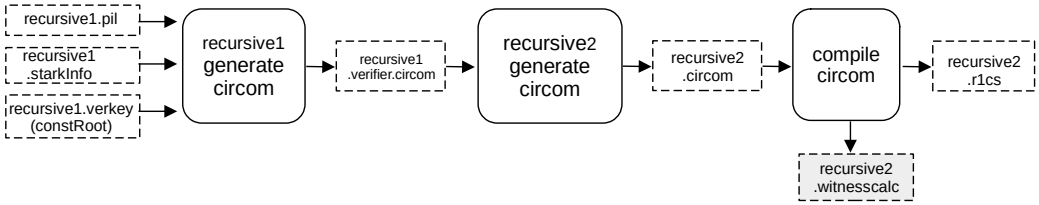

Figure 18: Convert the `recursive1` STARK to its verifier circuit called `recursive2`.

4.2.7 Setup `C2S` for `recursive2`

As before, from the previous R1CS description of the verification circuit we are going to obtain a machine-like construction whose correct execution, which will be described by a PIL `recursive2.pil` file, will be equivalent to the validity of the previous circuit. Also, a binary for all the constant polynomials `recursive2.const` defined by it and the helper file providing the witness values allocation into its corresponding position of the execution trace `recursive2.exec` are generated.

In order to finish the set up phase, having all the FRI-related parameters willing to be used in this step stored in a `recursive.starkstruct` file (located in the prover repository), we can generate the `recursive2.starkinfo` file through the `generate_starkinfo` service and build the constants' tree (and its respectively constant root). Since we want both

verifiers to be exactly the same as in the previous step, we are using the same blowup factor of $2^4 = 16$, allowing the number of queries to be 32.

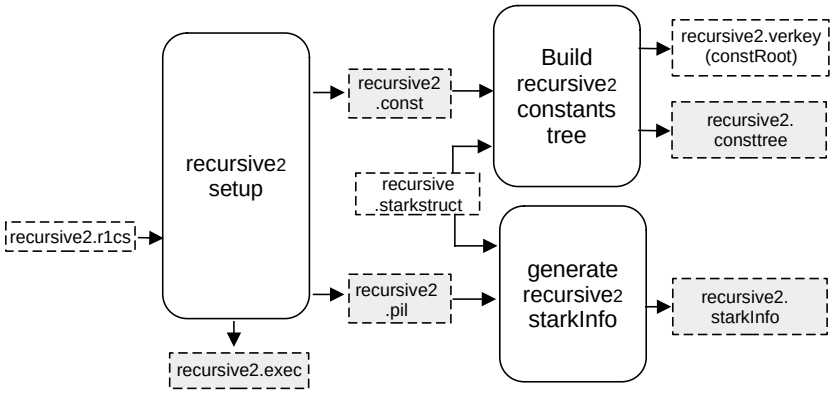

Figure 19: Convert the `recursive2` circuit to its associated STARK.

4.2.8 Setup `S2C` for `recursivef`

Up to this point we have a STARK proof π_{re2} verifying another π_{re2} proof (or π_{re1} in some edge cases). The idea now is, as before, generate a Circom circuit that verifies π_{re2} (or π_{re1} if no aggregation is taking place) by miming the FRI verification procedure as done before. To do that, we generate a verifier circuit `recursive2.verifier.circom` from the previously obtained `recursive2.pil` file, the `recursive2.starkinfo` file and the constant roots of the previous two proofs `recursive2_a.verkey.constRoot` and `recursive2_b.verkey.constRoot` by filling the `stark_verifier.circom.ejs` template as before.

The output circom file `recursivef.circom`, obtained running a different script called `genrecursivef` it is compiled into a R1CS `recursivef.r1cs` file and a witness calculator program `recursivef.witnesscal` which will be both used later on in order to build and fill the next execution trace.

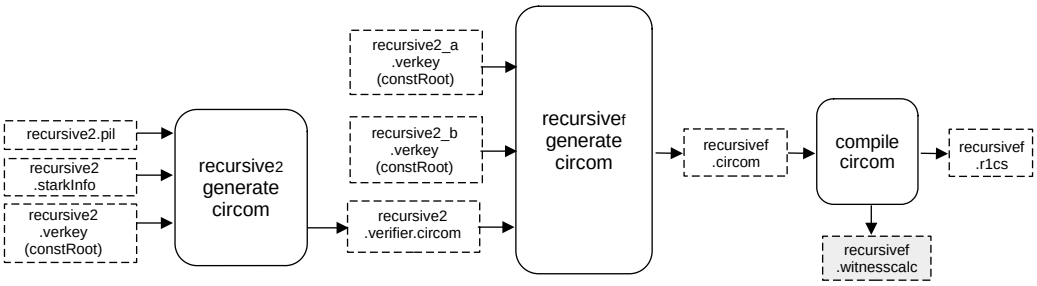

Figure 20: Convert the `recursive2` STARK to its verifier circuit called `recursivef`.

4.2.9 Setup `C2S` for `recursivef`

As before, from the R1CS description of the verification circuit we are going to obtain a machine-like construction whose correct execution, which will be described by a `recursive2.pil` PIL file, will be equivalent to the validity of the previous circuit. Also, a binary for all the constant polynomials `recursive2.const` defined by it and the helper file providing the witness values allocation into its corresponding position of the execution trace `recursive2.exec` are generated.

In order to finish the set up phase, having all the FRI-related parameters willing to be used in this step stored in a `recursivef.starkstruct` file (located in the prover repository), we can generate the `recursivef.starkinfo` file through the `generate_starkinfo` service and build the constants' tree (and its respectively constant root).

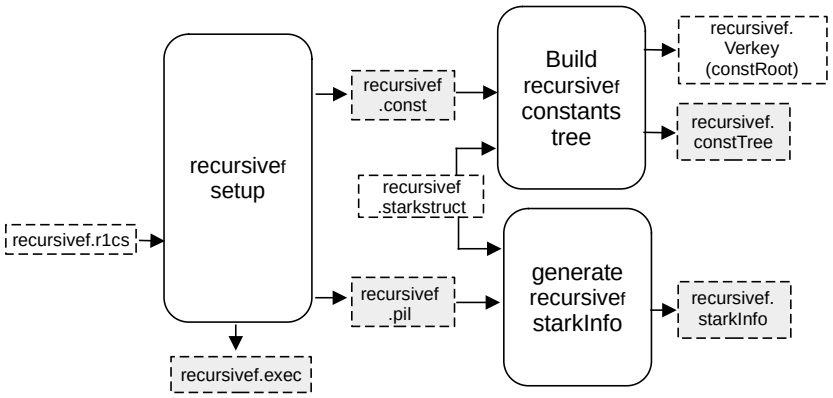

Figure 21: Convert the recursivef circuit to its associated STARK.

4.2.10 Setup `S2C` for `final`

Up to this point we have a STARK proof π_{ref} verifying a proof π_{re2}. As before, we generate a Circom circuit that verifies π_{ref} by miming the FRI verification procedure. To do that, we generate a verifier circuit `recursivef.verifier.circom` from the previously obtained `recursivef.pil` file, the `recursivef.starkinfo` file and the constant root `recursivef.verkey.constRoot` by filling the `stark_verifier.circom.ejs` verifier template. This verifier Circom file will be imported by the `final.circom` circuit in order to generate the circuit that will be proven using Groth16 procedure.

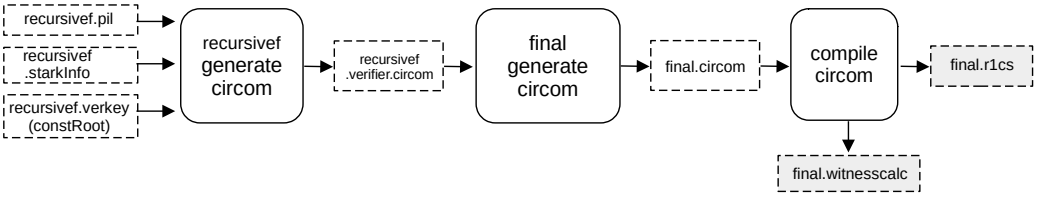

Figure 22: Convert the `recursivef` STARK to its verifier circuit that is called `final.circom`.

4.3 Proof Generation Phase

4.3.1 Proof of the zkEVM STARK

Up to this point we built an execution trace together with a PIL file describing the ROM of the zkEVM. Having both we can generate a STARK proof stating the correct execution of the zkEVM using the `pil-stark` tooling explained in section 2.3. In this step, a blowup factor of 2 is used, so the proof having a huge amount of polynomials becomes quite big. To amend that, the next step `c12a` will be a compression step, raising the blowup factor and aiming to reduce the number of polynomials.

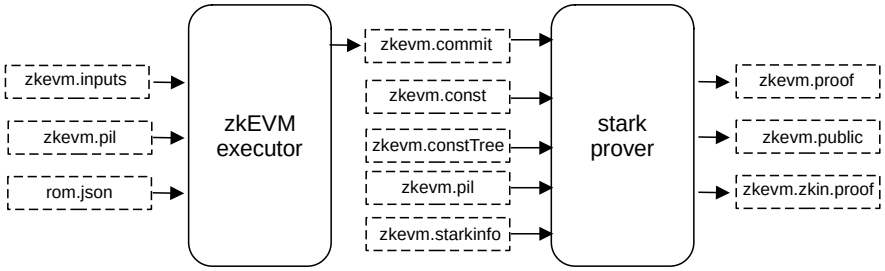

Figure 23: Generation for a zkEVM Proof

To generate the proof, `main_prover` service is used. The service requires to provide the execution trace (that is, the committed and constant polynomials files generated by the executor using the `pilcom` package), the constant tree binary file in order to be hashed to construct the constant root, the PIL file of the zkEVM ROM `zkev.pil` and all the information provided by the `zkevm.starkinfo.json` file, including all the FRI-related parameters such as the blowup factor or the configuration of the steps.

This step differs from the next ones as it is the first and it is intended to start the recursion. However, aiming uniformization code-wise, the Main Prover procedure choose to abstract the notion of proving and is intended to be the same at each step among the recursion.

4.3.2 Proof of `c12a`

To generate the proof verifying the previous `zkevm.proof`, we generate all the witness values and map them correctly into its corresponding position of the execution trace exactly in the same way as before, obtaining a binary file `ca12.commit` for the committed polynomials of the execution trace.

Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, we can generate a proof validating the previous big STARK proof. We use the same service `main_prover` as before to do it, providing also the previously built constant tree `ca12.constTree` and the `ca12.starkinfo` file. This will generate the proof and the publics joined in the `c12a.zkin.proof` file.

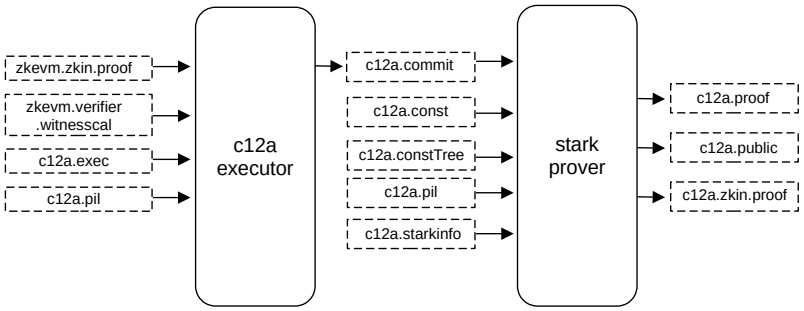

Figure 24: Generate a STARK proof for `c12a`.

4.3.3 Proof of `recursive1`

To generate the proof verifying the previous `c12a.proof`, we generate all the witness values and map them correctly into its corresponding position of the execution trace exactly in the same way as before, obtaining a binary file `recursive1.commit` for the committed polynomials of the execution trace.

Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, we can generate a proof validating the previous big STARK proof. We use the same service `main_prover` as before to do it, providing also the previously built constant tree `recursive1.constTree` and the `recursive1.starkinfo` file. This will generate the proof and the publics joined in the `recursive1.zkin.proof` file.

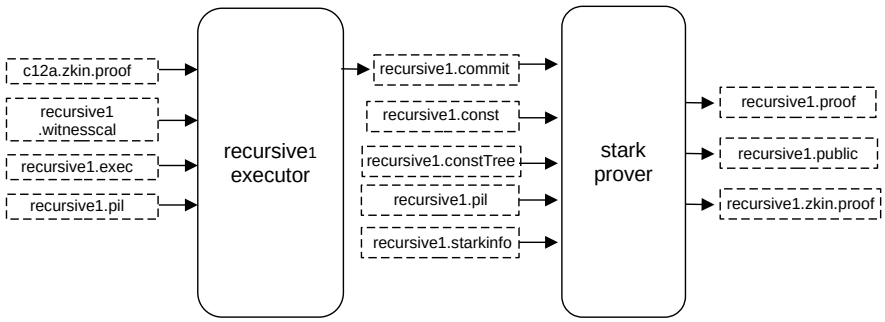

Figure 25: Generate a STARK proof for `recursive1`.

4.3.4 Proof of `recursive2`

To generate the proof verifying the previous `recursive1.proof`, we generate all the witness values and map them correctly into its corresponding position of the execution trace exactly in the same way as before, obtaining a binary file `recursive2.commit` for the committed polynomials of the execution trace.

Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, we can generate a proof validating the previous big STARK proof. We use the same service `main_prover` as before to do it, providing also the previously built constant tree `recursive2.constTree` and the `recursive2.starkinfo` file. This will generate the proof and the publics joined in the `recursive2.zkin.proof` file.

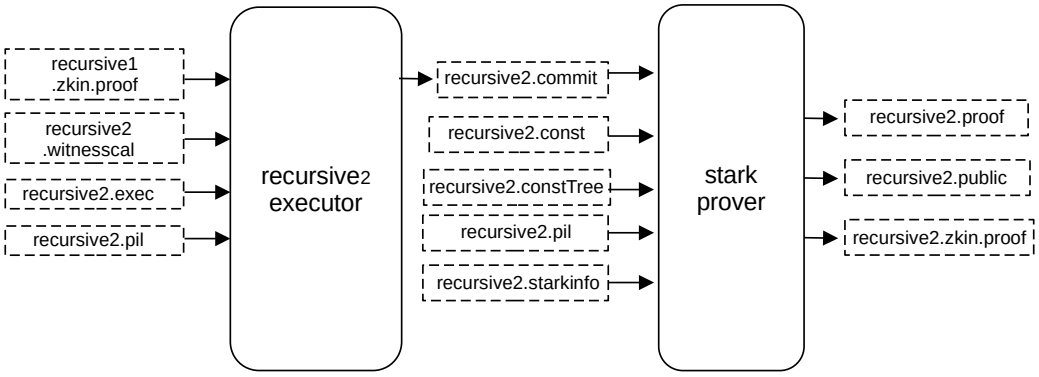

Figure 26: Generate a STARK proof for `recursive2`.

4.3.5 Proof of `recursivef`

To generate the proof verifying the previous `recursive2.proof`, we generate all the witness values and map them correctly into its corresponding position of the execution trace exactly in the same way as before, obtaining a binary file `recursivef.commit` for the committed polynomials of the execution trace.

Having the execution trace (that is, the committed and constant polynomials filled) and the PIL, we can generate a proof validating the previous big STARK proof. We use the

same service `main_prover` as before to do it, providing also the previously built constant tree `recursivef.constTree` and the `recursivef.starkinfo` file. This will generate the proof and the publics joined in the `recursivef.zkin.proof` file.

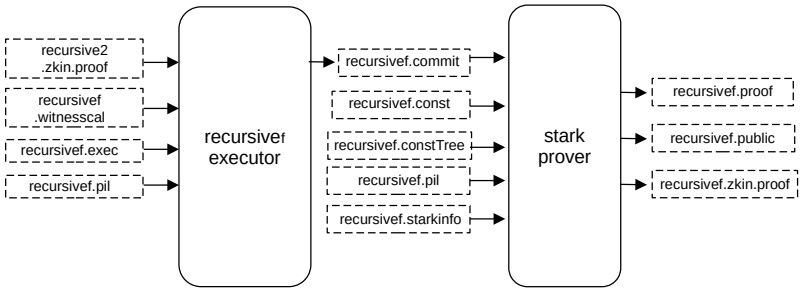

Figure 27: Generate a STARK proof for `recursivef`.

4.3.6 Proof of `final`

The last circuit, `final.circom` is the one used to generate the proof. At this moment a Groth16 proof is generated.

5 Remarks

The setup phase runs with the proverjs. The proof generation runs with the prover written in C. The circuits build in the setup phase can be used as many times as desired. The prover receives the information about the particular composition of proofs with an RPC API.

References