polygon zkEVM

Technical Document

Recursion, aggregation and composition of proofs v.1.1

June 21, 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*.

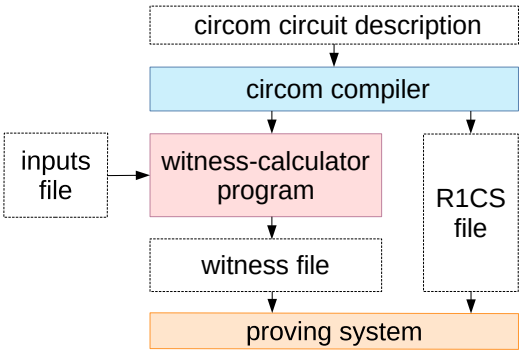

Figure 1: Circom

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.

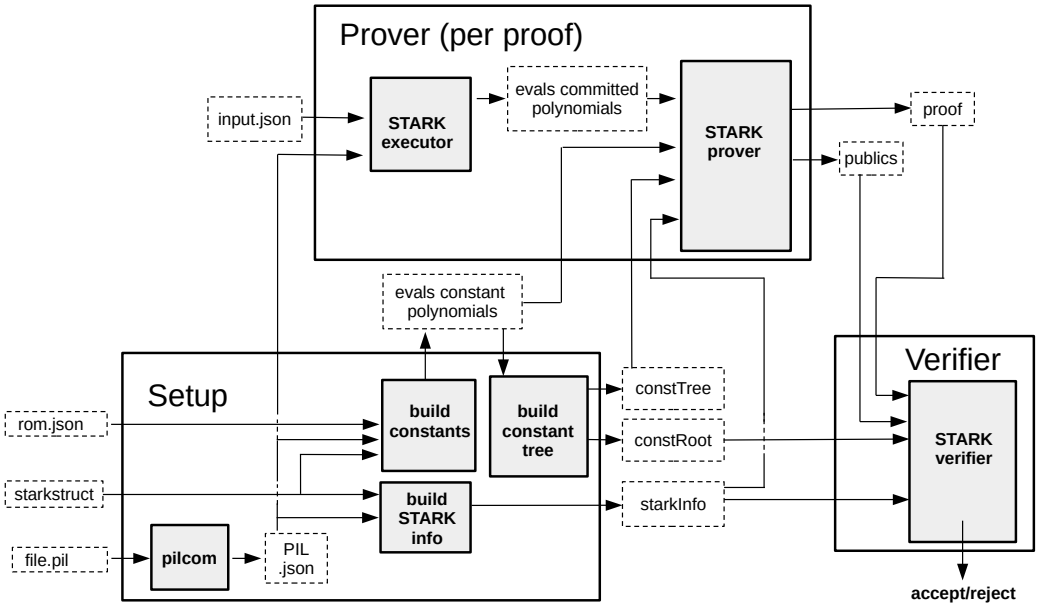

Figure 2: Non recursive STARK.

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}.

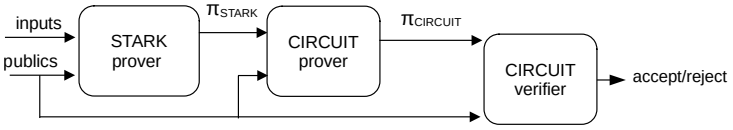

Figure 3: Simple composition.

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.

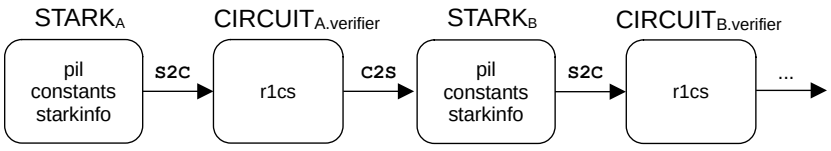

Figure 4: Recursion setup.

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.

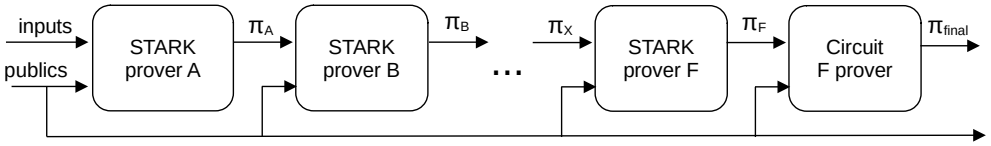

Figure 5: Recursive provers.

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.

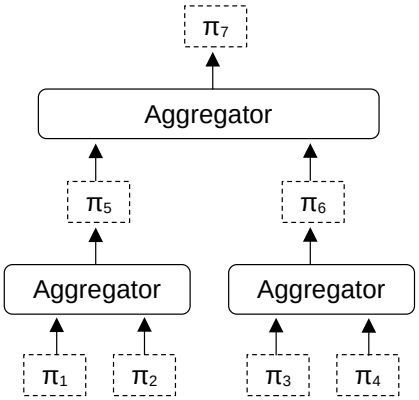

Figure 6: Aggregation example.

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.

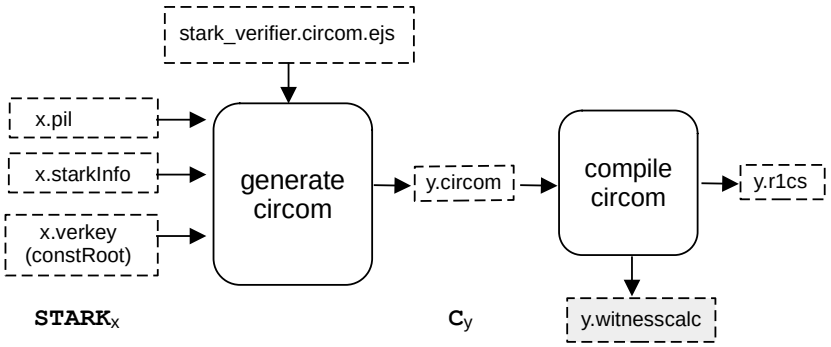

Figure 7: Setup S2C.

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.

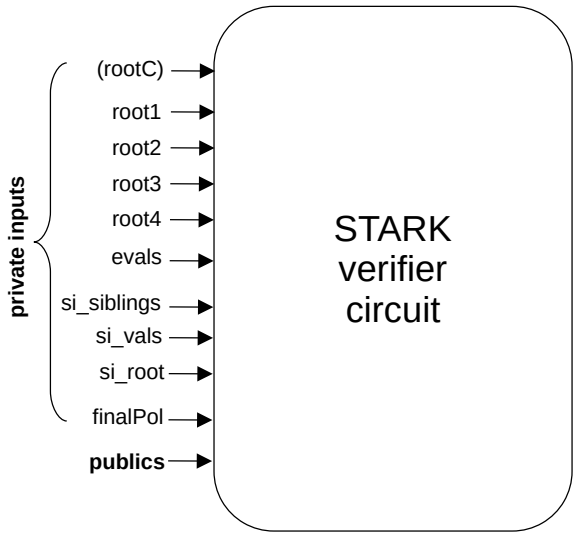

Figure 8: Inputs of the STARK verifier circuits.

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 ($\texttt{STARK}_\texttt{y}$).

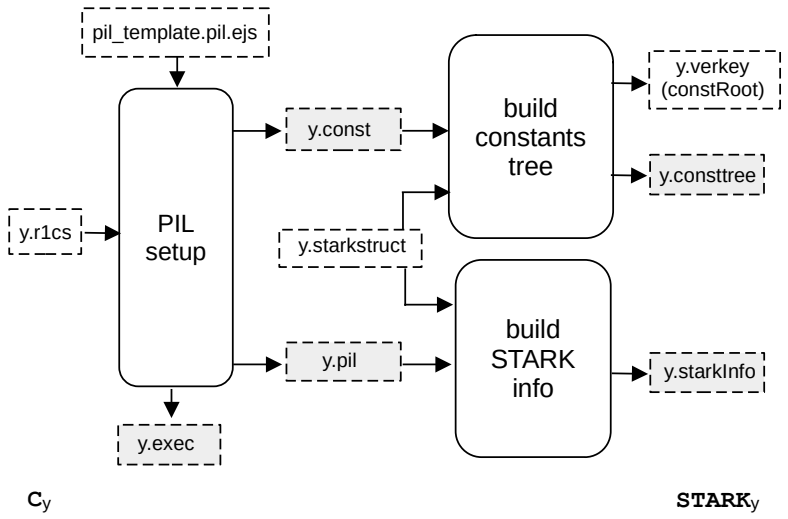

Figure 9: Setup recursion step `C2S`.

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`). For a complete description of the the `c12` arithmetization refer to Section 6.

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.

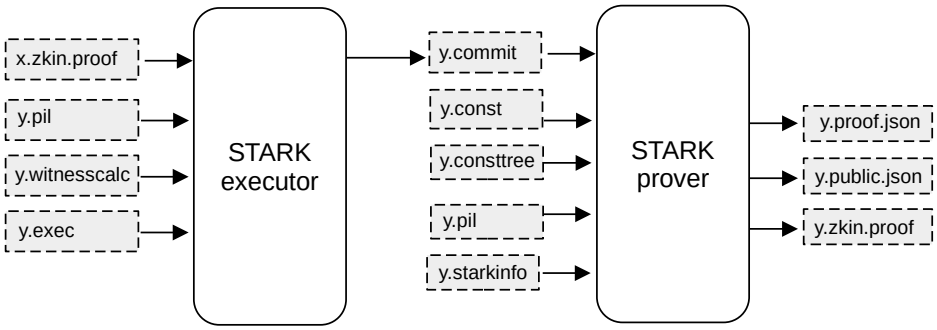

Figure 10: STARK `Proof` of a recursion step.

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.

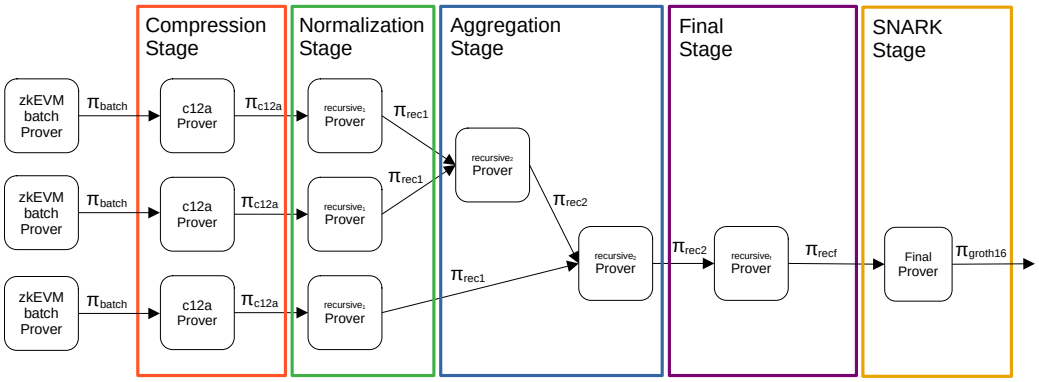

Figure 11: Proving architecture with recursion, aggregation and composition.

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.

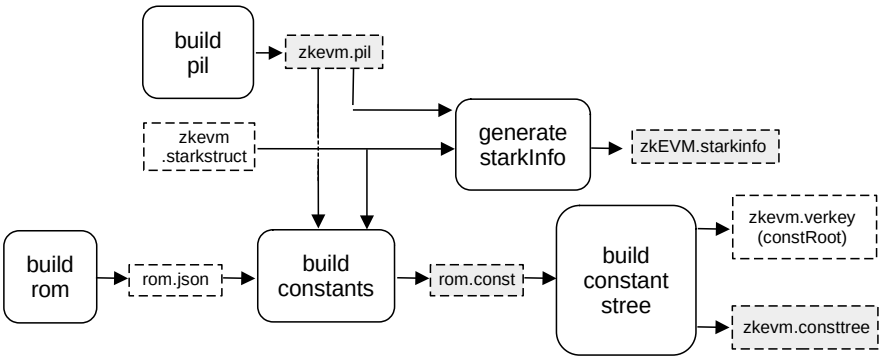

Figure 12: Build the zkEVM STARK

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).

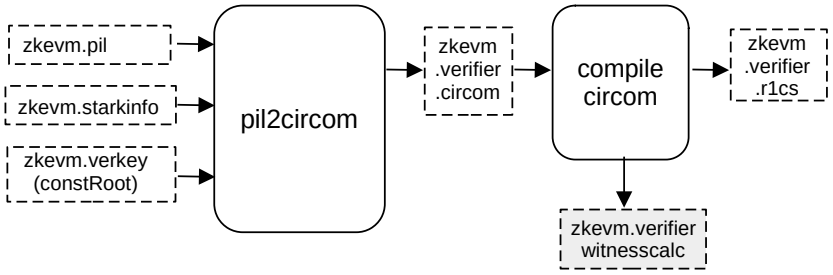

Figure 13: Converting the zkEVM STARK verification into a circuit.

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 $\mathcal{P}$lon$\mathcal{K}$ish circuit with custom gates and 12 polynomials aiming at compression.

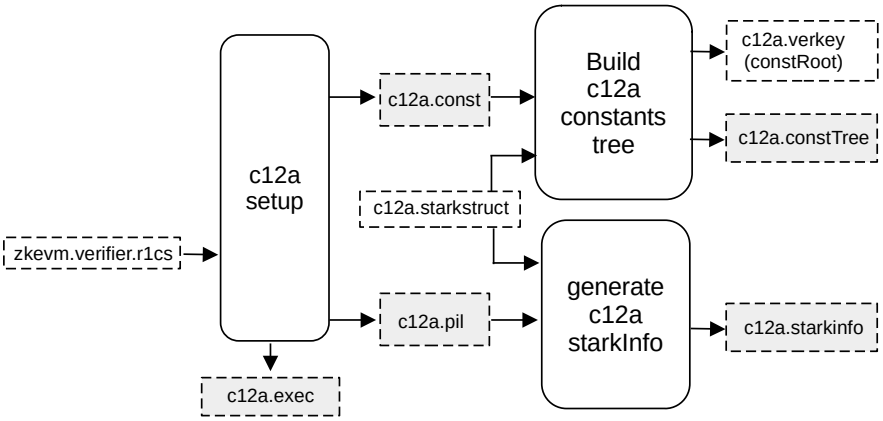

Figure 14: Convert the zkEVM verifier circuit to a STARK called `c12a`.

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.

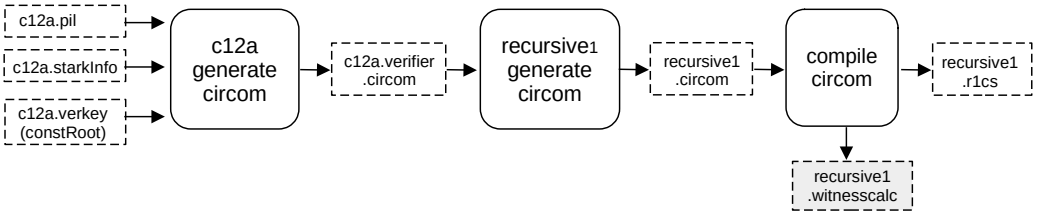

Figure 15: Convert the c12a STARK to a c12a verifier circuit.

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.

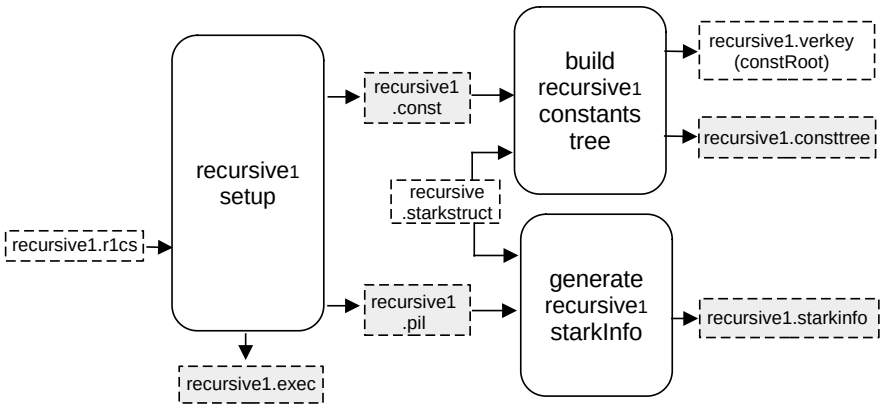

Figure 16: Convert the `recursive1` circuit to its associated STARK.

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.

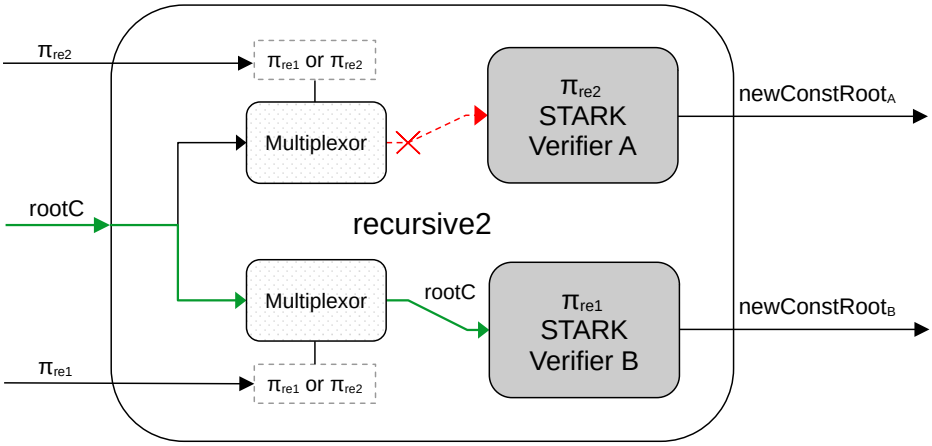

Figure 17: `recursive2` circuit.

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.

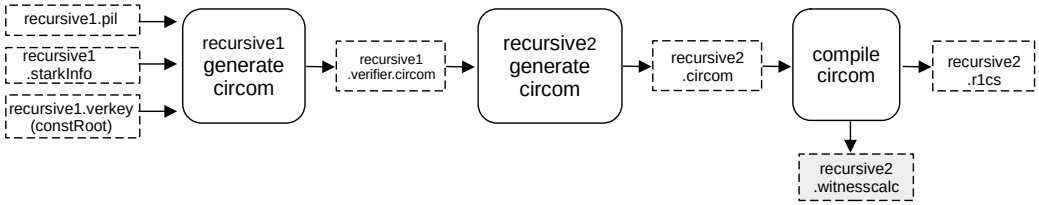

Figure 18: Convert the `recursive1` STARK to its verifier circuit called `recursive2`.

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.

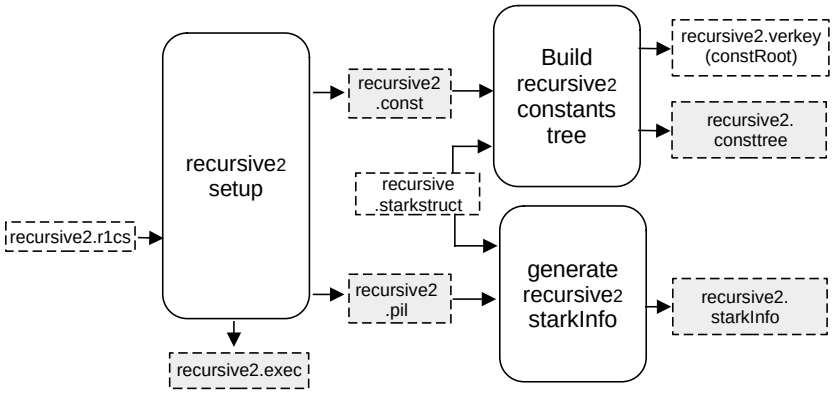

Figure 19: Convert the `recursive2` circuit to its associated STARK.

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.

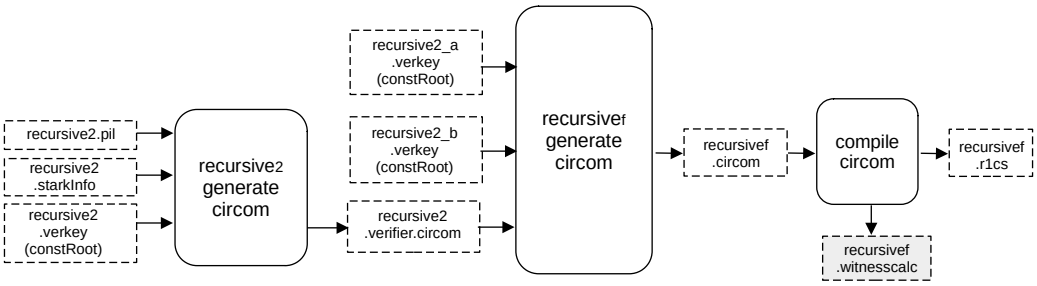

Figure 20: Convert the `recursive2` STARK to its verifier circuit called `recursivef`.

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).

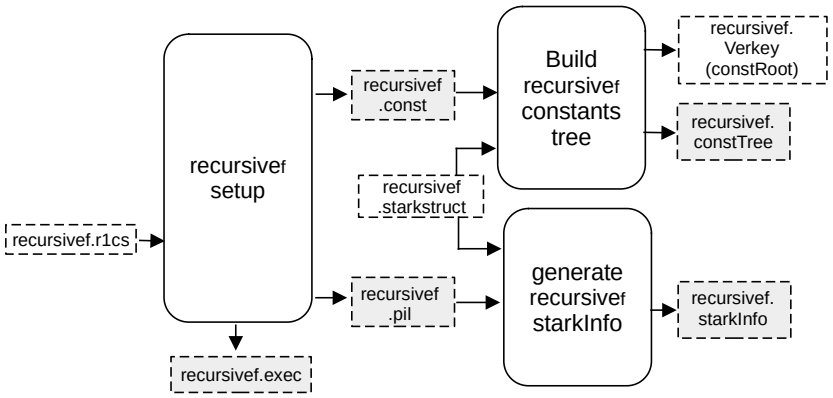

Figure 21: Convert the recursivef circuit to its associated STARK.

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.

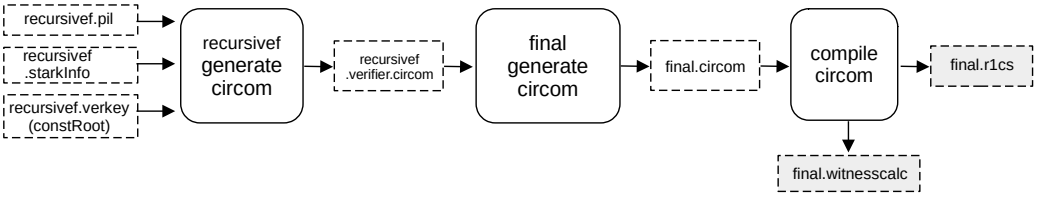

Figure 22: Convert the `recursivef` STARK to its verifier circuit that is called `final.circom`.

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.

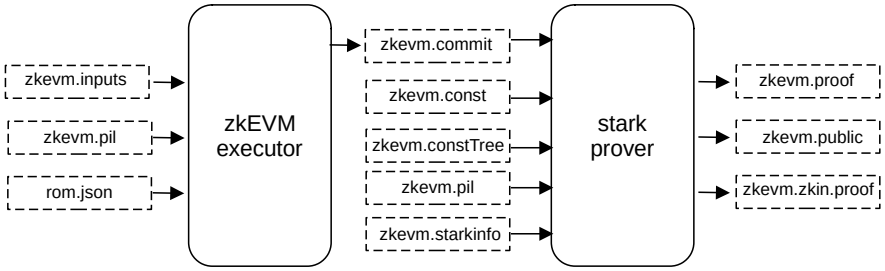

Figure 23: Generation for a zkEVM Proof

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.

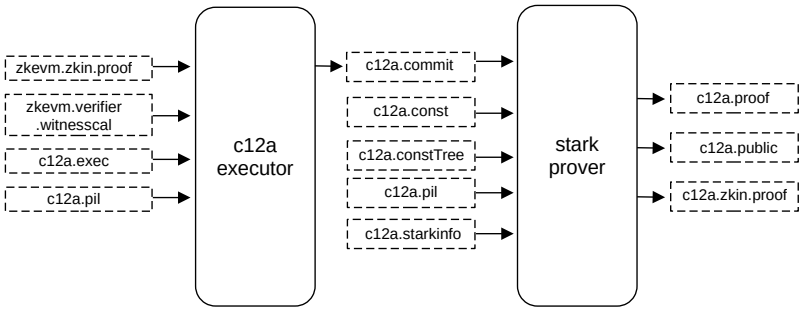

Figure 24: Generate a STARK proof for `c12a`.

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.

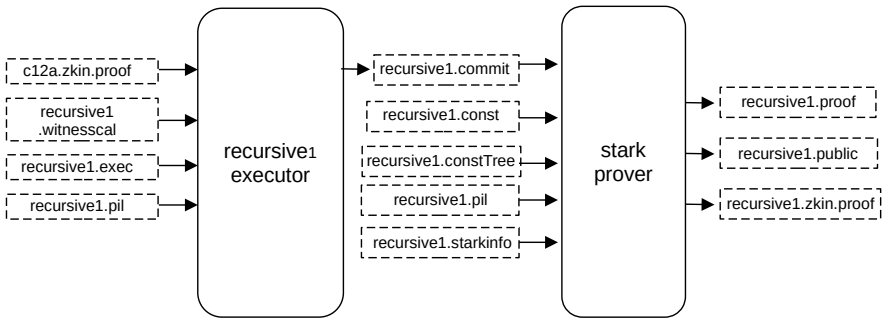

Figure 25: Generate a STARK proof for `recursive1`.

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.

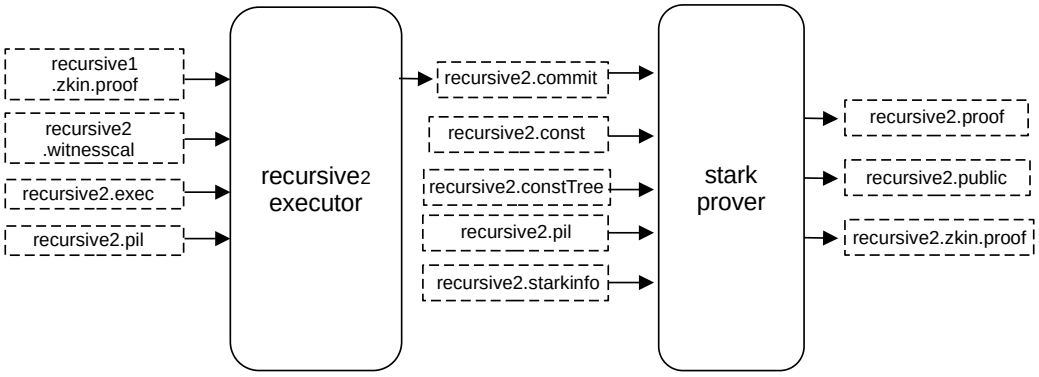

Figure 26: Generate a STARK proof for `recursive2`.

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.

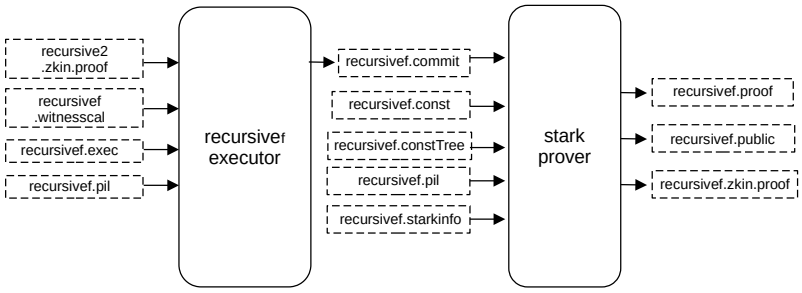

Figure 27: Generate a STARK proof for `recursivef`.

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 Building zkEVM Proofs

The setup phase runs with the proverjs. The circuits build in the setup phase can be used as many times as desired. In production, the proof generation runs with the prover written in C++. The prover receives the information about the particular composition of proofs via an gRPC API. Next we explain how to use each software component in more detail.

5.1 Preprocessing and Initialitiation

5.1.1 Requirements

In order to verify the Smart Contract, there will be necessary a machine with at least 256GB of RAM and 16 cores. This section's tutorial will give instructions for a `r6a.8xlarge` AWS instance, with 16 cores, 32 threads and 512GB of SSD. This instance uses Ubuntu 22.04 LTS costing 1.82 dollars per hour.

5.1.2 Setup

First of all, we ought prepare the OS in order to be able to run the verification procedure: we update the apt packages and install `tmux`, `git` and `curl`.

```
sudo apt update
sudo apt install -y tmux git curl
```

Posteriorly, the kernel configuration parameters should be changed in order to accept high amount of memory:

```
echo "vm.max_map_count=655300" | sudo tee -a /etc/sysctl.conf
sudo sysctl -w vm.max_map_count=655300
export NODE_OPTIONS="--max-old-space-size=230000"
```

Now, we install the `NodeSource Node.js 18.x` repository onto our Ubuntu machine. **It is important to check that the main version of Node is `18`.**

```
curl -sL "https://deb.nodesource.com/setup_18.x" -o nodesource_setup.sh
sudo bash nodesource_setup.sh
sudo apt install -y nodejs
node -v
```

Moreover, in order to be able to compile circuits involved, we need `circom` installed:

```
cd ~
git clone https://github.com/iden3/circom.git
cd circom
git checkout v2.1.4
git log --pretty=format:'%H' -n 1
```

The hasth of the commit should be

`ca3345681549c859af1f3f42128e53e3e43fe5e2`.

Now, we install and compile `cicom` using Rust's package manage and build system: `cargo`. It is important to check that the version of `circom` is `2.1.4`:

```
sudo apt install -y cargo
cargo build --release
cargo install --path circom
export PATH=$PATH:~/.cargo/bin
echo 'PATH=$PATH:~/.cargo/bin' >> ~/.profile
circom --version
```

In order to build the tree of constants in a fast way we have integrated a tool called `bctree`. We can build it using the following set of commands:

```
cd ~
git clone https://github.com/0xPolygonHermez/zkevm-prover.git
cd zkevm-prover
git checkout b9179f3b870c2922b11d09462b447d0fc35dd2f9
git submodule init
git submodule update
sudo apt install -y build-essential libomp-dev libgmp-dev nlohmann-json3-dev
     libpqxx-dev nasm libgrpc++-dev libprotobuf-dev grpc-proto libsodium-dev
     uuid-dev libsecp256k1-dev
make -j bctree
```

This step takes approximately 8 minutes to complete.

5.1.3 Preprocessing

The next step consists on building all the setup steps specified before in this document, such as building all constants and circuits. This can be done using the code below. Observe that we are actually using the previously built `bcee` tool above in the last command, under the `-bctree` flag.

```
cd ~
git clone https://github.com/0xPolygonHermez/zkevm-proverjs.git
cd zkevm-proverjs
git checkout 59694e4a8a9358772c23e8110f53aee049e2d5bd
npm install
tmux -c "npm run buildsetup --bctree=../zkevm-prover/build/bctree"
```

This step is quite long, it takes approximately 4.5 hours. 2 out of 4.5 hours are for the `powersOfTau28_hez_final.ptau` download, a file of 288GB that it's loaded only once.

5.2 Verification of the Smart Contract

As a final result of the previous steps, the smart contract that verifies the test has been generated. This file is `final.fflonk.verifier.sol`. At this point, it is possible to verify the smart contract using the source code or verify that the bytecode is the same.

5.2.1 Option 1: Verify the Smart Contract using the Source Code

To verify the bytecode, you must compile with the precisely same version, compiler, and parameters to be sure that even the metadata hash contained in the bytecode is exactly the same. The following instructions generate a project to build using the hardhat tool.

```
cd ~
mkdir contract
cd contract
npm init -y
npm install hardhat
mkdir -p contracts/verifiers
echo -e "module.exports={solidity:{compilers:[{version: \"0.8.17\",settings
    :{optimizer:{enabled:true,runs:999999}}}]}}" > hardhat.config.js
```

Once the project structure is created, we proceed to copy the smart contract generated in the previous step. This smart contract was saved on `~/zkevm-proverjs/build/proof`, and must be copied to `contracts/verifiers` with exactly the name `FflonkVerifier.sol`. If the name or the path changes, the hash of metadata changes too, for this reason, is essential to respect the exact name and path. This can be done by executing these commands:

```
cp ~/zkevm-proverjs/build/proof/final.fflonk.verifier.sol contracts/
    verifiers/FflonkVerifier.sol
sha256sum contracts/verifiers/FflonkVerifier.sol
```

We should double check that the output hash digest for the `FflonkVerifier.sol` file is exactly the one shown below:

`60a850bd73a316d6f85bfdf05bc69ee01f84d889004e98b07847719f59e49761.`

Once this is done and check, we can compile the Smart Contract using the following command:

```
npx hardhat compile
```

The bytecode of the Smart Contract was on the `bytecode` property of the `.json` file `FflonkVerifier.json` generated on path `artifacts/contracts/verifiers/FflonkVerifier.sol`:

```
608060405234801561001057600080fd5b50612a
078061002060003960 00f3fe6080604052348015
61001057600080fd5b506004361061002b576000
:
:
017fffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffe0908116603f01
168101908382118183101715612987576129876 1
28aa565b816040528281528860208487010111 15
6129a057600080fd5b826020860160208301376 0
006020848301015280965050505050506129c884
602085016128d9565b9050925092905056fea264
6970667358221220f7f0438c1c741f762e101c4e
ae732373046d40de3d0f2e0808e1fac1d863c78f
64736f6c63430008110033
```

The following command allows us to extract previous bytecode in one line of a file `FflonkVerifier.sol.compiled.bytecode`:

```
grep bytecode artifacts/contracts/verifiers/FflonkVerifier.sol/
    FflonkVerifier.json |sed 's/.*"\(0x.*\)".*/\1/' > FflonkVerifier.sol.
    compiled.bytecode
```

If you prefer you can copy by hand the content of the bytecode of the file

`artifacts/contracts/verifiers/FflonkVerifier.sol/FflonkVerifier.json`

to the file `FflonkVerifier.sol.compiled.bytecode`. Remember to copy only the content inside the double quotes (without double quotes).

Then, use the hash-digest to verify integrity of the compiled bytecode

```
sha256sum FflonkVerifier.sol.compiled.bytecode
```

The expected result is the following one

`44f70b6a1548f49214027169db477f03e92492f19ac5594c648edb478316cbec.`

5.2.2 Option 2: Verify the Smart Contract comparing the Bytecode

To download bytecode of deployed Smart Contract we need his address. In this case it's `0x4ceB990D2E2ee6d0e163fa80d12bac72C0F28D52`. Then, we can direct to Etherscan, Blockscout or Beaconcha to get transaction bytecode.

Some applications running on the terminal may limit the amount of input they will accept before their input buffers overflow. To avoid this situation create file `FflonkVerifier.sol.explorer.bytecode` with an editor such as `nano` or `vi`.

```
cd ~/contract
nano FflonkVerifier.sol.explorer.bytecode
```

Then, paste the clipboard to the file and compare that the two files providing to both Option 1 and Option 2 are actually the same. We can do it using a `diff`:

```
cd ~/contract
diff FflonkVerifier.sol.compiled.bytecode FflonkVerifier.sol.explorer.
    bytecode
```

Or using a hash file integrity verification method:

```
cd ~/contract
sha256sum FflonkVerifier.sol.*.bytecode
```

The output of the previous command in the terminal should be

```
44f70b6a1548f49214027169db477f03e92492f19ac5594c648edb478316cbec
    FflonkVerifier.sol.compiled.bytecode
44f70b6a1548f49214027169db477f03e92492f19ac5594c648edb478316cbec
    FflonkVerifier.sol.explorer.bytecode
```

5.3 zkEVM Proving

5.3.1 Prover JS

The `main_executor` script located in the `src` folder is used to generate the committed polynomials `commit.bin` of the execution trace for a compiled `rom.json` file coming from a `rom.zkasm` file by the zkASM compiler using a specific set of inputs from a file `input.json`. The common use of this script is shown below:

```
node src/main_executor <input.json> -r <rom.json> -o <commit.bin>
```

Below, we list all the possible additional parameters that can be added to modify its usage:

- `-t<test.json>`: Test.
- `-l<logs.json>`: Output for logs.
- `-s`: Skip `.pil` file compilation.
- `-d`: Debug mode.
- `-p`: select a concrete PIL program. For example, `pilprogram.pil`.
- `-P`: Load `pilConfig.json` file.
- `-u`: Unsigned transactions mode.
- `-e`: Skip asserts `newStateRoot` and `newLocalExitRoot`.
- `-v`: Verbose mode.

The `buildall` command is used as a main entrypoint to all the build setup steps specified in the previous sections.

```
npm run buildall
```

Below, we describe how to use it in full detail, listing all the involved flags:

- `--pil=<pil_file.pil>`: This option specifies the name of the PIL file to compile. For example, `--pil=src/main.pil`.
- `--pilconfig=<pilconfig.json>`: This option specifies the name of the PIL configuration file to use. The PIL configuration file contains various settings and options that affect the compilation process. For example, `--pilconfig=config/pilconfig.json`.
- `--starkstruct=debug`: This option generates an automatic STARK struct adapted to the PIL bits number. This option is typically used for debugging purposes.
- `--from=<step>`: This option specifies the step at which to start the build/rebuild process. For example, `--from=c12setup`.
- `--continue`: This option continues the build process from where it left off during the previous build.
- `--step=<step>`: This option specifies single specific build step to execute. For example, `--step=c12setup`.
- `--build=<build_dir>`: This option specifies the name of the build directory to use. The build directory is where the compiled output files are stored. For example, `--build=build/basic_proof`.
- `--input=<input_file>`: This option specifies the name of the input file to use for the build process. For example, `--input=test/myinputfile.json`.

5.3.2 Prover C++

The code and the instructions to compile and run the zkEVM Prover can be found at 0xPolygonHermez/zkevm-prover GitHub repository. The zkEVM Prover process can provide up to three RPC services.

The **Executor service** calls the Executor component of the zkEVM Prover to generate an execution trace for a set of L2 transactions. The execution trace can be used to prove the new state resulting after executing all the involved transactions. However, this service does not generate the proof, just the execution trace. This service is used as fast way to check if a proposed batch or batches of transactions are properly built and if they fit in a single execution trace. In other words, if the transactions of the batch or batches fit into a single proof. The interface of this service is defined at the file executor.proto.

The **StateDB service** provides an interface to access to database of the L2 state. This state is succinctly represented by the root of a Merkle tree and the service provides the corresponding data and their Merkle proofs. This service is used by the Executor component and by the Prover component and it represents the single source of truth for the current state. Among others, we can obtain account balances, nonce values, contract deployted bytecode, storage and so on. The interface of this service is defined at the file statedb.proto.

Finally, the **Aggregator service** offers the highest level of service among all the services provided by the zkEVM prover. In more detail, this service can create a proof for a single batch or a proof for multiple batches if the fit in the execution trace. In addition, this service is able to create an aggregated proof if a pair of independent proofs of consecutive batches are provided. When the Aggregator service is called to generate a proof, the service uses the Executor component to create the corresponding execution trace and from this trace the associated committed polynomials. Finally, with the pre-processed and committed polynomials the proof is generated. When called with a pair of proofs of consecutive batches the service generates an aggregated proof. The interface of this service is defined at the file aggregator.proto.

6 C12 PIL Description

Previously, we provided a brief overview of the high-level functioning of the `c12` arithmetization, highlighting its general properties and some insights of its PIL description. Now, our objective in this section is to provide a comprehensive and detailed explanation of the `c12` PIL arithmetization utilized during the generation and the constraining of the execution trace for the verification process of a FRI protocol.

6.1 From `r1cs` to $\mathcal{P}\mathfrak{lonK}$

Recall that the verifier procedure verifying a generated STARK is written in `circom`. This procedure is responsible for constructing a set of `r1cs` (Rank-1 Constraint System) constraints. Additionally, it is crucial to incorporate the utilization of custom gates that were specified earlier. Observe that custom gates are not directly checked in `circom` and the `.r1cs` file only contains computational information about its inputs and outputs. Consequently, the verification process for custom gates must be performed explicitly within the generated `pil` code that describes the verifier's verification procedure. The `pil` code will be responsible for incorporating the necessary validations for the custom gates, ensuring that they function as intended within the overall verification process.

In contrast to $\mathcal{P}\mathfrak{lonK}$ constraints, it is not straightforward to translate `r1cs` constraints into a fixed column execution trace along with a set of `pil` constraints. Therefore, it becomes necessary to perform a conversion from `r1cs` constraints to $\mathcal{P}\mathfrak{lonK}$ ones in order to proceed. In what follows, we describe this conversion.

Recall that a (m-dimensional) Rank-1 Constraint System (`r1cs`) over a set of signals $s_1, \ldots, s_n$ is composed of three matrices $A = (a_{i,j}), B = (b_{i,j}), C = (c_{i,j})$ each beloving to the matrix space $\mathrm{M}(m, n, \mathbb{F})$ where $\mathbb{F}$ represents the underlying field. We say that $s = (s_1, \ldots, s_n)$ satisfy the constraints if and only if

$$As \circ Bs = Cs.$$

where $\circ$ denotes the Hadamard product (or component-wise multiplication). More explicitly, $s_1, \ldots, s_n$ satisfies the constraints if and only if

$$\begin{gathered}(a_{1,1}s_1 + \cdots + a_{1,n}s_n) \cdot (b_{1,1}s_1 + \cdots + b_{1,n}s_n) = c_{1,1}s_1 + \cdots + c_{1,n}s_n \\ \cdots \\ (a_{m,1}s_1 + \cdots + a_{m,n}s_n) \cdot (b_{m,1}s_1 + \cdots + b_{m,n}s_n) = c_{m,1}s_1 + \cdots + c_{m,n}s_n\end{gathered}$$

Based on the previous explanation, it can be noted that the constraint $(1 + s_2) \cdot s_3 = s_4$ is not allowed in the R1CS formulation. In order to address the inclusion of constants in constraints, a designated signal s_1 is introduced to always maintain a value of 1.

Notice that we can perceive `r1cs` constraints as gates with an unbounded fan-in (meaning they can have any number of input wires) and an unbounded fan-out (meaning they can have any number of output wires). This flexibility allows `r1cs` to accommodate a wide range of computations. On the other hand, $\mathcal{P}\mathfrak{lon}\mathcal{K}$ gates operate with a fixed fan-in of 2 (meaning that each gate has exactly two input wires) and fan-out of 1 (meaning that each gate has exactly one output wire). A single $\mathcal{P}\mathfrak{lon}\mathcal{K}$ gate with inputs signals a, b and output signal c is determined by 5 selectors, namely q_R, q_L, q_M, q_O and q_C and we say that the tuple (a, b, c) satisfies the gate equation if and only if

$$a_R \cdot a + q_L \cdot b + q_M \cdot a \cdot b + q_O \cdot c + q_C = 0.$$

The transformation process from a Rank-1 Constraint System to a set of $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints involves mapping each individual `r1cs` constraint to one or more $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints. To ensure a correct, optimized and comprehensive conversion, the reduction process can be divided into distinct cases, each addressing a specific scenario. Within each case, specific rules and mappings will be defined to convert the `r1cs` constraints into the appropriate $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints. These rules take into consideration the unique characteristics of each case, such as the types of variables involved, the operations performed, and the desired properties in the resulting $\mathcal{P}\mathfrak{lon}\mathcal{K}$ constraints. The strategy for all of them consists on reducing sums adding constraints and variables. More specifically, consider the following linear combination $a_1 + a_2s_2 + \cdots + a_ns_n$, which has $n - 1$ non-constant therms. We can reduce this linear combination to contain $n-2$ non-constant therms adding another artificial signal, say v_1, and the constraint $a_{n-1}s_{n-1} + a_ns_n = v_1$ (with the corresponding $\mathcal{P}\mathfrak{lon}\mathcal{K}$ selectors $q_L = -a_{n-1}, q_R = -a_n, q_M = 0, q_O = 1$ and $q_C = 0$). Replacing this into the linear combination we get a new reduced linear combination $a_1 + a_2s_2 + \cdots + v_1$. We can perform the same strategy recursively to further reduce the linear combination to one having only two therms $a_1 + v_{T_a}$, where T_a denotes the total number of newly added variables. Let us discuss the complete strategy for each of the cases:

- In constraints where $A = 0$ and/or $B = 0$, we get a linear constraint of the form

  $$c_1 + c_2s_2 + \cdots + c_ns_n = 0.$$

  The idea to optimally attack this case is to reduce apply the reduction strategy in order to reduce the number of additions to 4 (3 non-constant factors and one reserved for the constant). More specifically, the reduced constraint will look like:

  $$c_1 + c_2s_2 + c_3s_3 + v_T = 0,$$

where T denotes the total amount of newly introduced signals. At this point we can directly convert the upper constraint into $\mathcal{P}\texttt{lonK}$ by setting $q_R = c_2, q_L = c_3, q_M = 0, q_O = 1$ and $q_C = c_1$.

- In constraints with $a_i = 0$ except when $i = 1$ (corresponding exactly to the constant wire), we get a linear constraint of the form

$$a_1 \cdot (b_1 + b_2 s_2 + \cdots + b_n s_n) = c_1 + c_2 s_2 + \cdots + c_n s_n.$$

In this case, In this case, the strategy is to multiply the constant a_1 with the linear combination and combine both sides of the equation:

$$\begin{aligned} &a_1 \cdot (b_1 + b_2 s_2 + \cdots + b_n s_n) = c_1 + c_2 s_2 + \cdots + c_n s_n \iff \\ &a_1 b_1 + a_1 b_2 s_2 + \cdots + a_1 b_n s_n = c_1 + c_2 s_2 + \cdots + c_n s_n \iff \\ &a_1 b_1 + a_1 b_2 s_2 + \cdots + a_1 b_n s_n - c_1 - c_2 s_2 - \cdots - c_n s_n = 0 \end{aligned}$$

Now, we proceed as before and we apply the reduction strategy in order to reduce the number of additions to 4 (3 non-constant factors and one reserved for the constant). More specifically, the reduced constraint will look like:

$$a_1 b_1 + a_1 b_2 s_2 + a_1 b_3 s_3 + a_1 v_T = 0,$$

where T denotes the total amount of newly introduced signals. At this point we can directly convert the upper constraint into $\mathcal{P}\texttt{lonK}$ by setting $q_R = a_1 \cdot b_2, q_L = a_1 \cdot b_3, q_M = 0, \acute{\text{I}} q_O = a_1$ and $q_C = a_1 b_1$.

- In constraints with $b_i = 0$ except when $i = 1$ (corresponding exactly to the constant wire), we get a linear constraint of the form

$$(a_1 + a_2 s_2 + \cdots + a_n s_n) \cdot b_1 = c_1 + c_2 s_2 + \cdots + c_n s_n.$$

These constraints are handled symmetrically to the previous method described.

- In this case fit all the other scenarios, so it deals with general constraints

$$(a_1 + a_2 s_2 + \cdots + a_n s_n) \cdot (b_1 + b_2 s_2 + \cdots + b_n s_n) = c_1 + c_2 s_2 + \cdots + c_n s_n$$

excluding the cases where each a_i or b_i is zero or where only a_1 (or b_1) differs from zero. In this case, the solution is to reduce each of the linear combinations to a single coefficient and a constant, say:

$$\begin{aligned} &(a_1 + a \cdot v_{T_a}) \cdot (b_1 + b \cdot v_{T_b}) = c_1 + c \cdot v_{T_c} \iff \\ &(a_1 + b_1 - c_1) + a_1 \cdot b v_{T_b} + b_1 \cdot a \cdot v_{T_a} + a \cdot b \cdot v_{T_a} \cdot v_{T_b} - c \cdot v_{T_c} = 0 \end{aligned}$$

We can directly convert the previous constraint to $\mathcal{P}\texttt{lonK}$ by setting $q_L = a \cdot b_1, q_R = a_1 \cdot b, q_M = a \cdot b, q_C = a_1 + b_1 - c_1$ and $q_O = -c$.

6.2 C12 Plonk Gates Verification

Once the `r1cs` has been transformed into a set of $\mathcal{P}\texttt{lonK}$ constraints, we can proceed to generate an execution trace along with a corresponding `pil` file for its verification. To ensure optimal verification of the `POSEIDON12` custom gates, it is preferable to use an execution trace with a width of 12 columns. Consequently, the state update of a single round of a Poseidon hash can be verified by examining the transition between two rows, without requiring additional rows.

However, when verifying regular $\mathcal{P}\mathfrak{lonK}$ gates in `pil` using 12 columns, we encounter a wastage of 7 constant columns. This wastage occurs because we only require 5 columns to accommodate the constants of the constraint. To address this issue, we can utilize 2 sets of constraints in each row, enabling the verification of 2 $\mathcal{P}\mathfrak{lonK}$ constraints within a single row. It's important to note that $\mathcal{P}\mathfrak{lonK}$ gates have a fan-in of 2 and a fan-out of 1. Consequently, by utilizing only 2 sets of signals per row, we end up wasting 6 witness columns. To address this issue, we can employ a technique that involves reusing constraints by utilizing the connection arguments available in `pil`. This approach allows us to optimize the allocation of witness columns and minimize wastage.

The idea is simple. At a single row of our execution trace we will have two sets of $\mathcal{P}\mathfrak{lonK}$ constraints, namely $Q = \{q_L, q_R, q_O, q_M, q_C\}$ and $Q' = \{q'_L, q'_R, q'_O, q'_M, q'_C\}$. The initial six witness columns ($\mathtt{a}[0], \ldots, \mathtt{a}[5]$) will correspond to Q, while the remaining six witness columns ($\mathtt{a}[6], \ldots, \mathtt{a}[11]$) will relate to Q'. More specifically, the following constraints will be necessary to be fulfilled:

$$
\begin{aligned}
\mathtt{a}[0] \cdot q_L + \mathtt{a}[1] \cdot q_R + \mathtt{a}[2] \cdot q_O + \mathtt{a}[0] \cdot \mathtt{a}[1] \cdot q_M + q_C = 0 \\
\mathtt{a}[3] \cdot q_L + \mathtt{a}[4] \cdot q_R + \mathtt{a}[5] \cdot q_O + \mathtt{a}[3] \cdot \mathtt{a}[4] \cdot q_M + q_C = 0 \\
\mathtt{a}[6] \cdot q'_L + \mathtt{a}[7] \cdot q'_R + \mathtt{a}[9] \cdot q'_O + \mathtt{a}[6] \cdot \mathtt{a}[7] \cdot q'_M + q'_C = 0 \\
\mathtt{a}[9] \cdot q'_L + \mathtt{a}[10] \cdot q'_R + \mathtt{a}[11] \cdot q'_O + \mathtt{a}[9] \cdot \mathtt{a}[10] \cdot q'_M + q'_C = 0
\end{aligned}
$$

In `pil` code, this is translated to:

```
pol a01 = a[0]*a[1];
pol g012 = C[3]*a01 + C[0]*a[0] + C[1]*a[1] + C[2]*a[2] + C[4];
g012*GATE = 0;

pol a34 = a[3]*a[4];
pol g345 = C[3]*a34 + C[0]*a[3] + C[1]*a[4] + C[2]*a[5] + C[4];
g345*GATE = 0;

pol a67 = a[6]*a[7];
pol g678 = C[9]*a67 + C[6]*a[6] + C[7]*a[7] + C[8]*a[8] + C[10];
g678*GATE = 0;

pol a910 = a[9]*a[10];
pol g91011 = C[9]*a910 + C[6]*a[9] + C[7]*a[10] + C[8]*a[11] + C[10];
g91011*GATE = 0;
```

However, to ensure soundness and achieve the desired order of witness signals, we need to utilize connection arguments. These gates enable us to verify that the signals appearing in a specific $\mathcal{P}\mathfrak{lonK}$ constraint are accurate.

If we fail to assert this, the prover could incorrectly claim that:

$$s_1 + s_2 = s_3 \quad \text{and} \quad s_4 - s_5 = s_6$$

while the actual $\mathcal{P}\mathfrak{lonK}$ constraint is:

$$s_4 + s_5 = s_6 \quad \text{and} \quad s_1 - s_2 = s_3$$

Without ensuring the correct ordering of the witness signals, the prover could manipulate the equations and present an inaccurate representation of the $\mathcal{P}\mathfrak{lonK}$ constraint. By asserting the correct connection between the witness signals and the constraints, we prevent such incorrect claims and guarantee the accurate representation of the desired $\mathcal{P}\mathfrak{lonK}$ constraint. To establish this, we employ the following `pil` constraint:

```
{a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10], a
    [11]} connect
    {S[0], S[1], S[2], S[3], S[4], S[5], S[6], S[7], S[8], S[9], S[10],
        S[11]};
```

where the `S` polynomials keeps track of the exact permutation that corresponds to the position of the constraints we have deliberately placed along the execution trace.

6.3 Poseidon Round Verification

Whenever `POSEIDON12` is 1, the PIL file will check that the vector

$$(a[0]', a[1]', a[2]', a[3]', a[4]', a[5]', a[6]', a[7]', a[8]', a[9]', a[10]', a[11]')$$

is obtained applying the `POSEIDON` permutation to the vector

$$(a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9], a[10], a[11]).$$

Recall that the `POSEIDON` permutation has two modes: the partial permutation and the full one. The partial permutation is characterized by only-applying the S-Box layer to one element of the state (the first one, for example). However, a full round applies the S-Box to the whole set of 12 elements which forms the state. Hence, we need a polynomial called `PARTIAL` which will distinguish if the current round is a partial round or a full round. More concretely, `PARTIAL` constant polynomial will be 1 if and only if the current round is a partial round and 0 otherwise.

First of all, we will compute, if necessary, the 7-th power of each element of the state in order to compute the S-Box layer. Moreover, we will add, at the beggining of the computation, the corresponding constant specified by the `Poseidon` permutation, which are stored in the polynomials `C[i]`. We will use the following code (written using `ejs`):

```
pol a<%- i %>_1 = a[<%- i %>] +  C[<%- i %>];
pol a<%- i %>_2 = a<%- i %>_1 * a<%- i %>_1;
pol a<%- i %>_4 = a<%- i %>_2 * a<%- i %>_2;
pol a<%- i %>_6 = a<%- i %>_4 * a<%- i %>_2;
pol a<%- i %>_7 = a<%- i %>_6 * a<%- i %>_1;
<%      if (i==0) { -%>
pol a<%- i %>_R = a<%- i %>_7;
<%      } else { -%>
pol a<%- i %>_R = PARTIAL * (a<%- i %>_1 - a<%- i %>_7) + a<%- i %>_7;
<%      } -%>
<% } -%>
```

We will carry `ai_R` to the next phase of the permutation to multiply it by the corresponding MDS matrix. Observe that we always exponentiate when $i = 0$ (that is, the first element of the state). However, the other elements are only exponentiated if and only if `PARTIAL` is 1. The last part of the validation is trivial to validate:

```
POSEIDON12 * (a[0]'  - (25*a0_R + 15*a1_R + 41*a2_R + 16*a3_R +  2*a4_R
    + 28*a5_R + 13*a6_R + 13*a7_R + 39*a8_R + 18*a9_R + 34*a10_R + 20*
    a11_R)) = 0;
POSEIDON12 * (a[1]'  - (20*a0_R + 17*a1_R + 15*a2_R + 41*a3_R + 16*a4_R
    +  2*a5_R + 28*a6_R + 13*a7_R + 13*a8_R + 39*a9_R + 18*a10_R + 34*
    a11_R)) = 0;
POSEIDON12 * (a[2]'  - (34*a0_R + 20*a1_R + 17*a2_R + 15*a3_R + 41*a4_R
    + 16*a5_R +  2*a6_R + 28*a7_R + 13*a8_R + 13*a9_R + 39*a10_R + 18*
    a11_R)) = 0;
POSEIDON12 * (a[3]'  - (18*a0_R + 34*a1_R + 20*a2_R + 17*a3_R + 15*a4_R
    + 41*a5_R + 16*a6_R +  2*a7_R + 28*a8_R + 13*a9_R + 13*a10_R + 39*
    a11_R)) = 0;
POSEIDON12 * (a[4]'  - (39*a0_R + 18*a1_R + 34*a2_R + 20*a3_R + 17*a4_R
    + 15*a5_R + 41*a6_R + 16*a7_R +  2*a8_R + 28*a9_R + 13*a10_R + 13*
    a11_R)) = 0;
POSEIDON12 * (a[5]'  - (13*a0_R + 39*a1_R + 18*a2_R + 34*a3_R + 20*a4_R
    + 17*a5_R + 15*a6_R + 41*a7_R + 16*a8_R +  2*a9_R + 28*a10_R + 13*
    a11_R)) = 0;
POSEIDON12 * (a[6]'  - (13*a0_R + 13*a1_R + 39*a2_R + 18*a3_R + 34*a4_R
    + 20*a5_R + 17*a6_R + 15*a7_R + 41*a8_R + 16*a9_R +  2*a10_R + 28*
    a11_R)) = 0;
POSEIDON12 * (a[7]'  - (28*a0_R + 13*a1_R + 13*a2_R + 39*a3_R + 18*a4_R
    + 34*a5_R + 20*a6_R + 17*a7_R + 15*a8_R + 41*a9_R + 16*a10_R +  2*
    a11_R)) = 0;
POSEIDON12 * (a[8]'  - ( 2*a0_R + 28*a1_R + 13*a2_R + 13*a3_R + 39*a4_R
    + 18*a5_R + 34*a6_R + 20*a7_R + 17*a8_R + 15*a9_R + 41*a10_R + 16*
    a11_R)) = 0;
POSEIDON12 * (a[9]'  - (16*a0_R +  2*a1_R + 28*a2_R + 13*a3_R + 13*a4_R
    + 39*a5_R + 18*a6_R + 34*a7_R + 20*a8_R + 17*a9_R + 15*a10_R + 41*
    a11_R)) = 0;
POSEIDON12 * (a[10]' - (41*a0_R + 16*a1_R +  2*a2_R + 28*a3_R + 13*a4_R
    + 13*a5_R + 39*a6_R + 18*a7_R + 34*a8_R + 20*a9_R + 17*a10_R + 15*
    a11_R)) = 0;
POSEIDON12 * (a[11]' - (15*a0_R + 41*a1_R + 16*a2_R +  2*a3_R + 28*a4_R
    + 13*a5_R + 13*a6_R + 39*a7_R + 18*a8_R + 34*a9_R + 20*a10_R + 17*
    a11_R)) = 0;
```

Numbers above are determined by the used MDS matrix of the permutation. More precisely, we are checking the matrix product shown below:

$$\begin{pmatrix} 25 & 15 & 41 & 16 & 2 & 28 & 13 & 13 & 39 & 18 & 31 & 20 \\ 20 & 17 & 15 & 41 & 16 & 2 & 28 & 13 & 13 & 39 & 18 & 34 \\ 34 & 20 & 17 & 15 & 41 & 16 & 2 & 28 & 13 & 13 & 39 & 18 \\ 18 & 34 & 20 & 17 & 15 & 41 & 16 & 2 & 28 & 13 & 13 & 39 \\ 39 & 18 & 34 & 20 & 17 & 15 & 41 & 16 & 2 & 28 & 13 & 13 \\ 13 & 39 & 18 & 34 & 20 & 17 & 15 & 41 & 16 & 2 & 28 & 13 \\ 13 & 13 & 39 & 18 & 34 & 20 & 17 & 15 & 41 & 16 & 2 & 28 \\ 28 & 13 & 13 & 39 & 18 & 34 & 20 & 17 & 15 & 41 & 16 & 2 \\ 2 & 28 & 13 & 13 & 39 & 18 & 34 & 20 & 17 & 15 & 41 & 16 \\ 16 & 2 & 28 & 13 & 13 & 39 & 18 & 34 & 20 & 17 & 15 & 41 \\ 41 & 16 & 2 & 28 & 13 & 13 & 39 & 18 & 34 & 20 & 17 & 15 \\ 15 & 41 & 16 & 2 & 28 & 13 & 13 & 39 & 18 & 34 & 20 & 17 \end{pmatrix} \cdot \begin{pmatrix} \texttt{a0\_R} \\ \texttt{a1\_R} \\ \texttt{a2\_R} \\ \texttt{a3\_R} \\ \texttt{a4\_R} \\ \texttt{a5\_R} \\ \texttt{a6\_R} \\ \texttt{a7\_R} \\ \texttt{a8\_R} \\ \texttt{a9\_R} \\ \texttt{a10\_R} \\ \texttt{a11\_R} \end{pmatrix} = \begin{pmatrix} \texttt{a[0]'} \\ \texttt{a[1]'} \\ \texttt{a[2]'} \\ \texttt{a[3]'} \\ \texttt{a[4]'} \\ \texttt{a[5]'} \\ \texttt{a[6]'} \\ \texttt{a[7]'} \\ \texttt{a[8]'} \\ \texttt{a[9]'} \\ \texttt{a[10]'} \\ \texttt{a[11'} \end{pmatrix}$$

6.4 Extended Field Operations Verification

Whenever `CMULADD` is 1, the PIL file will check that the following elements of $\mathbb{F}_{p^3}$

$$\begin{aligned} a &= \mathtt{a}[0] + \mathtt{a}[1] \cdot X + \mathtt{a}[2] \cdot X^2 \\ b &= \mathtt{a}[3] + \mathtt{a}[4] \cdot X + \mathtt{a}[5] \cdot X^2 \\ c &= \mathtt{a}[6] + \mathtt{a}[7] \cdot X + \mathtt{a}[8] \cdot X^2 \\ \mathtt{output} &= \mathtt{a}[9] + \mathtt{a}[10] \cdot X + \mathtt{a}[11] \cdot X^2 \end{aligned}$$

satisfy the relationship

$$a \cdot b + c = \mathtt{output}$$

using the field operations inherited from

$$\mathbb{F}_{p^3} \cong \mathbb{F}_p[X]/(X^3 - X - 1).$$

Given two elements $a_0 + a_1X + a_2X^2, b_0 + b_1X + b_2X^2 \in \mathbb{F}_{p^3}$, we can express its product by computing the product of the polynomials, using Euclidean division and taking equivalence classes in $\mathbb{F}_{p^3}$. It is not difficult to see, then, that we can express its product as

$$\begin{aligned} &(a_0 \cdot b_0 + a_1 \cdot b_2 + a_2 \cdot b_1) + \\ &(a_0 \cdot b_1 + a_1 \cdot b_0 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2) \cdot X + \\ &(a_0 \cdot b_2 + a_2 \cdot b_2 + a_2 \cdot b_0 + a_1 \cdot b_1) \cdot X^2 \end{aligned}$$

Hence, the PIL code below states the correctness of the output elements $\mathtt{a}[9], \mathtt{a}[10], \mathtt{a}[11]$ using polynomial relationships of degree less or equal than 2 using the operation defined above:

```
pol cA = (a[0] + a[1]) * (b[0] + b[1]);
pol cB = (a[0] + a[2]) * (b0 + b[2]);
pol cC = (a[1] + a[2]) * (b[1] + b[2]);
pol cD = a[0] * b[0];
pol cE = a[1] * b[1];
pol cF = a[2] * b[2];

CMULADD * (a[9] - (cC + cD - cE - cF) - c0) = 0;
CMULADD * (a[10] - (cA + cC - 2*cE - cD) - c1) = 0;
CMULADD * (a[11] - (cB - cD + cE) - c2) = 0;
```

The previous piece of code works since:

$$\begin{aligned} \mathtt{cA} &= (\mathtt{a0} + \mathtt{a1}) \cdot (\mathtt{b0} + \mathtt{b1}) = \mathtt{a0} \cdot \mathtt{b0} + \mathtt{a0} \cdot \mathtt{b1} + \mathtt{a1} \cdot \mathtt{b0} + \mathtt{a1} \cdot \mathtt{b1} \\ \mathtt{cB} &= (\mathtt{a0} + \mathtt{a2}) \cdot (\mathtt{b0} + \mathtt{b2}) = \mathtt{a0} \cdot \mathtt{b0} + \mathtt{a0} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b0} + \mathtt{a2} \cdot \mathtt{b2} \\ \mathtt{cC} &= (\mathtt{a1} + \mathtt{a2}) \cdot (\mathtt{b1} + \mathtt{b2}) = \mathtt{a1} \cdot \mathtt{b1} + \mathtt{a1} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b1} + \mathtt{a2} \cdot \mathtt{b2} \end{aligned}$$

Hence

$$\begin{aligned} \mathtt{a}[9] &= \mathtt{cC} + \mathtt{cD} - \mathtt{cE} - \mathtt{cF} + \mathtt{c0} = \mathtt{a0} \cdot \mathtt{b0} + \mathtt{a1} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b1} \\ \mathtt{a}[10] &= \mathtt{cA} + \mathtt{cC} - 2\mathtt{cE} - \mathtt{cD} + \mathtt{c1} = \mathtt{a0} \cdot \mathtt{b1} + \mathtt{a1} \cdot \mathtt{b0} + \mathtt{a1} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b1} + \mathtt{a2} \cdot \mathtt{b2} \\ \mathtt{a}[11] &= \mathtt{cB} - \mathtt{cD} + \mathtt{cE} + \mathtt{c2} = \mathtt{a0} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b2} + \mathtt{a2} \cdot \mathtt{b0} + \mathtt{a1} \cdot \mathtt{b1} \end{aligned}$$

6.5 Fast Fourier Transform Verification

Whenever `FFT4` is 1, the PIL file will check the correct computation of a Fast Fourier Transform using the Coley-Tucker's butterfly method. Depending on the custom gate being

validated, the input of the FFT can be 2 elements or 4 elements. Take into account that we apply the FFT to extended field elements, consisting on 3 field elements. Depending on the number of input elements, the constants for the computation `C` are adjusted in order to mimic the butterfly's formulas. Moreover, a parameter called `scale` is introduced in order to be able to perform inverse Fast Fourier Transforms.

However, we should be able to perform much bigger FFTs, so the custom gate is programmed in order to optimize the computation using the necessary number of four-size FFT and, if necessary, complete them with the complementary number of two-sized FFT (observe that only 1 of them may be necessary). Hence, we would also need a mechanism to chain them correspondingly following the butterfly diagram.

6.5.1 How to Chain FFTs

First of all suppose that we want to perform the Fast Fourier Transform of n elements in the base field $\mathbb{F}_p$ where n is a power of two having its exponent $\log_2(n)$ even. We are choosing a even exponent for the power of two since we will discuss first the case where no 2-sized FFT are needed. The idea of the FFT is to reuse computations in order to reduce complexity. This can be seen easily using the butterfly diagram. Below, we will show the butterfly diagram for the case $n = 16$. We can reduce the total computation of the FFT for 16 elements to a total amount of 8 FFT of 4 elements. However, chained FFT needs to be readapted and well-connected in order to express the correct computation. Observe that, before the first step, we should reverse the bits of the polynomial coefficients' indices in order to correctly order them all.

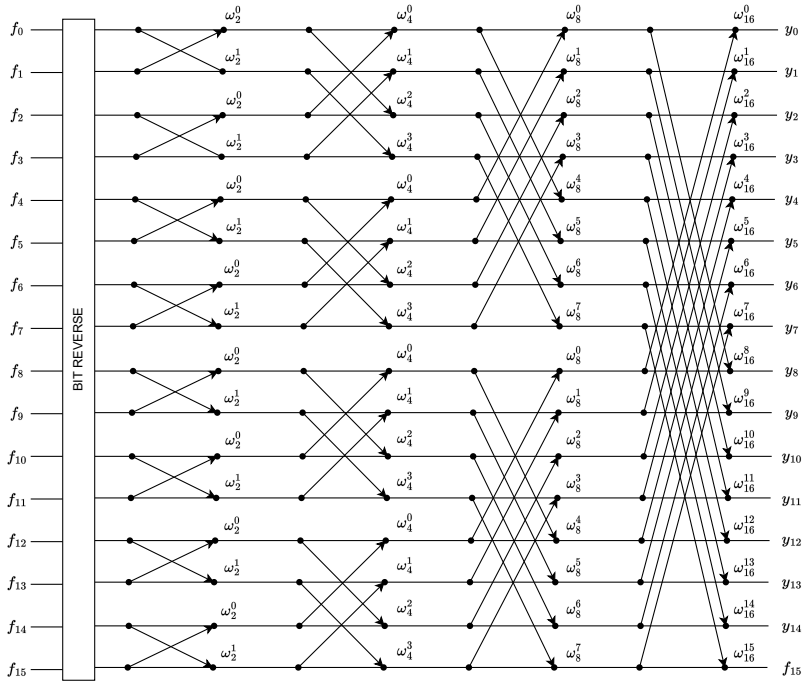

Figure 28: Butterfly diagram for 16 elements' Fast Fourier Transform.

Let us denote each of the $r + 1$ states of the n-sized Fast Fourier Transform as $(f_0^k, \ldots, f_{n-1}^k)$, where $r = \log_2(n)$. The initial state $(f_0^0, \ldots, f_{n-1}^0)$, representing the input, is labeled as step 0, while the final state $(f_0^r, \ldots, f_{n-1}^r)$, representing the output, is labeled as step r. Alternatively and aiming to clear pictures, we will also use $f_0, \ldots, f_{n-1}$ for the input and $y_0, \ldots, y_{n-1}$ for the output of the whole FFT. First of all, recall that, when

fixing a certain step $k \in \{0, \ldots, r-1\}$, the next state $(f_0^{k+1} \ldots, f_{n-1}^{k+1})$ of the butterfly structure can be directly computed from the previous one $(f_0^k, \ldots, f_{n-1}^k)$ as follows, for $j \in \{0, \ldots, n/2-1\}$:

$$f_j^{k+1} = f_j^k + f_{j+2^k}^k \cdot \omega_{2^{k+1}}^j,$$
$$f_{j+2^k}^{k+1} = f_j^k + f_{j+2^k}^k \cdot \omega_{2^{k+1}}^{j+2^k}$$

being $\omega_{2^{k+1}} \in \mathbb{F}_p^*$ a primitive 2^{k+1}-root of unity. However, we can optimize the computation by exploiting the fact that:

$$-\omega_{2^{k+1}}^j = \omega_{2^{k+1}}^{j+2^k} \quad \text{for } j \in \{0, \ldots, 2^k - 1\}.$$

By utilizing this property, we can avoid calculating half of the powers of $\omega_{2^{k+1}}$. Hence, we get the following optimized butterfly diagram:

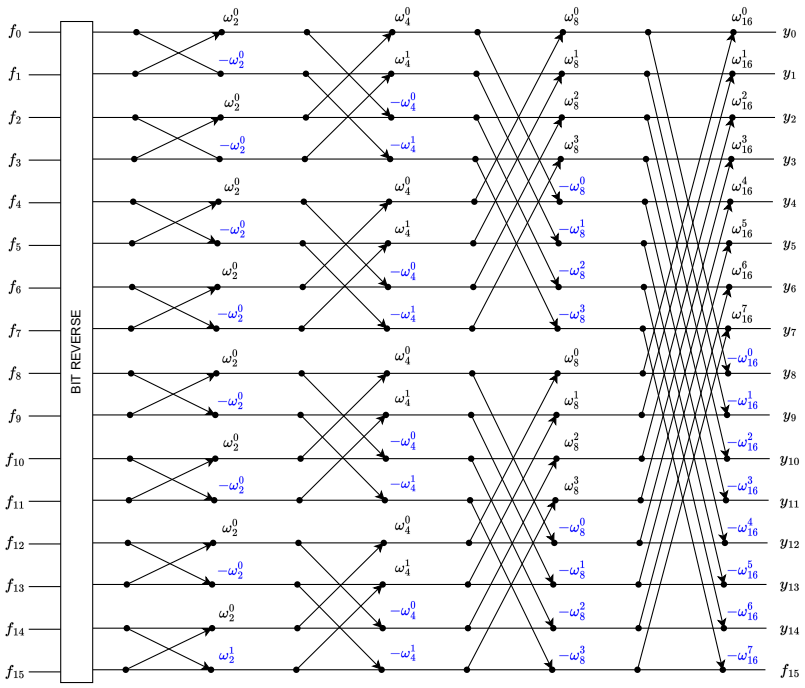

Figure 29: Optimized butterfly diagram for 16 elements' Fast Fourier Transform.

Hence, the previous formula for computing FFTs become the ones below, for each $j \in \{0, \ldots, n/2-1\}$:

$$f_j^{k+1} = f_j^k + f_{j+2^k}^k \cdot \omega_{2^{k+1}}^j,$$
$$f_{j+2^k}^{k+1} = f_j^k - f_{j+2^k}^k \cdot \omega_{2^{k+1}}^j$$

In the picture above it is important to observe some pattern. The components y_0, y_4, y_8 and y_{12} of the output of the whole FFT of 16 elements actually depend on the same intermediate 4 coefficients $f_0^2, f_4^2, f_8^2, f_{12}^2$ appearing in the step 2 of the FFT. In the picture below we show in red the wires that are connected to the components y_0, y_4, y_8 and y_{12} and we can easily see that the previous statement is true. But this is not the only coincidence.

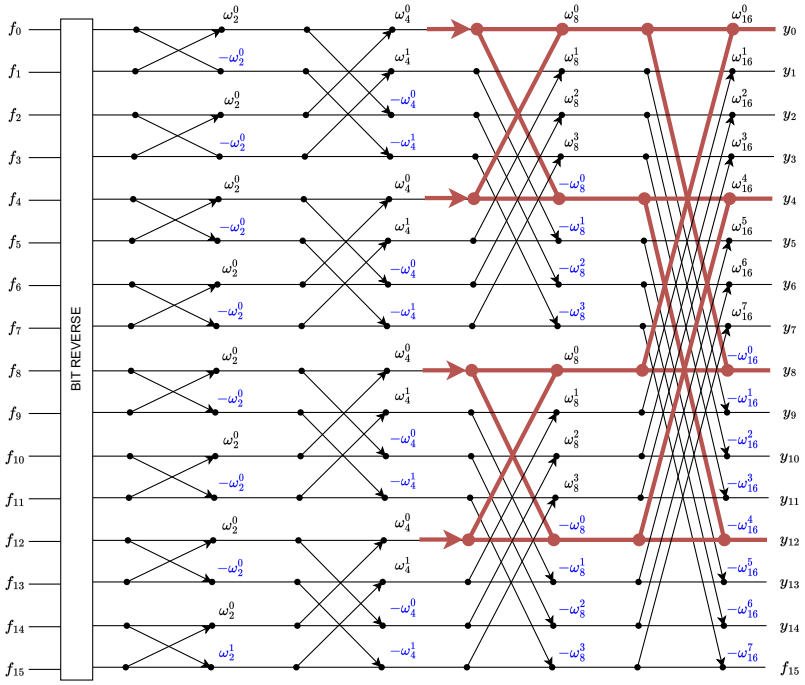

Figure 30: Optimized butterfly diagram for 16 elements' Fast Fourier Transform.

Observe that, in fact, the subgraph formed by the red wires is actually another FFT of 4 elements, with the output being y_0, y_4, y_8 and y_{12}. Therefore, y_0, y_4, y_8 and y_{12} can be expressed as a linear combination of f_0, f_4, f_8 and f_{12} having coefficients some convenient roots of unity. However, observe that, in this step, the used roots have to be modified accordingly. We depict the extracted subgraph in the figure below, in order to be able to see that in a clear way.

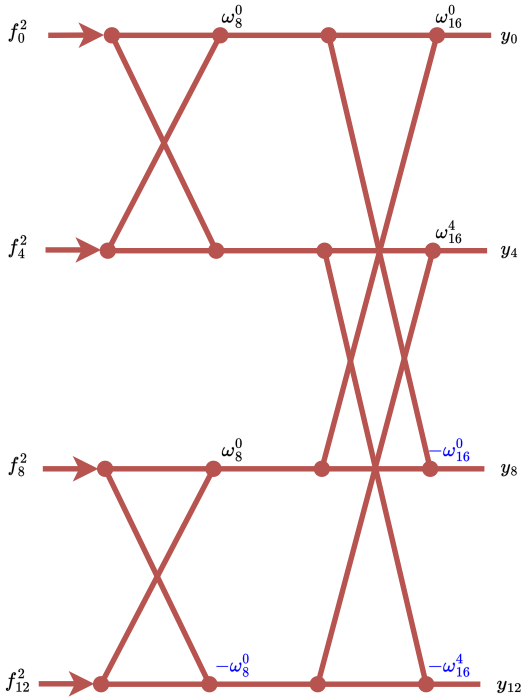

Figure 31: Subgraph representing a 4-sized FFT chaining in the second step.

In this case, the formulas to obtain y_0, y_4, y_8 and y_{12} from f_0^2, f_4^2, f_8^2 and f_{12}^2 are the following:

$$\begin{aligned}
y_0 &= f_0^3 + f_8^3 \cdot \omega_{16}^0 = (f_0^2 + f_4^2 \cdot \omega_8^0) + (f_8^2 + f_{12}^2 \cdot \omega_8^0) \cdot \omega_{16}^0 \\
&= f_0^2 + f_4^2 \cdot \omega_8^0 + f_8^2 \cdot \omega_{16}^0 + f_{12}^2 \cdot \omega_8^0 \cdot \omega_{16}^0 \\
y_4 &= f_4^3 + f_{12}^3 \cdot \omega_{16}^4 = (f_0^2 - f_4^2 \cdot \omega_8^0) + (f_8^2 - f_{12}^2 \cdot \omega_8^0) \cdot \omega_{16}^4 \\
&= f_0^2 - f_4^2 \cdot \omega_8^0 + f_8^2 \cdot \omega_{16}^4 - f_{12}^2 \cdot \omega_8^0 \cdot \omega_{16}^4 \\
y_8 &= f_0^3 - f_8^3 \cdot \omega_{16}^0 = (f_0^2 + f_4^2 \cdot \omega_8^0) - (f_8^2 + f_{12}^2 \cdot \omega_8^0) \cdot \omega_{16}^0 \\
&= f_0^2 + f_4^2 \cdot \omega_8^0 - f_8^2 \cdot \omega_{16}^0 - f_{12}^2 \cdot \omega_8^0 \cdot \omega_{16}^0 \\
y_{12} &= f_4^3 - f_{12}^3 \cdot \omega_{16}^4 = (f_0^2 - f_4^2 \cdot \omega_8^0) - (f_8^2 - f_{12}^2 \cdot \omega_8^0) \cdot \omega_{16}^4 \\
&= f_0^2 - f_4^2 \cdot \omega_8^0 - f_8^2 \cdot \omega_{16}^4 + f_{12}^2 \cdot \omega_8^0 \cdot \omega_{16}^4
\end{aligned}$$

To provide a more specific explaination, let's focus on determining the elements f_j^k that belong to the same 4-sized FFT at the step k, where k is an even integer and n is a power of 4 (so that no 2-sized FFT are needed at the end). Mathematically, we aim to compute a set of representatives for the classes of equivalence $\{f_j^k\}_{j=0}^{n-1} /_{\sim^k}$, where $\sim^k$ denotes the equivalence relation $f_j^k \sim^k f_{j'}^k$ if they belong to the same 4-sized FFT. Notice that if f_j^{k+2} belongs to a equivalence class $f_j^k \in \{f_j^k\}_{j=0}^{n-1} /_{\sim^k}$, where j is the smallest index among all the elements in the class, then all the elements of its class can be expressed as:

$$\left(f_j^k, f_{j+2^k}^k, f_{j+2\cdot 2^k}^k, f_{j+3\cdot 2^k}^k\right)$$

The equivalence relation in the set $\{f_0^k, \ldots, f_{n-1}^k\}$ naturally induces an equivalence relation in the set of indices $\{0, \ldots, n-1\}$. Therefore, we can interchangeably consider both perspectives. The first representative of each class is given by $f_{S_j^k}^k$, where S_j^k is given

by the sequence $\left(S_j^k\right)_j$ for $j \in \{0, \ldots, n/4 - 1\}$:

$$\begin{aligned} S_j^k &= S_{j-1}^k + 1 && \text{if } S_{j-1}^k + 1 \not\equiv 0 \pmod{2^k} \\ S_j^k &= S_{j-1}^k + 3 \cdot 2^k + 1 && \text{if } S_{j-1}^k + 1 \equiv 0 \pmod{2^k} \end{aligned}$$

Let us examine why the previous formula works more carefully. The following example provides a list of the corresponding indices for all the elements in each equivalence class, arranged in the natural order and computed by examining the butterfly diagram. This example considers the case of $n = 64$ and $k = 2$:

$$\begin{aligned} &(0, 4, 8, 12), (1, 5, 9, 13), (2, 6, 10, 14), (3, 7, 1, 15), \\ &(16, 20, 21, 28), (17, 21, 25, 29), (18, 22, 26, 30), (19, 2, 27, 31), \\ &(32, 36, 40, 44), (33, 37, 41, 45), (34, 38, 42, 46), (35, 39, 43, 47), \\ &(48, 52, 56, 60), (49, 53, 57, 61), (50, 54, 58, 62), (51, 55, 59, 63) \end{aligned}$$

being each of the first representatives S_j^k the following sequence of indexes

$$0, 1, 2, 3, 16, 17, 18, 19, 32, 33, 34, 35, 48, 49, 50 \text{ and } 51.$$

We notice that the sequence starts with $0, 1, 2, 3$, but there is an abrupt jump to 16. Tis jump occurs because the index 4 is already present in the class of 0. Hence, we need to jump to the next free slot. We can compute this free slot from the last computed index, in this case 15 (which is computed from S_3^2 as $S_3^2 + 3 \cdot 2^2$), and add one:

$$S_4^2 = S_3^2 + 3 \cdot 2^2 + 1.$$

To correctly determine all the values of S_j^k, it is essential to identify when these jumps occur. To do so, observe that we only need to check whether the consecutive index $S_{j-1}^k + 1$ is a multiple of 2^k or not. If $S_{j-1}^k + 1$ is a multiple, it indicates that $S_{j-1}^k + 1$ is already present in the computed classes. In such cases, we need to compute the next index S_j^k by adding 1 to the last element of the current class, which is $S_{j-1}^k + 3 \cdot 2^k$. If not, we only need to add one to the current index, since the next one is not present yet.

For instance, in the given example, we can see that 15 satisfies the relation

$$2^k = 2^2 = 4 \mid 4 = 3 + 1 = S_3 + 1$$

while

$$\begin{aligned} 4 \nmid 1 &= 0 + 1 = S_0 + 1 \\ 4 \nmid 2 &= 1 + 1 = S_1 + 1 \\ 4 \nmid 3 &= 2 + 1 = S_2 + 1. \end{aligned}$$

Once we have defined the set of classes for each step k of the FFT, we can compute the next state corresponding to step $k + 2$ by independently computing the 4-sized FFT of each class. It's important to note that, since we are in the step k of the n-sized FFT, we will need the values of ω_{2^k} and $\omega_{2^{k+1}}$. However, recall that we have the following fact:

$$\omega_{2^{k+1}}^2 = \omega_{2^k}.$$

For $k \in \{0, 2, \ldots, \log_2(n) - 2\}$, we can establish the following relationships:

$$
\begin{aligned}
f^{k+2}_{S^k_j} &= f^k_{S^k_j} + \omega^{\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2\cdot 2^k} + \omega^{2\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2^k} + \omega^{3\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+3\cdot 2^k} \\
f^{k+2}_{S^k_j+2^k} &= f^k_{S^k_j} + \omega_2 \cdot \omega^{\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2\cdot 2^k} - \omega^{2\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2^k} - \omega_2 \cdot \omega^{3\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+3\cdot 2^k} \\
f^{k+2}_{S^k_j+2\cdot 2^k} &= f^k_{S^k_j} - \omega^{\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2\cdot 2^k} + \omega^{2\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2^k} - \omega^{3\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+3\cdot 2^k} \\
f^{k+2}_{S^k_j+3\cdot 2^k} &= f^k_{S^k_j} - \omega_2 \cdot \omega^{\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2\cdot 2^k} - \omega^{2\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+2^k} + \omega_2 \cdot \omega^{3\mathfrak{s}^k_j}_{2^{k+2}} \cdot f^k_{S^k_j+3\cdot 2^k}
\end{aligned}
$$

where

$$\mathfrak{s}^k_j = S^k_j \pmod{2^k}.$$

The aforementioned equalities are derived from the fact that

$$\omega^j_{2^{k+1}} \cdot \omega_2 = \omega^j_{2^{k+1}} \cdot \omega^{2k}_{2^{k+1}} = \omega^{j+2^k}_{2^{k+1}}$$

which represents the two powers of $\omega_{2^{k+1}}$ associated with the same 4-sized FFT class.

Now, let us move to the case where n is an odd power of two (meaning that $\log_2(n)$ is odd). In the previous case, where $\log_2(n)$ is even, we only need to employ 4-sized FFT chains. However, in this case, we must incorporate a final 2-sized FFT step at the end of the whole process. At the last step, we will need to execute $n/2$ 2-sized FFTs, connecting f^{r-1}_j and $f^{r-1}_{j+n/2}$ for $j \in \{0, \ldots, \frac{n}{2} - 1\}$. Thus, the formulas for the final step, applicable to $j \in \{0, \ldots, \frac{n}{2} - 1\}$, are as follows:

$$
\begin{aligned}
f^r_j &= f^{r-1}_j + f^{r-1}_{j+n/2} \cdot \omega^j_n \\
f^r_{j+n/2} &= f^{r-1}_{j+n/2} - f^{r-1}_j \cdot \omega^j_n
\end{aligned}
$$

Here, we have utilized the fact that $\omega^j_n = -\omega^{j+n/2}_n$.

6.5.2 Fast Fourier Transform in Extension Fields

In our case, it is important to emphasize that we are working with a cubic extension of the field $\mathbb{F}_p$. Therefore, in this section, we will explore the process of reducing the Fast Fourier Transform in an extension field to a number of FFTs equal to its degree, which is three in our case.

Supose that we want to compute the FFT of a polynomial with coeficients in a extension finite field $\mathbb{F}_{p^r}$. Let $f(x) \in \mathbb{F}_p$ be an irreducible polynomial with $\deg(f(x)) = r$. We know that $\mathbb{F}_{p^r} \cong \mathbb{F}_p[x]/(f(x))$. Let $\{1, \ldots, \alpha^{r-1}\}$ be a $\mathbb{F}_p$-basis, we can write any element $\beta \in \mathbb{F}_{p^r}$ as $\beta = \sum_{i=1}^{r-1} a_i \cdot \alpha^i$.

Note that we can write any polynomial $p(x) \in \mathbb{F}_{p^r}[x]$ as

$$p(x) = \sum_{j=1}^{n} \beta_j x^j = \sum_{j=1}^{n} \left(\sum_{i=1}^{r-1} a^i_j \alpha^i \right) x^j = \sum_{i=1}^{r-1} \alpha^i \left(\sum_{j=1}^{n} a^i_j x^n \right).$$

Hence, in order to compute the FFT of $p(x)$, we only have to compute FFTs of the polynomials

$$p_i(x) = \sum_{j=1}^{n} a^i_j x^n$$

for $i \in \{1, \ldots, r-1\}$ (observe that $p_i(x) \in \mathbb{F}_p[x]$).

6.5.3 PIL Verification of the FFT

Once we have described the chaining process for FFTs, let us describe how to verify its usage in PIL. We are given 12 witness values, denoted as $\mathtt{a}[0], \ldots, \mathtt{a}[11]$. These values are used to form four extended field elements, namely $f_0, f_1, f_2, f_3 \in \mathbb{F}_{p^3}$ constructed as follows:

$$\begin{aligned} f_0 &= \mathtt{a}[0] + \mathtt{a}[1] \cdot X + \mathtt{a}[2] \cdot X^2 \\ f_1 &= \mathtt{a}[3] + \mathtt{a}[4] \cdot X + \mathtt{a}[5] \cdot X^2 \\ f_2 &= \mathtt{a}[6] + \mathtt{a}[7] \cdot X + \mathtt{a}[8] \cdot X^2 \\ f_3 &= \mathtt{a}[9] + \mathtt{a}[10] \cdot X + \mathtt{a}[11] \cdot X^2 \end{aligned}$$

Our goal is to verify whether the next row of witness values, denoted as $\mathtt{a}[0]', \ldots, \mathtt{a}[11]'$ mapped to the following extended field elements

$$\begin{aligned} f_0' &= \mathtt{a}[0]' + \mathtt{a}[1]' \cdot X + \mathtt{a}[2]' \cdot X^2 \\ f_1' &= \mathtt{a}[3]' + \mathtt{a}[4]' \cdot X + \mathtt{a}[5]' \cdot X^2 \\ f_2' &= \mathtt{a}[6]' + \mathtt{a}[7]' \cdot X + \mathtt{a}[8]' \cdot X^2 \\ f_3' &= \mathtt{a}[9]' + \mathtt{a}[10]' \cdot X + \mathtt{a}[11]' \cdot X^2 \end{aligned}$$

satisfies the following relationship

$$(f_0', f_1', f_2', f_3') = \mathtt{FFT4}(f_0, f_1, f_2, f_3)$$

where $\mathtt{FFT4}(f_0, f_1, f_2, f_3)$ denotes the result of the 4-sized Fast Fourier Transform of f_0, f_1, f_2 and f_3. Additionally, since we are chaining FFTs as explained earlier, we need to keep track of the current step of the n-sized FFT and its corresponding constants.

The polynomial constraints that need to be verified are of the following form:

$$\mathtt{FFT4} \cdot (\mathtt{a}[i]' - g_i) = 0$$

Here, the selector $\mathtt{FFT4}$ indicates that an FFT check is being performed, and g_i represents a linear combination of the previous row of witness values, with coefficients determined by the current round. The coefficients correspond to the correct computation of the FFT.

To compute g_i, we set up constants denoted as $\mathtt{C}[k]$ for different cases of FFT steps. Let's consider the cases where we perform a 4-sized FFT and a 2-sized FFT.

For a 4**-sized FFT step**, the constants are:

$$\begin{aligned} \mathtt{C}[0] &= 1 \\ \mathtt{C}[1] &= \omega_{2^{k+2}}^{2\mathfrak{s}_j^k} \\ \mathtt{C}[2] &= \omega_{2^{k+2}}^{\mathfrak{s}_j^k} \\ \mathtt{C}[3] &= \omega_{2^{k+2}}^{3\mathfrak{s}_j^k} \\ \mathtt{C}[4] &= \omega_2 \cdot \omega_{2^{k+2}}^{\mathfrak{s}_j^k} \\ \mathtt{C}[5] &= \omega_2 \cdot \omega_{2^{k+2}}^{3\mathfrak{s}_j^k} \\ \mathtt{C}[6] &= 0 \\ \mathtt{C}[7] &= 0 \\ \mathtt{C}[8] &= 0 \end{aligned}$$

On the other hand, for a 2**-sized FFT step** (that is, we are at the last step of a n-sized FFT with n an odd power of two), we set up the constants as follows:

$$
\begin{aligned}
\mathtt{C}[0] &= 0 \\
\mathtt{C}[1] &= 0 \\
\mathtt{C}[2] &= 0 \\
\mathtt{C}[3] &= 0 \\
\mathtt{C}[4] &= 0 \\
\mathtt{C}[5] &= 0 \\
\mathtt{C}[6] &= 1 \\
\mathtt{C}[7] &= \omega_n^j \\
\mathtt{C}[8] &= \omega_n^{j+1}
\end{aligned}
$$

To optimize the overall procedure, we employ a strategy that involves performing two consecutive 2-sized FFTs at each step. This is achieved by utilizing two consecutive roots of unity within the constant values. This optimization approach proves to be optimal because the number of necessary 2-sized FFTs doubles the amount of necessary 4-sized FFTs, resulting in an efficient computation process.

Furthermore, we can introduce a scale factor of $1/n$ for each of the constants and change the corresponding root of unity to its inverse if we need to perform the inverse FFT. For a 4**-sized FFT step**, the constants become:

$$
\begin{aligned}
\mathtt{C}[0] &= 1/n \\
\mathtt{C}[1] &= 1/n \cdot \omega_{2^{k+2}}^{-2\mathfrak{s}_j^k} \\
\mathtt{C}[2] &= 1/n \cdot \omega_{2^{k+2}}^{-\mathfrak{s}_j^k} \\
\mathtt{C}[4] &= 1/n \cdot \omega_2 \cdot \omega_{2^{k+2}}^{-\mathfrak{s}_j^k} \\
\mathtt{C}[5] &= 1/n \cdot \omega_2 \cdot \omega_{2^{k+2}}^{-3\mathfrak{s}_j^k} \\
\mathtt{C}[6] &= 0 \\
\mathtt{C}[7] &= 0 \\
\mathtt{C}[8] &= 0
\end{aligned}
$$

On the other hand, for a 2**-sized FFT step**, we set up the constants as follows:

$$
\begin{aligned}
\mathtt{C}[0] &= 0 \\
\mathtt{C}[1] &= 0 \\
\mathtt{C}[2] &= 0 \\
\mathtt{C}[3] &= 0 \\
\mathtt{C}[4] &= 0 \\
\mathtt{C}[5] &= 0 \\
\mathtt{C}[6] &= 1/n \\
\mathtt{C}[7] &= 1/n \cdot \omega_n^{-j} \\
\mathtt{C}[8] &= 1/n \cdot \omega_n^{-j-1}
\end{aligned}
$$

Using these constants, we can compute the values of g_i as follows:

```
pol g0  = C[0]*a[0] + C[1]*a[3] + C[2]*a[6] + C[3]*a[9]  + C[6]*a[0] + C
    [7]*a[3];
pol g3  = C[0]*a[0] - C[1]*a[3] + C[4]*a[6] - C[5]*a[9]  + C[6]*a[0] - C
    [7]*a[3];
pol g6  = C[0]*a[0] + C[1]*a[3] - C[2]*a[6] - C[3]*a[9]  + C[6]*a[6] + C
    [8]*a[9];
pol g9  = C[0]*a[0] - C[1]*a[3] - C[4]*a[6] + C[5]*a[9]  + C[6]*a[6] - C
    [8]*a[9];

pol g1  = C[0]*a[1] + C[1]*a[4] + C[2]*a[7] + C[3]*a[10] + C[6]*a[1] + C
    [7]*a[4];
pol g4  = C[0]*a[1] - C[1]*a[4] + C[4]*a[7] - C[5]*a[10] + C[6]*a[1] - C
    [7]*a[4];
pol g7  = C[0]*a[1] + C[1]*a[4] - C[2]*a[7] - C[3]*a[10] + C[6]*a[7] + C
    [8]*a[10];
pol g10 = C[0]*a[1] - C[1]*a[4] - C[4]*a[7] + C[5]*a[10] + C[6]*a[7] - C
    [8]*a[10];

pol g2  = C[0]*a[2] + C[1]*a[5] + C[2]*a[8] + C[3]*a[11] + C[6]*a[2] + C
    [7]*a[5];
pol g5  = C[0]*a[2] - C[1]*a[5] + C[4]*a[8] - C[5]*a[11] + C[6]*a[2] - C
    [7]*a[5];
pol g8  = C[0]*a[2] + C[1]*a[5] - C[2]*a[8] - C[3]*a[11] + C[6]*a[8] + C
    [8]*a[11];
pol g11 = C[0]*a[2] - C[1]*a[5] - C[4]*a[8] + C[5]*a[11] + C[6]*a[8] - C
    [8]*a[11];
```

By checking these polynomial constraints, we can ensure the accuracy of the n-sized FFT computation for different scenarios, including both forward and inverse FFT, depending on the constants used.

6.6 Polynomial Evaluation Verification

Whenever `EVPOL4` is 1, the PIL file will check that the value

$$\mathtt{a}[6]' + \mathtt{a}[7]' \cdot X + \mathtt{a}[8]' \cdot X^2 \in \mathbb{F}_{p^3}$$

is obtained by evaluating the polynomial

$$p(Z) = \mathtt{d0} \cdot Z^4 + \mathtt{d1} \cdot Z^3 + \mathtt{d2} \cdot Z^2 + \mathtt{d3} \cdot Z + \mathtt{d4}$$

at a given point

$$z = \mathtt{a}[3]' + \mathtt{a}[4]' \cdot X + \mathtt{a}[5]' \cdot X^2$$

being `d0`, `d1`, `d2`, `d3` and `d4` $\in \mathbb{F}_{p^3}$ defined as:

$$\begin{aligned}
\mathtt{d0} &= \mathtt{a}[0]' + \mathtt{a}[1]' \cdot X + \mathtt{a}[2]' \cdot X^2 \\
\mathtt{d1} &= \mathtt{a}[9] + \mathtt{a}[10] \cdot X + \mathtt{a}[11] \cdot X^2 \\
\mathtt{d2} &= \mathtt{a}[6] + \mathtt{a}[7] \cdot X + \mathtt{a}[8] \cdot X^2 \\
\mathtt{d3} &= \mathtt{a}[3] + \mathtt{a}[4] \cdot X + \mathtt{a}[5] \cdot X^2 \\
\mathtt{d4} &= \mathtt{a}[0] + \mathtt{a}[1] \cdot X + \mathtt{a}[2] \cdot X^2
\end{aligned}$$

This evaluation will be done using Horner's rule, so that $p(Z)$ will be rewritten as

$$p(Z) = (((\mathtt{d0} \cdot Z + \mathtt{d1}) \cdot Z + \mathtt{d2}) \cdot Z + \mathtt{d3}) \cdot Z + \mathtt{d4}.$$

Also observe that all the operations performed at the evaluation are operations in $\mathbb{F}_{p^3}$, so we will perform the operations in PIL using the following `CMulAdd` function, written in `ejs`. Observe that the logic is exactly the same than the one presented in Section 6.4.

Note that the order selected to input the parameters of the `EVPOL4` gate is totally arbitrary and we should only have to take care that is **completely** aligned with the Circom's custom gate. Moreover, observe that, since there are more than 12 parameters, they all not fit in a row of the execution trace, forcing us to use 2 of them to include them. This is not even a problem since the validation only occupies 3 columns.

```
<% function CMulAdd(s, a0, a1, a2, b0, b1, b2, c0, c1, c2) {
    const code = [];
    code.push(`pol A${s} = (${a0} + ${a1})  * (${b0} + ${b1});`);
    code.push(`pol B${s} = (${a0} + ${a2})  * (${b0} + ${b2});`);
    code.push(`pol C${s} = (${a1} + ${a2})  * (${b1} + ${b2});`);
    code.push(`pol D${s} = ${a0} * ${b0};`);
    code.push(`pol E${s} = ${a1} * ${b1};`);
    code.push(`pol F${s} = ${a2} * ${b2};`);
    code.push(`pol acc${s}_0 = C${s}+ D${s} - E${s} - F${s} + ${c0};`);
    code.push(`pol acc${s}_1 = A${s}+ C${s}- 2*E${s} - D${s} + ${c1};`);
    code.push(`pol acc${s}_2 = B${s}- D${s} + E${s} + ${c2};`);
    code.push(`\n`);
    return code.join("\n");
} -%>
```

Now, we can compute $p(z) \in \mathbb{F}_{p^3}$ using the previous function, accumulating the results by using the Horner's rule:

```
// acc = d0 * x + d1
<%- CMulAdd(
        "1",
        "a[0]'",
        "a[1]'",
        "a[2]'",
        "a[3]'",
        "a[4]'",
        "a[5]'",
        "a[9]",
        "a[10]",
        "a[11]"
    )
-%>
```

```
// acc2 = acc * x + d2
<%- CMulAdd(
        "2",
        "acc1_0",
        "acc1_1",
        "acc1_2",
        "a[3]'",
        "a[4]'",
        "a[5]'",
        "a[6]'",
        "a[7]",
        "a[8]"
    )
-%>
```

```
// acc3 = acc2 * x + d3
<%- CMulAdd(
        "3",
        "acc2_0",
        "acc2_1",
        "acc2_2",
        "a[3]'",
        "a[4]'",
        "a[5]'",
        "a[3]",
        "a[4]",
        "a[5]"
    )
-%>
```

```
// acc4 = acc3 * x + d4
<%- CMulAdd(
        "4",
        "acc3_0",
        "acc3_1",
        "acc3_2",
        "a[3]'",
        "a[4]'",
        "a[5]'",
        "a[0]",
        "a[1]",
        "a[2]"
    )
-%>
```

Now, the only what remains is to check that the last values obtained `acc3_0`, `acc3_1`, `acc3_2` are equal to the supposed committed evaluation of p at z, $\mathtt{a}[6]'$, $\mathtt{a}[7]'$ and $\mathtt{a}[8]'$ respectively:

```
EVPOL4 * (a[6]' - acc4_0) = 0;
EVPOL4 * (a[7]' - acc4_1) = 0;
EVPOL4 * (a[8]' - acc4_2) = 0;
```

7 Appendix: Proof Composition with SNARKjs and PIL-STARK

In this section, we are going to show the current proving capabilities of the `SNARKjs` and `PIL-STARK` tool stack. In particular, we are going to see how some cases of proof composition can be achieved, extending the power of both `SNARKjs` and `PIL-STARK` beyond what they were originally defined for.

In every case throughout this section we are going to be proving the following statement:

"I known a (secret) value $a_1 = 1$ such that the $(2^5 + 1 =)$ 33-th element of the Fibonacci sequence on (public) input $a_0 = 1$ and a_1 is equal to 3524578."

In the first two sections, we recall the workflow that needs to be followed to generate a (depth 0) proof.

7.1 Depth 0: Circom + SNARKjs

We represent the statement in Circom, which means that we must represent the computation of the Fibonacci sequence in R1CS format:

```
pragma circom 2.1.5;

template Fibonacci(n) {
    signal input a0;
    signal input a1;
    signal output out;

    signal im[n-1];

    for (var i=0; i<n-1; i++) {
        if (i==0) {
            im[i] <== a0 + a1;
        } else if (i==1) {
            im[i] <== a1 + im[0];
        } else {
            im[i] <== im[i-2] + im
                [i-1];
        }
    }

    out <== im[n-2];
    log("out =", out);
}

component main {public [a0]} =
    Fibonacci(2**5);
```

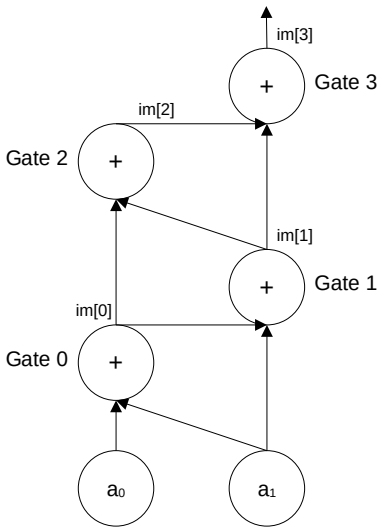

Figure 32: Circom code for the Fibonacci sequence and the equivalent representation as an arithmetic circuit.

The execution trace for the particular instance of this circuit (i.e., $a_0 = 1$ and $a_1 = 1$)

is as follows:

Execution Trace			
Gate 0	1	1	2
Gate 1	1	2	3
Gate 2	2	3	5
⋮	⋮	⋮	⋮
Gate 32	1346269	2178309	3524578

where the numbers colored in red are the information known by the prover, while the numbers colored in green are the information known to both the prover and the verifier.

After the description of the circuit in Circom we can proceed to generate a valid zk-SNARK[2] for this particular instance using `SNARKjs`:

1. Compile the circuit:

```
$ mkdir -p build && circom src/fibonacci.circom --r1cs --wasm -o build

template instances: 1
non-linear constraints: 0
linear constraints: 0
public inputs: 1
public outputs: 1
private inputs: 1
private outputs: 0
wires: 3
labels: 35
```

2. Create a `input.json` file with the inputs for our circuit in the same directory:

```
$ cat <<EOT > input.json
  {"a0": 1, "a1": 1}
  EOT
```

3. Compute the witness for our inputs:

```
$ snarkjs wc build/fibonacci_js/fibonacci.wasm src/input.json build/
    fibonacci.wtns

out = 3524578
```

4. Download a sufficiently large "powers of tau" file:

```
$ wget https://hermez.s3-eu-west-1.amazonaws.com/
    powersOfTau28_hez_final_10.ptau -O build/powersOfTau.ptau
```

5. Preprocess the circuit:

```
$ snarkjs pks build/fibonacci.r1cs build/powersOfTau.ptau build/
    fibonacci.zkey

[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Plonk constraints: 2
[INFO]  snarkJS: Setup Finished
```

6. Export the verification key for proof verification:

[2]We are going to use PlonK for simplicity, but `SNARKjs` also supports Groth16 and Fflonk.

```
$ snarkjs zkev build/fibonacci.zkey build/fibonacci-vk.json

[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: plonk
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
```

7. Generate the PlonK proof:

```
$ snarkjs pkp build/fibonacci.zkey build/fibonacci.wtns build/fibonacci
    .proof.json build/fibonacci.public.json
```

8. Finally, verify the proof:

```
$ snarkjs pkv build/fibonacci-vk.json build/fibonacci.public.json build
    /fibonacci.proof.json

[INFO]  snarkJS: OK!
```

7.2 Depth 0: PIL + PIL-STARK

Now, we represent the statement in PIL. This means that we must represent the computation of the Fibonacci sequence in AIR format, i.e., as a set of polynomial constraints over a domain $G = \langle g \rangle$. In this case, if the following constraints are satisfied for all $x \in G$:

$$\begin{aligned}(1 - L_{32}(x)) \cdot (a_0(x \cdot \omega_{32}) - a_1(x)) &= 0,\\ (1 - L_{32}(x)) \cdot (a_1(x \cdot \omega_{32}) - (a_0(x) + a_1(x))) &= 0,\\ L_1(x) \cdot (a_0(x) - 1) &= 0,\\ L_{32}(x) \cdot (a_1(x) - 3524578) &= 0 = 0.\end{aligned}$$

then the original statement is true. Here, L_1, L_{32} are the first and last Lagrange polynomials over G, respectively; and a_0, a_1 are defined to hold the values of the Fibonacci sequence in each state transition (see Figure 33).

```
constant %N = 2**5;

namespace Fibonacci(%N);
    pol constant L1;    //
        1,0,0,.....,0
    pol constant LN;    //
        0,0,.....,0,1
    pol commit a0, a1;

    (1-LN) * (a0' - a1) = 0;
    (1-LN) * (a1' - (a0 + a1)) =
        0;

    public in0 = a0(0);
    public out = a1(%N-1);
    L1 * (a0 - :in0) = 0;
    LN * (a1 - :out) = 0;
```

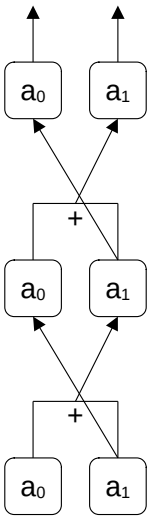

Figure 33: PIL code for the Fibonacci sequence and the equivalent representation as a state machine with registers a_0 and a_1.

The execution trace for the particular instance of this state machine (i.e., $a_0(g) = 1$ and $a_1(g) = 1$) is as follows:

Execution Trace				
Registers	L_1	L_{32}	a_0	a_1
State 1	1	0	1	1
State 2	0	0	1	2
State 3	0	0	2	3
⋮	⋮	⋮	⋮	⋮
State 31	0	1	1346269	2178309
State 32	0	1	2178309	3524578

After the description of the circuit in PIL we can proceed to generate a valid eSTARK proof for this particular instance using `pil-stark`:

1. Populate the constant polynomials in the execution trace:

```
$ mkdir -p build && node src/main_buildconst.js -o build/fibonacci.
    const

file Generated Correctly
```

2. Populate the committed polynomials in the execution trace:

```
$ node src/main_buildcommit.js -i src/input.json -o build/fibonacci.
    commit

Result: 3524578
file Generated Correctly
```

3. Verify that the execution trace generated in the previous two steps is valid:

```
$ node node_modules/pilcom/src/main_pilverifier.js build/fibonacci.
    commit -p src/fibonacci.pil -c build/fibonacci.const

PIL OK!!
```

4. Create a `starkstruct.json` file with the eSTARK parameters in the same directory:

```
$ cat <<EOT > starkstruct.json
{
    "nBits": 5,
    "nBitsExt": 6,
    "nQueries": 64,
    "verificationHashType": "GL",
    "steps": [
        {"nBits": 6},
        {"nBits": 4},
        {"nBits": 2}
    ]
}
EOT
```

5. Generate the `starkinfo.json` file needed to generate the eSTARK:

```
$ node node_modules/pil-stark/src/main_genstarkinfo.js -p src/fibonacci
    .pil -s src/starkstruct.json -i build/starkinfo.json

files Generated Correctly
```

6. Preprocess the state machine:

```
$ node node_modules/pil-stark/src/main_buildconsttree.js -c build/
    fibonacci.const -p src/fibonacci.pil -s src/starkstruct.json -t
    build/fibonacci.consttree -v build/fibonacci.verkey.json

files Generated Correctly
```

7. Generate the eSTARK proof:

```
$ node node_modules/pil-stark/src/main_prover.js -m build/fibonacci.
    commit -c build/fibonacci.const -t build/fibonacci.consttree -p src
    /fibonacci.pil -s build/starkinfo.json -o build/fibonacci.proof.
    json -z build/fibonacci.zkin.json -b build/fibonacci.public.json

files Generated Correctly
```

8. Finally, verify the proof:

```
$ node node_modules/pil-stark/src/main_verifier.js -p src/fibonacci.pil
     -s build/starkinfo.json -v build/fibonacci.verkey.json -o build/
    fibonacci.proof.json -b build/fibonacci.public.json

Verification Ok!!
```

7.3 Depth 0: Circom + PIL-STARK

In this section, we show how to generate an eSTARK proof over the Fibonacci circuit written in Circom directly.

1. Compile the circuit:

```
$ mkdir -p build && circom src/fibonacci.circom --O1 --prime goldilocks
     --r1cs --wasm -o build

template instances: 1
non-linear constraints: 0
linear constraints: 31
public inputs: 1
public outputs: 1
private inputs: 1
private outputs: 0
wires: 34
labels: 35
```

   Notice how in this case the goldilocks prime field was chosen instead of the prime field over which is defined the BN128 elliptic curve (the default field in Circom). This is because the conversion of a circuit written in Circom to a state machine written in PIL is only defined, for the moment, over the goldilocks field.

2. Create a `input.json` file with the inputs for our circuit in the same directory:

```
$ cat <<EOT > input.json
  {"a0": 1, "a1": 1}
  EOT
```

3. Compute the witness for our inputs:

```
$ snarkjs wc build/fibonacci_js/fibonacci.wasm src/input.json build/
    fibonacci.wtns

out = 3524578
```

4. Now that we have computed the witness (or more precise, the R1CS) we convert the R1CS representation of the circuit to an equivalent PlonK representation of the same circuit:

```
$ node node_modules/pil-stark/src/main_plonksetup.js -r build/fibonacci
    .r1cs -p build/fibonacci.pil -e build/fibonacci.exec -c build/
    fibonacci.const

files Generated Correctly
```

   The output state machine written in PIL is the following:

```
constant %N = 2**6;

namespace Global(%N);
    pol constant L1;
    pol constant L2;

namespace PlonkCircuit(%N);
    pol constant S[3];
    pol constant Qm, Ql, Qr, Qo, Qk;
    pol commit a[3];

    public pub0 = a[0](0);
    public pub1 = a[0](1);
    Global.L1 * (a[0] - :pub0) = 0;
    Global.L2 * (a[0] - :pub1) = 0;

// Normal plonk ecuations
    pol a01 = a[0]*a[1];
    Qm*a01 + Ql*a[0] + Qr*a[1] + Qo*a[2] + Qk = 0;

// Connection equations
    {a[0], a[1], a[2]} connect {S[0], S[1], S[2]};
```

   This process also outputs a file containing the constant polynomials generated during the conversion.

5. Populate the committed polynomials in the execution trace:

```
$ node node_modules/pil-stark/src/main_plonkexec.js -w build/fibonacci.
    wtns -p build/fibonacci.pil -e build/fibonacci.exec -m build/
    fibonacci.commit

file Generated Correctly
```

6. Verify that the execution trace generated in the previous two steps is valid:

```
$ node node_modules/pilcom/src/main_pilverifier.js build/fibonacci.
    commit -p build/fibonacci.pil -c build/fibonacci.const

PIL OK!!
```

7. Create a `starkstruct.json` file with the eSTARK parameters in the same directory:

```
$ cat <<EOT > starkstruct.json
{
    "nBits": 6,
    "nBitsExt": 7,
    "nQueries": 64,
    "verificationHashType": "BN128",
    "steps": [
        {"nBits": 7},
        {"nBits": 5},
        {"nBits": 3}
    ]
}
EOT
```

Importantly, the number of bits needs to be increased from 5 to 6 due to the overhead of the R1CS to PlonK transformation (see the output state machine).

Now, we can proceed as in the previous example to generate the eSTARK.

8. Generate the `starkinfo.json` file needed to generate the eSTARK:

```
$ node node_modules/pil-stark/src/main_genstarkinfo.js -p build/
    fibonacci.pil -s src/starkstruct.json -i build/starkinfo.json

files Generated Correctly
```

9. Preprocess the state machine:

```
$ node node_modules/pil-stark/src/main_buildconsttree.js -c build/
    fibonacci.const -p build/fibonacci.pil -s src/starkstruct.json -t
    build/fibonacci.consttree -v build/fibonacci.verkey.json

files Generated Correctly
```

10. Generate the eSTARK proof:

```
$ node node_modules/pil-stark/src/main_prover.js -m build/fibonacci.
    commit -c build/fibonacci.const -t build/fibonacci.consttree -p
    build/fibonacci.pil -s build/starkinfo.json -o build/fibonacci.
    proof.json -z build/fibonacci.zkin.json -b build/fibonacci.public.
    json

files Generated Correctly
```

11. Finally, verify the proof:

```
$ node node_modules/pil-stark/src/main_verifier.js -p build/fibonacci.
    pil -s build/starkinfo.json -v build/fibonacci.verkey.json -o build
    /fibonacci.proof.json -b build/fibonacci.public.json

Verification Ok!!
```

In the following sections, we expand beyond the previous two workflows by a combination of either `PIL-STARK` with itself or `PIL-STARK` with `SNARKjs`.

7.4 Depth 1: Circom + PIL-STARK + SNARKjs

We start by showing how to generate a SNARK proof that an eSTARK proof is valid over the statement represented in Circom:

1. Compile the circuit:

```
$ mkdir -p build && circom src/fibonacci.circom --O1 --prime goldilocks
     --r1cs --wasm -o build

Everything went okay, circom safe
```

2. Create a `input.json` file with the inputs for our circuit in the same directory:

```
$ cat <<EOT > input.json
    {"a0": 1, "a1": 1}
    EOT
```

3. Compute the witness for our inputs:

```
$ snarkjs wc build/fibonacci_js/fibonacci.wasm src/input.json build/
    fibonacci.wtns

out = 3524578
```

4. Now that we have computed the witness (or more precise, the R1CS) we convert the R1CS representation of the circuit to an equivalent PlonK representation of the same circuit:

```
$ node node_modules/pil-stark/src/main_plonksetup.js -r build/fibonacci
    .r1cs -p build/fibonacci.pil -e build/fibonacci.exec -c build/
    fibonacci.const

files Generated Correctly
```

5. Populate the committed polynomials in the execution trace:

```
$ node node_modules/pil-stark/src/main_plonkexec.js -w build/fibonacci.
    wtns -p build/fibonacci.pil -e build/fibonacci.exec -m build/
    fibonacci.commit

file Generated Correctly
```

6. Verify that the execution trace generated in the previous two steps is valid:

```
$ node node_modules/pilcom/src/main_pilverifier.js build/fibonacci.
    commit -p build/fibonacci.pil -c build/fibonacci.const

PIL OK!!
```

7. Create a `starkstruct.json` file with the eSTARK parameters in the same directory:

```
$ cat <<EOT > starkstruct.json
{
    "nBits": 6,
    "nBitsExt": 7,
    "nQueries": 64,
    "verificationHashType": "BN128",
    "steps": [
        {"nBits": 7},
        {"nBits": 5},
        {"nBits": 3}
    ]
}
EOT
```

8. Generate the `starkinfo.json` file needed to generate the eSTARK:

```
$ node node_modules/pil-stark/src/main_genstarkinfo.js -p build/
    fibonacci.pil -s src/starkstruct.json -i build/starkinfo.json

files Generated Correctly
```

9. Preprocess the state machine:

```
$ node node_modules/pil-stark/src/main_buildconsttree.js -c build/
    fibonacci.const -p build/fibonacci.pil -s src/starkstruct.json -t
    build/fibonacci.consttree -v build/fibonacci.verkey.json

files Generated Correctly
```

10. Generate the eSTARK proof:

```
$ node node_modules/pil-stark/src/main_prover.js -m build/fibonacci.
    commit -c build/fibonacci.const -t build/fibonacci.consttree -p
    build/fibonacci.pil -s build/starkinfo.json -o build/fibonacci.
    proof.json -z build/fibonacci.zkin.json -b build/fibonacci.public.
    json --proverAddr 0x7BAbF98C66454aF8a3C366F893f99EBa26a15c66

files Generated Correctly
```

11. We can verity that the proof is correct:

```
$ node node_modules/pil-stark/src/main_verifier.js -p build/fibonacci.
    pil -s build/starkinfo.json -v build/fibonacci.verkey.json -o build
    /fibonacci.proof.json -b build/fibonacci.public.json

Verification Ok!!
```

12. Next, we generate an eSTARK verifier as a circuit written in Circom:

```
$ node node_modules/pil-stark/src/main_pil2circom.js -p build/fibonacci
    .pil -s build/starkinfo.json -v build/fibonacci.verkey.json -o
    build/verifier.circom

file Generated Correctly
```

This will allow the recursivity to happen: if we generate a valid SNARK on input the previously generated eSTARK proof, then we will prove that the eSTARK proof is valid.

13. We compile the circuit using the circuits in `circuits.gl` needed to compile the eSTARK verifier circuit:

```
$ circom build/verifier.circom --r1cs --wasm -l node_modules/pil-stark/
    circuits.bn128 -l node_modules/circomlib/circuits -o build

template instances: 465
non-linear constraints: 2123834
linear constraints: 0
public inputs: 0
public outputs: 1
private inputs: 14682
private outputs: 0
wires: 2125925
labels: 5010997
```

14. Create a `input.json` file with the inputs for our circuit in the same directory:

```
$ cat <<EOT > input.json
  {"a0": 1, "a1": 1}
  EOT
```

15. Compute the witness for our inputs:

```
$ snarkjs wc build/verifier_js/verifier.wasm build/fibonacci.zkin.json
    build/verifier.wtns

out = 3524578
```

16. Download sufficiently large "powers of tau" file:

```
$ wget https://hermez.s3-eu-west-1.amazonaws.com/
    powersOfTau28_hez_final_16.ptau -O build/powersOfTau.ptau
```

17. Preprocess the circuit:

```
$ snarkjs pks build/verifier.r1cs build/powersOfTau.ptau build/
    fibonacci.zkey

[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Plonk constraints: 2
[INFO]  snarkJS: Setup Finished
```

18. Export the verification key for the verification of the proof:

```
$ snarkjs zkev build/veirifer.zkey build/veirifer-vk.json

[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: plonk
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
```

19. Generate the PlonK proof:

```
$ snarkjs pkp build/veirifer.zkey build/veirifer.wtns build/veirifer.
    proof.json build/veirifer.public.json
```

20. Finally, verify the proof:

```
$ snarkjs pkv build/veirifer-vk.json build/veirifer.public.json build/
    veirifer.proof.json

[INFO]  snarkJS: OK!
```

7.5 Unlimited Depth with PIL-STARK

In this section, we show how to generate an eSTARK proof that an eSTARK proof is valid over the Fibonacci sequence written in PIL. However, the same workflow can be repeated over and over to obtain an unlimited depth proof composition. As shown in the previous section, this unlimited eSTARK composition can always be ended up with a final SNARK.

The following workflow exhibits the full power of the `PIL-STARK` and `SNARKjs` tool stack.

1. Populate the constant polynomials in the execution trace:

```
$ mkdir -p build && node src/main_buildconst.js -o build/fibonacci.
    const

file Generated Correctly
```

2. Populate the committed polynomials in the execution trace:

```
$ node src/main_buildcommit.js -i src/input.json -o build/fibonacci.
    commit

Result: 3524578
file Generated Correctly
```

3. Verify that the execution trace generated in the previous two steps is valid:

```
$ node node_modules/pilcom/src/main_pilverifier.js build/fibonacci.
    commit -p src/fibonacci.pil -c build/fibonacci.const

PIL OK!!
```

4. Create a `starkstruct.json` file with the eSTARK parameters in the same directory:

```
$ cat <<EOT > fibonacci.starkstruct.json
{
    "nBits": 5,
    "nBitsExt": 6,
    "nQueries": 64,
    "verificationHashType": "GL",
    "steps": [
        {"nBits": 6},
        {"nBits": 4},
        {"nBits": 2}
    ]
}
EOT
```

5. Generate the `starkinfo.json` file needed to generate the eSTARK:

```
$ node node_modules/pil-stark/src/main_genstarkinfo.js -p src/fibonacci
    .pil -s src/fibonacci.starkstruct.json -i build/starkinfo.json

files Generated Correctly
```

6. Preprocess the state machine:

```
$ node node_modules/pil-stark/src/main_buildconsttree.js -c build/
    fibonacci.const -p src/fibonacci.pil -s src/fibonacci.starkstruct.
    json -t build/fibonacci.consttree -v build/fibonacci.verkey.json

files Generated Correctly
```

7. Generate the eSTARK proof:

```
$ node node_modules/pil-stark/src/main_prover.js -m build/fibonacci.
    commit -c build/fibonacci.const -t build/fibonacci.consttree -p src
    /fibonacci.pil -s build/starkinfo.json -o build/fibonacci.proof.
    json -z build/fibonacci.zkin.json -b build/fibonacci.public.json

files Generated Correctly
```

8. We can verify that the proof is correct:

```
$ node node_modules/pil-stark/src/main_verifier.js -p src/fibonacci.pil
     -s build/starkinfo.json -v build/fibonacci.verkey.json -o build/
    fibonacci.proof.json -b build/fibonacci.public.json

Verification Ok!!
```

9. Next, we generate an eSTARK verifier as a circuit in Circom:

```
$ node node_modules/pil-stark/src/main_pil2circom.js -p src/fibonacci.
    pil -s build/starkinfo.json -v build/fibonacci.verkey.json -o build
    /verifier.circom

Verification Ok!!
```

10. We compile the circuit:

```
$ circom --O1 --prime goldilocks --r1cs --wasm build/verifier.circom -l
     node_modules/pil-stark/circuits.gl -o build

Verification Ok!!
```

11. Compute the witness for our inputs:

```
$ snarkjs wc build/verifier_js/verifier.wasm build/fibonacci.zkin.json
    build/verifier.wtns

out = 3524578
```

12. Convert the resulting circuit to a state machine:

```
$ node node_modules/pil-stark/src/main_plonksetup.js -r build/verifier.
    r1cs -p build/verifier.pil -c build/verifier.const -e build/
    verifier.exec

files Generated Correctly
```

13. Populate the committed polynomials in the execution trace:

```
$ node node_modules/pil-stark/src/main_plonkexec.js -w build/verifier.
    wtns -p build/verifier.pil -e build/verifier.exec -m build/verifier
    .commit

files Generated Correctly
```

14. Verify that the execution trace generated in the previous two steps is valid:

```
$ node node_modules/pilcom/src/main_pilverifier.js build/verifier.
    commit -p build/verifier.pil -c build/verifier.const

PIL OK!!
```

15. Create a `verifier.starkstruct.json` file with the eSTARK parameters in the same directory:

```
$ cat <<EOT > verifier.starkstruct.json
{
    "nBits": 16,
    "nBitsExt": 17,
    "nQueries": 64,
    "verificationHashType": "GL",
    "steps": [
        {"nBits": 17},
        {"nBits": 14},
        {"nBits": 10},
        {"nBits": 6}
    ]
}
EOT
```

Now, we can proceed as in the previous example to generate the eSTARK.

16. Generate the `starkinfo.json` file needed to generate the eSTARK:

```
$ node node_modules/pil-stark/src/main_genstarkinfo.js -p build/
    verifier.pil -s src/verifier.starkstruct.json -i build/verifier.
    starkinfo.json

files Generated Correctly
```

17. Preprocess the state machine:

```
$ node node_modules/pil-stark/src/main_buildconsttree.js -c build/
    verifier.const -p build/verifier.pil -s src/verifier.starkstruct.
    json -t build/verifier.consttree -v build/verifier.verkey.json

files Generated Correctly
```

18. Generate the eSTARK proof:

```
$ node node_modules/pil-stark/src/main_prover.js -m build/verifier.
    commit -c build/verifier.const -t build/verifier.consttree -p build
    /verifier.pil -s build/verifier.starkinfo.json -o build/verifier.
    proof.json -z build/verifier.zkin.json -b build/verifier.public.
    json

files Generated Correctly
```

19. Finally, verify the proof:

```
$ node node_modules/pil-stark/src/main_verifier.js -p build/verifier.
    pil -s build/verifier.starkinfo.json -v build/verifier.verkey.json
    -o build/verifier.proof.json -b build/verifier.public.json

Verification Ok!!
```

As explained at the beginning, one can repeat this workflow by again generating an eSTARK verifier as a circuit written in Circom and following Steps 10-18, creating `starkstruct.json`'s files at each repetition.

References

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl, July 2018.

[GW20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 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, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

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

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