polygon zkEVM

Knowledge Layer

Architecture

Basic Principles of the Polygon zkEVM Proving System

Version: 70b684c7c12e57010971e955c66917ef5bc3a100

January 31, 2024

1 Introduction to the Prover

The **Prover** is a software component whose main goal is to generate a proof that for the correct execution of a given program with an specific set of inputs. While the process of generating a proof is resource-consuming, the time required to verify a constructed proof is significantly shorter, enabling it to be done by a smart contract. The main usage of the prover in the zkEVM context, presented in Figure 1, is to generate a proof π_{i+1} stating that, given a batch of transactions $Batch_i$ and a L2 State $S_i^{L2_x}$, the state has transitioned to $S_{i+1}^{L2_x}$.

Figure 1: The Prover component generates proof π_{i+1} that state the accuracy of the state transition.

To produce this proof, the initial step involves the creation of an execution matrix. An **execution matrix** (also called, **execution trace**) is a matrix that records all the intermediate computations constituting a larger computation. In the zkEVM context, we can think that the big computation is the state transition function and the little intermediate ones are the zkEVM instructions or opcodes. As we will note later on, a single EVM opcode may be implemented using one or more zkEVM opcodes, indicating a lack of a strict one-to-one correspondence between EVM opcodes and zkEVM ones.

1.1 Execution matrix: A Toy Example

Suppose that we want to model the following computation

$$[(x_0 + x_1) \cdot 4] \cdot x_2$$

using an execution trace, being $x := (x_0, x_1, x_2)$ a given set of inputs. Also suppose that the only available instructions (or operations) are the ones described below:

1. Copy inputs into cells of the execution trace.
2. Sum two cells of the same row, and leave the result in a cell of the next row (referred to as the `ADD` instruction).
3. Multiply by a constant, and leave the result in a cell of the next row (referred to as the `TIMES4` instruction).
4. Multiply two cells of the same row, and leave the result in a cell of the next row (referred to as the `MUL` instruction).

We will use a matrix with three **columns** `A`, `B` and `C`. Also suppose that the number of rows is bounded by some constant N, which we will call **length** of the execution trace. We can depict it as in Figure 2. The columns of an execution trace are often called **registers** (so we may name them interchangeably).

A	B	C
a_0	b_0	c_0
a_1	b_1	c_1
...	...	...
a_N	b_N	c_N

Figure 2: General form of an execution trace of length N having three registers `A`, `B` and `C`.

Let's provide an specific example giving the following set of inputs $x = (1, 2, 5)$. We can model our desired computation

$$[(x_0 + x_1) \cdot 4] \cdot x_2 = [(1 + 2) \cdot 4] \cdot 5 = 60.$$

to fit our execution trace using only the available operations as follows:

step
1
2
3
4

A	B	C
1	2	
3		4
12	5	
60		

$[a_0 = x_0, b_0 = x_1]$	ADD
	TIMES4
$[b_2 = x_2]$	MUL

Figure 3: Execution trace modeling the computation $[(x_0 + x_1) \cdot 4] \cdot x_2$ with the specific set of inputs $(1, 2, 5)$.

Let's walk through the execution trace step by step:

1. First of all, we use the instruction that copies the inputs 1 and 2 into the columns A and B respectively and then we invoke the ADD instruction.
2. Following the ADD instruction, the second row of register A now holds the sum of 1 and 2, resulting in 3. At this point, we utilize the TIMES4 instruction to obtain $3 \cdot 4 = 12$.
3. With 12 now in the third row of register A, we proceed to copy the third input, 5, into register B. The MUL instruction is then invoked to multiply 12 by 5, representing the multiplication of the current values in registers A and B.
4. Consequently, we obtain 60 as the outcome of the MUL instruction, which is the final output for the entire computation.

Observe that in the procedure above we have only used the available instructions proposed in our scenario. Also notice that, for any other choice of x, the shape of the matrix would remain the same, but most of the entries would be different. For example, the matrix in Figure 4 appears when using $x = (5, 3, 2)$ as input:

A	B	C
5	3	
8		4
32	2	
64		

Figure 4: The same program with other input gives the same shape but different values.

If examine the color-coded columns in the execution trace we can identify two distinct types:

- **Witness columns** (colored in white): These columns depend on the input values. For example, the columns A and B change whenever we input different sets of values.
- **Witness columns** (colored in gray): In contrast, these columns remain unaffected by the input values. In this example, the column C is constant because it is only used to store the 4 which is used always in the second step of the computation when invoking the TIMES4 instruction. Note that fixed columns dont change as they are an intrinsic part of the computation.

2 The Executor

The **Executor** (Figure 5) is the component whose main purpose is to generate a correct execution trace from a given set of inputs.

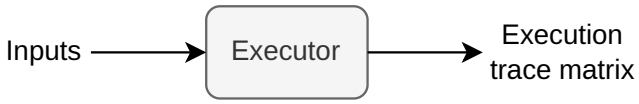

Figure 5: The goal of the Executor is to take some inputs and generate an execution trace.

There are two approaches when implementing the executor component, depending on whether we prefer single computation executor, that is, an executor that only works for a single specific program or a general purpose executor, which can run several computations.

Single-computation executor A **single-computation executor** is specifically tailored for a particular computation. This means that changing this computation involves modifying the executor. This approach is the faster computationally speaking but at the cost that the single-computation executor is not easy to change, test or audit. We also call this executor a **native executor**. An electronic analogy of the native executor are Application Specific Integrated Circuitss (ASICs) circuits. An ASIC is a circuit specifically designed to run, very efficiently, a single computation.

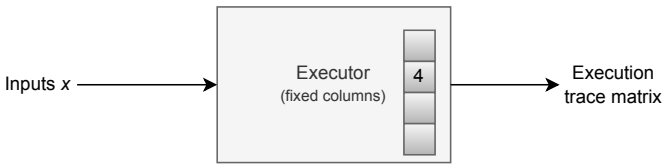

Figure 6: A single computation executor always perform the same operation maybe with different sets of inputs.

General-computation executor General-computation executors can run a wide variety of programs. In this approach, the executor component not only reads the inputs, as in the previous approach but also interprets a program that guides the executor on how to compute the execution trace for a specific calculation.

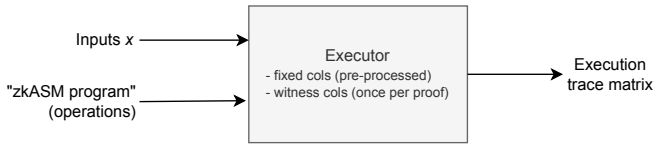

Figure 7: In this case, we should add a zkASM program as an input for the Executor.

For example, below we have two different computations and their respective execution traces. These two are computed by the same general-executor but with different programs (that is, different ordered sets of instructions describing the computation) and different inputs.

We already know the first program, explained before (Figure 3). It computes $[(x_0 + x_1) \cdot 4] \cdot x_2$ using $(1, 2, 5)$ as set of inputs. Recall that we model it using the available instructions. However, we can modify the used instructions, its order and the set of inputs and obtain the following execution trace:

step	A	B	C		
1	2		4	$[a_0 = x_0]$	TIMES4
2	8		4		TIMES4
3	32	3		$[b_2 = x_1]$	MUL
4	96				

Figure 8: Modifying the used instructions and its order we can get different execution traces modeling a wide variety of computation.

Observe that the former execution trace (Figure 8) actually models the following computation

$$(x_0 \cdot 16) \cdot x_1.$$

In our context, input programs are written in a language called **zkASM**, read as *zk-assembly*. zkASM is the language developed by the team that is used to write instructions for the general executor. In the repository `zkasmcom-vscode` there is a syntax highlighter for VSCode. An example of zkASM which is present in the project can be found in Figure 9.

```
1  STEP => A
2  0                                   :ASSERT ; Ensure it is the beginning of the execution
3
4  CTX                                 :MSTORE(forkID)
5  CTX - %FORK_ID                      :JMPNZ(failAssert)
6
7  B                                   :MSTORE(oldStateRoot)
```

Figure 9: zkASM Language Example.

We have implemented a zkASM compiler (See Figure 10) that reads a zkASM specification file and compiles it to an output file with the list steps and instructions which the executor will consume in order to compute the execution trace.

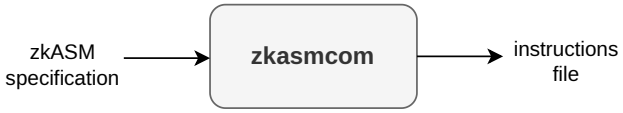

Figure 10: The zkASM compiler reads a program written in a `.zkasm` file and outputs a `.json` file indicating the instructions that the executor needs to follow in order to fill the execution trace accordingly.

In Figure 11 we present the advantages and drawbacks of each approach. The single-computation executor is faster because it does not need to read assembly; it can efficiently generate the execution trace for the specific computation, optimizing the process for that particular case. Nonetheless, the drawback lies in its inflexibilitybeing challenging to modify, test, or audit. The general-executor approach is needed within the zkEVM context because both the EVM and the zkEVM evolve, and it is more efficient to keep modifying the assembly code than the whole executor. The zkASM program that processes EVM transactions is called **zkEVM ROM** (Read Only Memory) or simply the ROM. By changing the ROM, we can make our L2 zkEVM more and more closer to the L1 EVM. Henceforth, we have different versions of the zkEVM ROM. Each of these versions will be denoted with a unique identifier called `forkID`. It is worth mentioning that another advantage of using a ROM-based approach is that we can test small parts of the assembly program in isolation. However, we will have both, each one serving different purposes. The single-computation executor is still work in progress.

Executor Type	Pros	Cons
Single-computation	Faster	Less flexible
General-computation	More flexible	Slower

Figure 11: Pros and Cons of each Single-computation and General-computation Executors.

3 Polynomial Identity Language (PIL)

In the previous section we have discussed about **execution**. However, since we are dealing with the proving system, we need some way to state the **correctness** of a given execution. The execution correctness is enforced by a set of constraints that must be fulfilled by the execution trace. This constrains are a set of identities that impose the relations between the different cells of the execution trace, determining them uniquely. Recall the previous computation example $[(x_0 + x_1) \cdot 4] \cdot x_2$. In this case, the correctness of the execution trace is granted by the following set of constraints.

Program (computation):

$$[(x_0 + x_1) \cdot 4] \cdot x_2$$

A	B	C
1	2	
3		4
12	5	
60		

Constraints:

$$a_0 = x_0$$
$$b_0 = x_1$$
$$a_1 = a_0 + b_0$$
$$a_2 = a_1 \cdot 4$$
$$b_2 = x_2$$
$$a_3 = a_2 \cdot b_2$$

Figure 13: The set of constraints (Right) for the execution trace (Left). It can be seen that if the constraints are satisfied, the program modeled using the execution trace is, indeed, correct. The reader can check that the values on the left satisfy the constraints on the right.

Remark. We would like to warn the reader that thorough this document we will explain the concepts of the execution trace and the constrains in a didactic way, constructing examples and adding more complexity step by step. For this reason, some things like the way to write the constrains are not fully correct, but this will be fixed by the end of the document.

In our cryptographic back-end, each column of the matrix (equivalently, each register) is transformed it into a polynomial having degree the number of rows of the execution trace which, by performance matters, will be a power of two (in order to be possible to perform an interpolation via FFTs). This transformation is done by interpolating the values of a column over some domain (more precisely, a subgroup of roots of unity). That is, we change arithmetic constraints by polynomial constraints by means of interpolation. Constraints are actually defined among these polynomials using a language called **PIL** (Polynomial Identity Language).

We have implemented a PIL compiler that reads a PIL specification file and compiles it to output a JSON file. This file contains the list of all the constraints and additional informtaion about the polyomials that can be consumed by the prover (See Figure 14). The repository `pilcom-vscode` contains a PIL syntax highlighter for VSCode.

Figure 14: The `pilcom` compiler is in charge of compiling a file `.pil` written in PIL and obtain a `.json` file which will be consumed directly by the prover in order to generate the proof.

Putting all this together, Figure 15 shows a simplified scheme that outlines the systematic process of generating and verifying proofs. The cryptographic backend receives the results of the execution (that is, both the witness and fixed columns) and the constraints, compiled using `pilcom`. The prover is in charge of generate the proof π and the given set of publics, which are a set of values which are known both by the prover and the verifier (See Section 3.1 for more details). Following this, the verifier gains the capability to verify the correctness of the computation performed by the prover. The verification process involves using a specialized verification algorithm having as inputs the generated proof π and the provided public values. The successful outcome of this verification signifies that the prover executed the computation accurately.

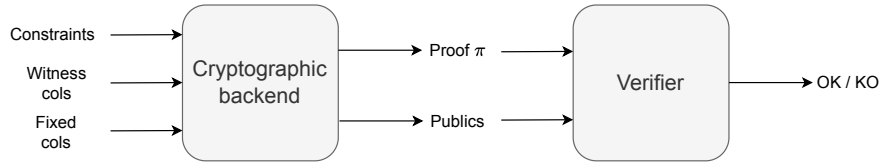

Figure 15: The cryptographic backend receives the execution results and constraints and generates proof π and the corresponding public values. The verifier will use them to confirm the accuracy of the computation.

As we have said, a valid proof π assures the verifier that a given computation is correct given specific public inputs. This proof is **compact** and demands minimal resources for validation. This fact enables smart contracts to act as verifiers. Our current setup relies on a back-end cryptographic system, where the ultimate verifier (we will see the meaning of this later on) is an algorithm called **FFlonk**. Practically, a smart contract in the Layer 1 implements this algorithm for the verification of the proof. The gas cost for this validation is approximately of 200,000 gas units.

3.1 Public and private inputs

Zero-Knowledge technology enables the distinction between public and private values within the cells of the execution trace, providing the capability to designate which information should be publicly available and which should remain confidential. For instance, one application involves proving knowledge of the pre-image of a hash having publicly known digest `hash_value` while keeping the actual pre-image value `hash_pre-image` confidential (See Figure 16):

$$\texttt{hash\_value} = H(\texttt{hash\_pre-image}).$$

Publics are key important not only in the proving phase, as they are used to compute the execution trace, but also in the verifying phase. As shown in Figure 15, they are given to the verifier together with the proof so the verifier can actually verify the computation done by the prover. By default, all values in PIL are considered private. However, you can specify a particular value to be public by using the keyword `public`.

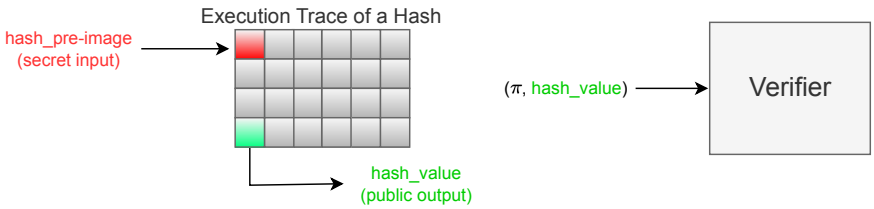

Figure 16: Demonstrating the distinction between public and private values, this figure illustrates a hash pre-image proof. `hash_value` is public, while `hash_pre-image` remains confidential. The initial cell of the execution trace signifies the secret input, and through a sequence of instructions (designed to model the hash function H), it culminates in the production of the hash output. Subsequently, the cryptographic backend generates proof π and public values, intended for verification by the verifier.

4 Execution Trace Shape Design

In this section we will discuss some aspects about the shapes (or dimensions) of the execution traces. In an execution trace, each row is in charge of validating a **single zkASM operation** or **part of an operation**. For example, suppose that we are given a set of 3 operations implemented in zkASM `OP1`, `OP2` and `OP3`. These operations change the next value of the `A` register, as show in Figure 17. Take into account that primes mean next value of some column. For example a' means the next value in the `A` column.

$$\begin{aligned}
&\texttt{OP1} : a' = a + b + c \\
&\texttt{OP2} : a' = a + b + c + d + e \\
&\texttt{OP3} : a' = a + b + c + d + e + f + g + h
\end{aligned}$$

Figure 17: This constraints define how each of the operations work within the execution trace.

Consider also that we have an execution trace matrix of 6 columns as shown below:

A	B	C	D	E	F

`OP1` and `OP2` can be easily fit in the matrix, but `OP3` gives a problem at first look, since we have 8 summands but only 6 registers. The question is: *can we fit this computation inside the matrix? If so, how can we do it?* We purpose two straightforward strategies:

1. Increasing the number of columns so that we can fit every summand. More specifically, we can add registers `G` and `H`. This strategy has several contras: one the one hand, adding extra columns increases the proving time and, on the other hand, wider matrices may have too many unused cells and, furthermore, they might be inefficient for mixing many different instructions. We will see examples of the later right below.

2. Use the next row in order to fit some of the remaining operands of the operation. Increasing the number of rows is also delicate, because, since the number of rows is

a power of two, we can only double the size of the matrix, not add extra rows one by one. So we have to be careful when adding rows, making sure that (if possible) we do not double the size of the matrix unnecessarily, since the cost of doing so

Here, we face a dilemma regarding whether to expand the execution matrix in width or length. Later on we will introduce a third approach involving *look ups*. For now, let's examine through an example how we can accommodate OP3 within a 6-column matrix:

Working Example We will use the second approach. We can define an execution matrix in which OP3 uses two rows: we store g and h in the next row filling the next values of the registers B and C. See Figure 18 to visualize the new arrangement of OP3. Then we can perform OP3 as follows, keeping both OP1 and OP2 operating in the same way:

$$\texttt{OP3} : a' = a + b + c + d + e + f + b' + c'$$

A	B	C	D	E	F	
a_0	b_0	c_0				OP1
a_1	b_1	c_1	d_1	e_1		OP2
a_2	b_2	c_2	d_2	e_2	f_2	OP3
a_3	b_3	c_3				

Figure 18: In this table the yellow cell represent the right hand side operands of the OP3 instruction. The green cell represents the output of summing together all the yellow cells. Note that there are other possible arrangements in order to fit OP3 considering the shape of this execution matrix.

This example shows that we can reduce the number of needed columns by increasing the number of rows. In the next example we will show how to optimize the proving system representing the computation of applying OP1 and OP2 consecutively by means of simply modifying the shape of the execution matrix.

One approach may be using 6 columns as before A, B, C, D, E and F, as shown in Figure 19. In this case we fit the computation in 2 rows, having the output in the third row.

A	B	C	D	E	F	
a_0	b_0	c_0				OP1
a_1	b_1	c_1	d_1	e_1		OP2
a_2						

Figure 19: Design of the execution trace having 6 columns. Blue cells represent unused cells.

Here, we have a configuration with 6 columns and 2 rows, with an additional row used for allocating the output. This results in a total of 18 cells, with 9 cells remaining unused. It's important to highlight that the unused cells constitute half of the matrix's space, which seems quite inefficient.

Alternatively, we can arrange this problem by redefining OP2 to use 3 columns and 2 rows as follows:

$$\texttt{OP2} : a' = a + b + c + a' + b'$$

This way, we can remove the three last registers and only use a matrix half bigger than the last one for the same purpose.

A	B	C	
a_0	b_0	c_0	OP1
a_1	b_1	c_1	OP2
a_2	b_2	c_2	

Figure 20: In this revised layout, we utilize the entire space, ensuring that no cells remain unused.

In this case, we have a configuration with 6 columns and 3 rows. This results in a total of 18 cells, all of them used. To sum up, **the number of unused cells strongly depends on the instructions executed and the shape of the execution matrix.**

5 Selector Columns

Until now, to illustrate the behavior of the general executor we have been talking about the *k-th row performing an instruction* OPx. But since we are using constraints with the entire columns and not individual cells (recall that we are actually interpolating), we need to add **selector columns** in order to indicate which operation we are performing in each row and reflect it within constraints. In other words, we have to ensure that the set of constraints is fulfilled in **every row**. Selector columns are used to control whether the constraints of an operation apply or not, meaning whether in a specific row we apply OPx or OPy. For example, we add two selector columns OP1 and OP2 into the previous example as shown in Figure 21.

$$\texttt{OP1} : a' = a + b + c$$
$$\texttt{OP2} : a' = a + b + c + b' + c'$$

A	B	C	OP1	OP2
a_0	b_0	c_0	1	0
a_1	b_1	c_1	0	1
a_2	b_2	c_2		

Figure 21: Columns OP1 and OP2 flag whenever one of these two operations is being performed.

But now, we need to introduce this columns into the constraints so that they become satisfied in every row. In this case, we introduce the following constraint:

$$op1 \cdot (1 - op2) \cdot (a + b + c - a') + op2 \cdot (1 - op1) \cdot (a + b + c + a' + b' - c') = 0$$

Remark. Observe that, at this moment, constraints are not **instruction related** anymore. In previous sections, we constraint each of the operations separately, which can not be proved with any known proving system directly.

We can check that if we are in the first row, then $op1 = 1$ and $op2 = 0$, so the constrain becomes:

$$1 \cdot (1 - 0) \cdot (a + b + c - a') + 0 \cdot (1 - 1) \cdot (a + b + c + a' + b' - c') = \boxed{a + b + c - a' = 0},$$

which corresponds to performing OP1 in te first row. Same can be checked for OP2. Note that this selector columns do not depend on the inputs but on the computation itself, so they can be preprocessed.

6 The Execution Trace and the zkEVM

In our current cryptographic back-end, the dimension of the execution trace is predetermined. Additionally, Also, we dont know exactly what EVM opcodes (and as a consequence zkEVM operations) will be executed, since this depends on the particular transactions of the L2 batch (See Figure 22). The pre-fixed shape fixes in turn the amount of computation that we can do, which in our case is the amount and type of L2 transactions for which we can generate a proof. In general, it is hard to optimize the shape of a single execution trace matrix:

- **Narrow matrices may easily hit the max row limit**: Increasing the number of rows is delicate because, since the number of rows is a power of two, we can only double the size of the matrix, not add extra rows one by one. So we have to be careful when adding rows, making sure that (if possible) we do not double the size of the matrix unnecessarily.
- **Wide matrices might be inefficient:** Wide matrices may have too many unused cells and, furthermore, they might be inefficient for mixing many different instructions. Moreover, adding more columns also increase proving time.

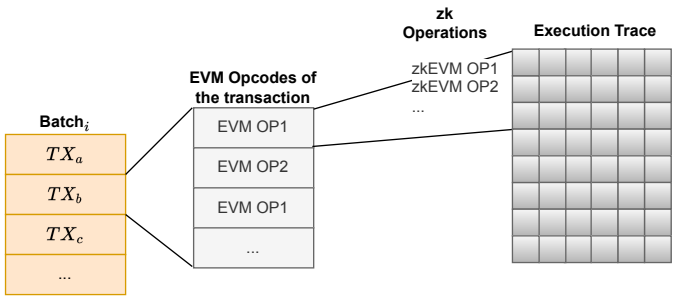

Figure 22: Every transaction in a batch consists of multiple EVM Opcodes, and each EVM Opcode corresponds to multiple zkEVM Opcodes, the instructions in charge of filling the execution trace.

7 zkEVMs Compatibility/Equivalence

In the post *The different types of ZK-EVMs*, Vitalik Buterin explores the concept of EVM compatibility/equivalence. We say that layer 2 is EVM compatible or equivalent if it can run EVM byte code without modifying the underlying smart contract logic. EVM compatibility allow L2s to use existing Ethereum smart contracts, patterns, standards, and tooling. Being EVM compatible is important for the widespread adoption of these L2 since this allows using existing tools can be used. In practice, there are several types of compatibility (See Figure 23):

- **Type 1**: Fully Ethereum equivalent, i.e., they do not change any part of the Ethereum system but generating proofs can take several hours.
- **Type 2**: Fully EVM-equivalent, but changes some different internal representations like how they store the state of the chain, for the purpose of improving Zero-Knowledge proof generation times.

- **Type 2.5**: Fully EVM-equivalent, except they use different gas costs for some operations to "significantly improve worst-case prover times".

- **Type 3**: Almost EVM-equivalent zkEVMs make sacrifices in exact equivalence to further enhance prover times and simplify EVM development.

- **Type 4**: High-level language equivalent zkEVMs compile smart contract source code written in a high-level language to a friendly language for Zero-Knowledge, resulting in faster prover times but potentially introducing incompatibilities and limitations.

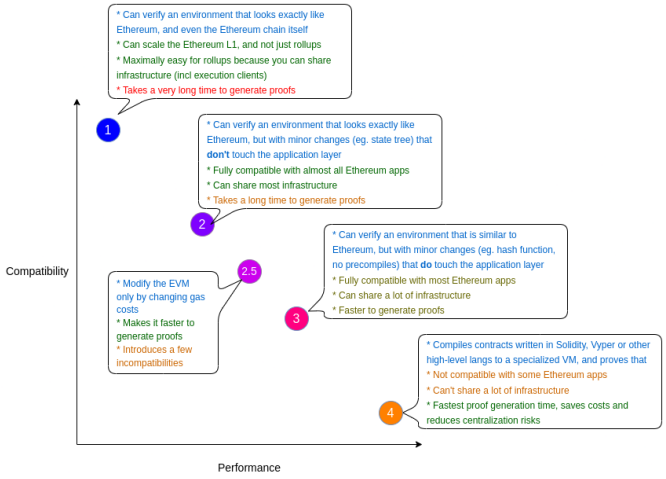

Figure 23: Vitalik Buterin's categorization of ZK-EVM compatibility types.

At present, our compatibility stands at type 2.5, with ongoing efforts directed towards achieving type 2. The objective is not to attain type 1 compatibility, as type 2 yields greater throughput when using zero-knowledge technologies. A notable distinction lies in the storage mechanisms employed, where Ethereum L1 utilizes a Patricia Trie, which proves to be highly inefficient for verification. In contrast, zkEVM employs a binary sparsed Merkle tree, offering enhanced efficiency for proof generation.

8 Secondary Execution Trace Matrices and Lookup Arguments

8.1 Introduction

Lets assume that our cryptographic backend only allows to define constraints with additions and multiplications. Lets consider also that we want to implement a exponentiation operation (which we will denote by `EXP`). Then, we can use several rows doing multiplications to implement `EXP`. A portion of the execution trace (that implements the operation $2^5 = 32$) is shown in Figure 24.

A	B	C
2	5	2
2	4	4
2	3	8
2	2	16
2	1	32

Figure 24: Execution trace modeling the exponentiation operation 2^5. Blue cells represent the inputs and the green cell the output of the computation.

Where the columns `A`, `B` and `C` have the following constraints between them:

1. $a' = a$, the `A` column represents the base.
2. $b' = b - 1$, the `B` column stores the decreasing exponent.
3. $c' = c \cdot a$, the `C` column stores the intermediate results.

When implementing this operation in the main execution trace, it's important to know that each `EXP` operation will consume multiple rows. In fact, The exact number of rows required by `EXP` varies depending on the exponent. This approach leads to a very complicated set of constraints together with a huge amount of consumed rows by operation. To solve this, we can take another approach in using **tailor-made secondary execution traces** for specific operations.

In this approach, there is a **main execution trace** and there are also **secondary execution traces**. In the cryptographic back-end, we use a mechanism called lookup argument to link these execution trace matrices. In particular, the lookup argument provides the constraints necessary to check that certain cells of a row in an execution trace matrix match other cells in a row of another execution trace matrix.

8.2 Linking Execution Traces via Lookup Arguments

Let us continue with the `EXP` example to show how the several execution traces approach works. We will have **two execution traces**. In the **main execution trace**, shown in Figure 25, each `EXP` operation will occupy just one row. For the first EXP operation, inputs are 2 and 5 and the result is 32. Observe that the column `EXP` flags whenever an `EXP` operation is performed. In such rows, we should ensure that the value of the column `C` precisely corresponds to the result of exponentiating the value in column `A` to the value in column `B`. The correctness of the result will be validated in a **secondary execution matrix**.

Remark. An execution trace matrix can be seen as a set of states **(state machine)**, in which each row is a state. We will use both, the terms **execution trace matrix** and **state machine** interchangeably.

The secondary state machine will operate very similar as the one shown before in Figure 24. In this case, the columns have the following sense:

1. The `A` column represents the base. Observe that this column is allowed to contain several values as we may check several `EXP` operations in the same state machine as long as we do not consume more rows than the fixed maximum.
2. The `B` column stores the exponent.
3. The `C` column stores the intermediate results.

A	B	C	EXP
···	···	···	0
2	5	32	1
···	···	···	0
···	···	···	0
···	···	···	0
···	···	···	0
3	2	9	1
···	···	···	0
···	···	···	0

Figure 25: Depiction of the main execution trace. When `EXP` column is equal to 1 means that we should check that the value of `C` can be derived by exponentiating the value in column `A` to the power of the value in column `B`. To do so, we will use a secondary execution trace.

4. The column `D` stores a decreasing counter. In each `EXP` operation, it starts with the same value as the exponent and keeps decreasing until arriving 1, meaning that the current operation has been finished.

5. The column `EXP` flags whenever the operation is actually finished. Rows with `EXP` equal to 1 are the ones where we can state consistency of `EXP` operations of the Main State Machine via a lookup argument.

The connection between the two state machines will be established through a lookup argument. Whenever `EXP` selector's value in the Main State Machine is 1, it will be searched whether the values in the columns `A`, `B` and `C` of the same row, are present in the columns `A`, `B` and `C` respectively of the Secondary State Machine in a row also having `EXP` selector equal to 1. Affirmative response will mean that the `EXP` operation in the Main State Machine is performed correctly and, otherwise, that the prover has cheated.

Main State Machine

A	B	C	EXP
···	···	···	0
2	5	32	1
···	···	···	···
3	2	9	1
···	···	···	0

: Lookup

`EXP` Secondary State Machine

A	B	C	D	EXP
2	5	2	5	0
2	5	4	4	0
2	5	8	3	0
2	5	16	2	0
2	5	32	1	1
3	2	3	2	0
3	2	9	1	1

Figure 26: The linkage between the Main State Machine and the `EXP` State Machine is done by checking equality between yellow rows of both state machines. In the Main SM, we just put the inputs/outputs, that we call *free*, in a single row.

Recall that there is a PIL compiler that reads a PIL specification file and compiles it to an output file with the list of constraints and a format that can be consumed by the prover to generate the proof. In the PIL language, the state machines, that is, the different subexecution matrices linked together via lookup arguments, are called `namespaces`. In the `zkevm-proverjs` repository, you can find the PIL specification of the whole zkEVM. There are several files, one for each different state machine, such that `binary.pil`, that is responsible for constraining **binary operations** (such that additions of 256 numbers

represented in base 2 and equality/inequality comparators), or `mem.pil`, which is responsible for managing **memory-related opcodes**. The Main State Machine is constrained in the `main.pil`, which serves as the main entrypoint for the whole PIL of the zkEVM.

8.3 Final Remarks and Future Improvements

The columns of each state machine are defined by the design of its corresponding execution trace. Due to constraints in our existing cryptographic backend, it is mandatory that all state machines share the same number of rows. The computation of an L2 batch can have branches and loops and hence, each L2 batch execution can use a different number of operations in the zkEVM. Consequently, the number of rows utilized by each state machine depends on the specific operations carried out during batch execution. Since the number of rows is fixed (and the same for all state machines) we can have unused rows. But, what is more important is that obviously, the size of the computation being proved must fit in the execution trace matrices available.

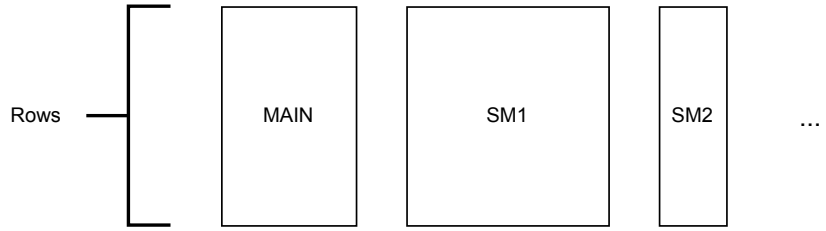

Figure 27: Due to the specific cryptographic backend used in the prover, all the same state machines should have the same fixed amount of rows.

Currently, we are in the process of developing a new version of PIL, referred to as **PIL2**. **PIL2** is designed to operate with a more potent cryptographic backend capable of generating an adequate number of subexecution traces to accommodate the whole batch processing without running out of rows. Additionally, we are collaborating with other projects within the *zk projects* at Polygon to establish a standardized format for the PIL output file, named `pilout`.

9 Recap of the Prover and the Verifier

The Prover, shown in Figure 28, is the component within the Proving System which is in charge of generating a proof for the correct execution of some program. The program, which takes care of the computational aspect of the computation, is written in a language called **zkASM**, developed by the Polygon team. Providing also the private and the public inputs, the **Executor** component within the Prover is able to generate the execution traces designed to model the willing computation. The executor produces two binary files: one containing the **fixed columns**, which should only be generated once if we do not change the computation itself, and the other containing the **witness columns**, which vary with inputs and thus need to be generated anew for each proof. The files containing the preprocessed fixed columns and the processed witness columns for the zkEVM are temporary stored in binary files and are quite large, consisting on more than 100Gb. Subsequently, the cryptographic backend of the prover, in conjunction with the compiled PIL constraints through `pilcom` and the execution trace binary files generated earlier, can produce the proof and provide the public values, both inputs and outputs, for the verifier.

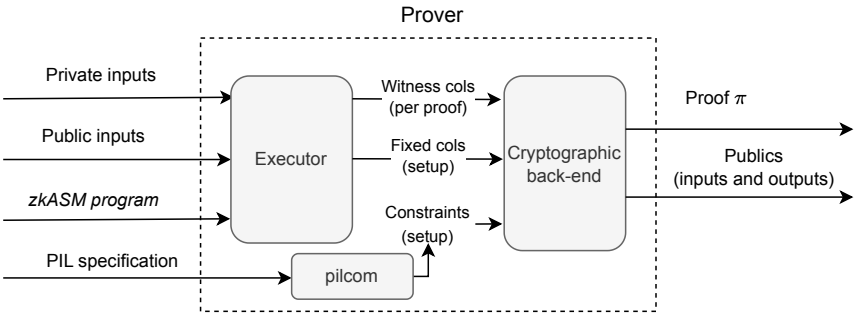

Figure 28: Schema of the Prover component. The Prover component has 3 subcomponents: the Executor, which is in charge of populating the execution trace based on some computation with values depending on some inputs, the PIL compiler or `pilcom`, which compiles PIL files into JSON files prepared to be consumed by the Prover, and the cryptographic backend, which takes the output of both the Executor and the `pilcom` and generates the proof and the publics for the verifier.

Posteriorly, the verifier (depicted in Figure 29) use both the proof and the public values to check if the prover has performed the computation in a correct way.

Figure 29: The verifier utilizes both the proof and the public values to validate the correctness of the computation performed by the prover.