The zkEVM Architecture

Part I: zkEVM Proving System Principles

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>
Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

Version: 784c6bb2ad019af81d627151d79122dffed4755b

December 30, 2023

Outline

Pre-requisites

Basics of the Ethereum L1 execution layer:

- How smart contracts work.
- Basics about token smart contracts (mainly ERC20).

Basics of cryptography concepts:

- Digital signatures.
- Hashes and Merkle trees.

Programming languages:

- Golang (node), C++ (executor/prover), Solidity (smart contracts) and Javascript (miscellaneous).

Outline

L2 Scaling and Strategies

Review the road to scalability and the L2 scaling strategies at the concepts.

L2 Design for the Polygon zkEVM

a) How users send L2 transactions and who receives them?
   - The zkEVM uses **unicast** to let user send their transaction (calls to an RPC).
   - The zkEVM also enables posting L2 transactions via a method in a **smart contract** as an anti-censorship measure (called ”forced batches”).

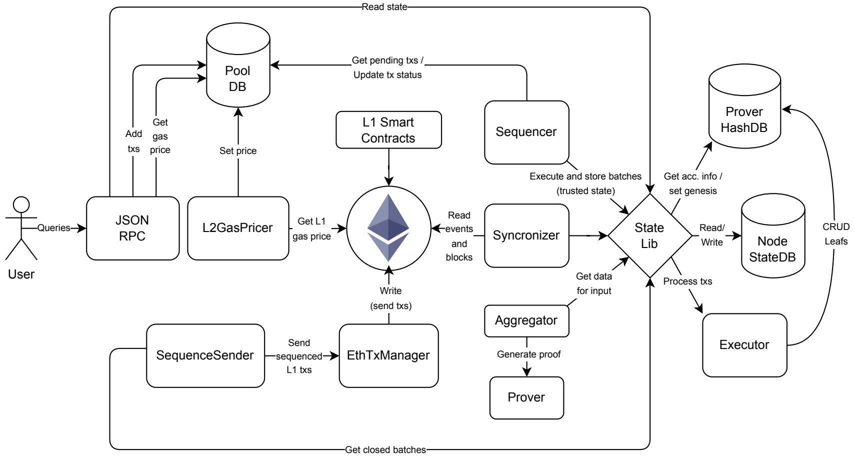
b) How L2 transactions are made publicly available (if so)?
   - The zkEVM is a **rollup**, the L2 data is available in L1.

c) Who processes the L2 transactions and how, and, when it is publicly considered that a new state is correctly computed?
   - In the zkEVM, currently, there is a **centralized aggregator node** that proves the processing of the L2 transactions.
   - However, this node cannot cheat because there is a **succinct computation verification** (using zero-knowledge technology).

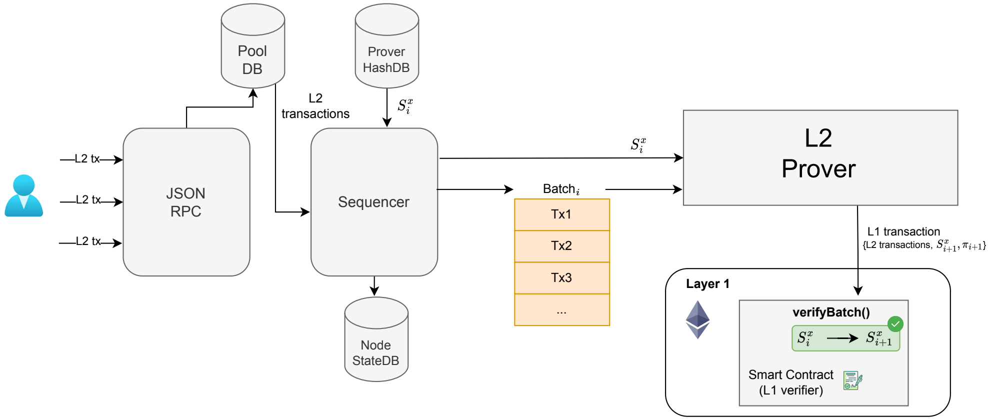
d) What type of applications the L2 supports? simple or rich processing?
   - zkEVM is rich processing since it is an **EVM**.

Outline

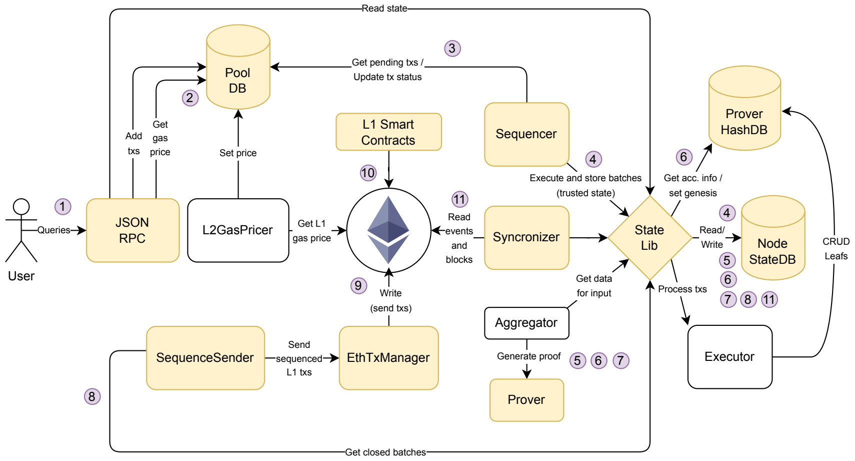
Complete Architecture of the zkEVM



Basic zkEVM L2 Processing (Simplified Flow)



Processing an L2 zkEVM Transaction (Simplified Flow) i



Remark. By now, consider that white boxes don't exist. Also take into account that the functionality of some yellow boxes will be re-engineered regarding this simplified flow.

Processing an L2 zkEVM Transaction (Simplified Flow) ii

1. The **user** creates a standard Ethereum transaction for the L2 (e.g. using the metamask wallet) and, sends it to the **JSON RPC** API of the node, which is an almost standard Ethereum JSON RPC with some extra endpoints.
2. The **JSON RPC** stores the received L2 transactions in the **pool** database of pending L2 transactions.
3. The **sequencer** creates (closes) a batch by selecting L2 transactions from the **pool** (with some criteria).
4. The **sequencer** stores the data of the new batch in the node's **StateDB**.
5. The **prover** queries the node's **StateDB** to read the data of the new batches to be proved.
6. The **prover** also reads the **HashDB** to obtain the necessary data to proof the current L2 state (root of the L2 state and hashes for Merkle proofs).
7. The **prover** generates the proof and stores it with its related data in the node's **StateDB**.

Processing an L2 zkEVM Transaction (Simplified Flow) iii

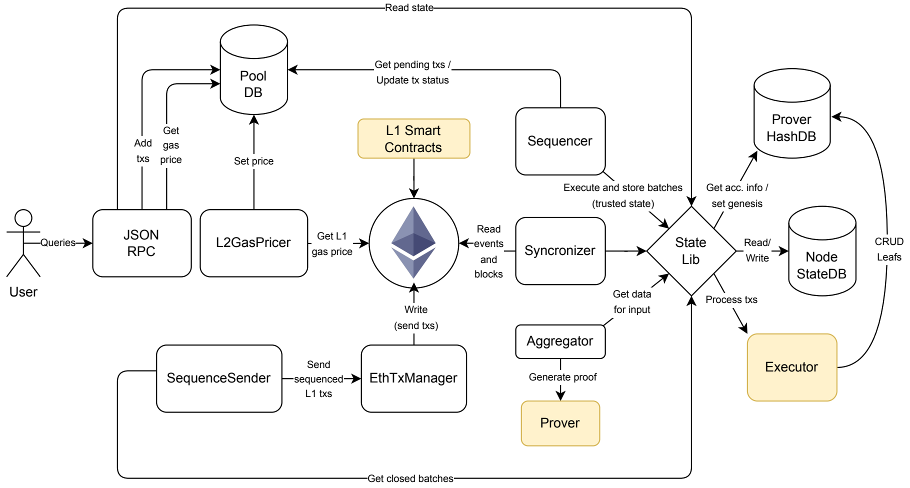
8. The sequenceSender reads the node's StateDB checking for any new proved batches.
9. The sequenceSender decides when it is the best moment to create and send the L1 transaction with the proof to the L1 zkEVM smart contract. The sequenceSender sends the transaction through the EthTxManager. The EthTxManager uses an L1 Ethereum node to do so (e.g. geth/prysm) and it makes sure that the transaction is included in a block (managing the L1 gas fees if necessary).
10. The L1 zkEVM smart contract processes the transaction and, if the proof is correctly verified, updates and stores the new L2 state.
11. Finally, the synchronizer, who is monitoring events of the L1 zkEVM smart contract realizes that a new batch is consolidated and stores this information in the node's StateDB.

Remark About Building the Software

- Each component in a box can be instantiated in an isolated executable.
- The State library is imported by components (is not instantiated).
- While the JSON RPC is devoted to external communication, the component internal communication takes place using two types of interfaces:
    - gRPC APIs.
    - Postgre APIs with the databases.
- However, many components are built together into a single executable that can be configured as desired when started.
- Finally, remark that the databases can be built as databases in a single Postgre server or they can be split as desired in multiple PostgreSQL servers.

Outline

Architecture of the zkEVM: Prover and Executor



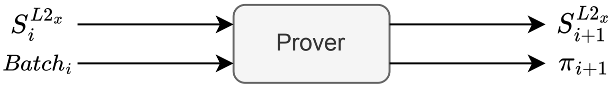
List of To Be Covered Concepts

- Provers. ☐
- Execution trace. ☐
- Witness and fixed columns. ☐
- Executors (general purpose and computation-specific). ☐
- zk Assembly. ☐
- ROM of the zkEVM. ☐
- forkId. ☐
- PIL (Polynomial Identity Language). ☐
- PIL2 (WIP). ☐
- Publics and privates. ☐
- Verifiers (Fflonk). ☐
- Selector columns. ☐
- zkEVM compatibility/equivalence types. ☐
- Secondary execution matrices A.K.A state machines. ☐
- PIL namespaces. ☐
- State machine interconnection with lookups. ☐

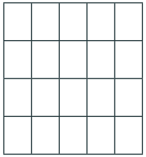
Functions of the L2 Prover

Prover

The ”Prover” is a component whose main goal is to generate a **proof** that for the correct execution of a given program with an specific set of inputs. The proving process is a resource-consuming process.



- To generate such a proof, we first need to create an **execution trace**.
- An execution trace is just a **matrix** (or grid) of cells with rows and columns.



Execution Trace Example i

- Let $x = (x_0, x_1, x_2)$ a vector of given inputs.
- We want to implement an execution trace for the following computation:

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

- Suppose that we only have the following operations available:
    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 (`ADD`).
    3. Multiply by a constant, and leave the result in a cell of the next row (`TIMES4`).
    4. Multiply two cells of the same row, and leave the result in a cell of the next row (`MUL`).

Execution Trace Example ii

- Let’s consider that our execution trace has 3 columns and a bounded number of rows (so that we can fit the needed computation in it):

A	B	C
a_0	b_0	c_0
a_1	b_1	c_1
...	...	...
a_n	b_n	c_n

- The columns of an execution trace are often called **registers** (so we may name them interchangeably).

Execution Trace Example iii

- Suppose we are given the following inputs $x = (x_0, x_1, x_2) = (1, 2, 5)$.
- We can model our desired computation $[(x_0 + x_1) \cdot 4] \cdot x_2$ to fit our execution trace using only the available operations as follows:

A	B	C		
1	2		$[a_0 = x_0, b_0 = x_1]$	ADD
3		4		TIMES4
12	5		$[b_2 = x_2]$	MUL
60				

Witness and Fixed Columns

- Notice that if we change the inputs $x = (x_0, x_1, x_2) = (5, 3, 2)$, we can perform exactly the same computation as before but the execution trace changes (most of) its values **but not its shape**.

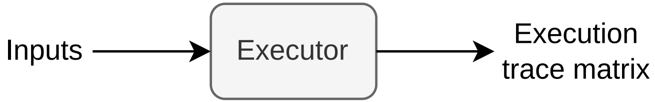
A	B	C
5	3	
8		4
32	2	
64		

- The columns that depend on the input (in this case A and B) are called **witness columns**.
- The columns that are the same for all inputs (in this case column C) are known as **fixed columns** (which we will mark in gray color).
- Note that fixed columns don't change as they are an intrinsic part of the computation.

Functions of the Executor

Executor

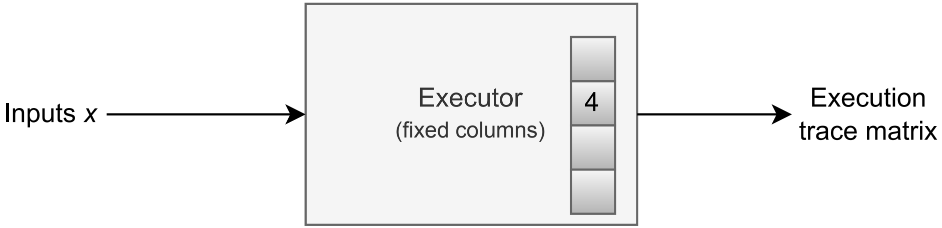
The ”Executor” is a component whose main purpose is to generate a (correct) execution trace from a given set of inputs.



How to implement the Executor i

Approach #1:

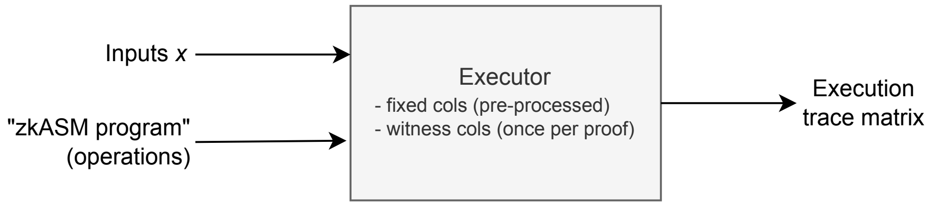
- As a component that runs just one given computation.
- Application Specific Integrated Circuit (ASIC) as electronic analogy: an ASIC is a circuit specifically designed to run, very efficiently, a single computation.
- We also call this executor a **native executor**.



How to implement the Executor ii

Approach #2:

- As a general purpose processor, which means as a component that can run several computations or ”programs”.
- In this case, the executor is as follows:


Inputs *x*
"zkASM program" (operations)
Executor
- fixed cols (pre-processed)
- witness cols (once per proof)
Execution trace matrix

Example of an Executor that Reads Assembly Programs

Using an executor that reads assembly, we could run two zkASM programs:

Program 1: $[(x_0 + x_1) \cdot 4] \cdot x_2$ having $x = (x_0, x_1, x_2) = (1, 2, 5)$ as inputs.

A	B	C		
1	2		$[a_0 = x_0, b_0 = x_1]$	`ADD`
3		4		`TIMES4`
12	5		$[b_2 = x_2]$	`MUL`
60				

Program 2 $(x_0 \cdot 16) \cdot x_1$ having $x = (x_0, x_1) = (2, 3)$ as inputs.

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

`https://github.com/0xPolygonHermez/zkevm-rom`

Single-computation vs General-computation Executor

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

- The single-computation executor is faster because it does not need to read assembly, it can implement the generation of the execution trace for the computation and this process can be optimized for this computation.
- However, the single-computation executor is not easy to change, test or audit.
- In zkEVM we will have both, each one serving different purposes.
- The single-computation executor is WIP.

zkASM: Assembly Language for the zkEVM

- zkASM is the language developed by the team that is used to write the program that a compiler will build and the executor will interpret in order to build the execution trace.

```
STEP => A
0                                   :ASSERT ; Ensure it is the beginning of the execution

CTX                                 :MSTORE(forkID)
CTX - %FORK_ID                      :JMPNZ(failAssert)

B                                   :MSTORE(oldStateRoot)
```

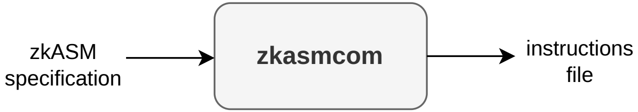
zkASM Language Example.

- In the repository `zkasmcom-vscode` there is a syntax highlighter for VSCode.

The zkASM Compiler

zkASM Compiler

We have implemented a zkASM compiler 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.



ROM-based Executor

- We need a **general-computation** executor because:
    - Our implementation of the EVM evolves.
    - The EVM itself also evolves.
- An architecture with assembly programs is faster to develop, test and audit than a specific implementation.
- We call the Ethereum program that processes EVM transactions the **EVM ROM** (Read Only Memory) or simply the **ROM**.

`forkId`

- By changing the ROM, we make our L2 zkEVM more and more closer to the L1 EVM.
- So we have versions of the zkEVM ROM.
- Each of these versions will be denoted with an identifier called `forkId`.
- Another advantage of using a ROM-based approach is that we can test small parts of the assembly program in isolation.
- Finally, mention that:
    - We are also developing a native executor (we will see why later).
    - Having the two approaches allows us to check that execution traces generated match.

List of To Be Covered Concepts

- Provers. ☑
- Execution trace. ☑
- Witness and fixed columns. ☑
- Executors (general purpose and computation-specific). ☑
- zk Assembly. ☑
- ROM of the zkEVM. ☑
- forkId. ☑
- PIL (Polynomial Identity Language). ☐
- PIL2 (WIP). ☐
- Publics and privates. ☐
- Verifiers (Fflonk). ☐
- Selector columns. ☐
- zkEVM compatibility/equivalence types. ☐
- Secondary execution matrices A.K.A state machines. ☐
- PIL namespaces. ☐
- State machine interconnection with lookups. ☐

Execution Correctness

The execution correctness is enforced by a set of constraints that must be fulfilled by the execution trace:

Program (computation):

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

A	B	C
1	2	
3		4
12	5	
60		

Constraints:

$$a_0 = x_0$$

$$b_0 = x_1$$

$$a_1 = a_0 + b_0$$

$$a_2 = a_1 \cdot 4$$

$$b_2 = x_2$$

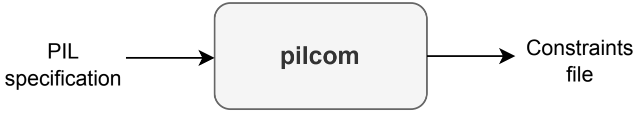
$$a_3 = a_2 \cdot b_2$$

The PIL Language and its Compiler

- In our cryptographic backend:
  - Each column is transformed into a polynomial (of the degree the number of rows).
  - Constraints are defined over these polynomials.
  - We describe constraints using a language called **PIL (Polynomial Identity Language)**.

PIL Compiler

We have implemented 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.



- The repository `pilcom-vscode` contains a PIL syntax highlighter for VSCode.

Publics and Privates i

- With zk-technology, we can create execution traces where some of the inputs are **private**.

Example 1

A	B	C
🟩 1	🟩 2	⬜
3		⬜ 4
12	🟩 5	⬜
🟩 60		⬜

Example 2

A	B	C
🟩 1	🟧 2	⬜
3		⬜ 4
12	🟩 5	⬜
🟩 60		⬜

🟩 : publics

🟧 : privates

⬜ : fixed

- Recall that columns A and B are **witness** while the C column is **fixed**.

Publics and Privates ii

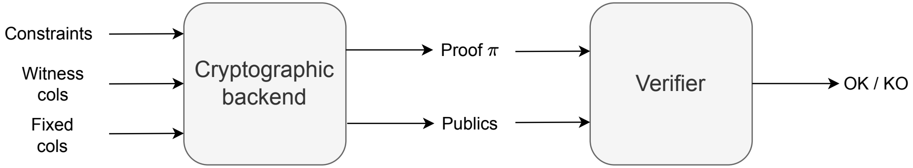
A	B	C
1	2	
3		4
12	5	
60		

- **Public inputs**: $\{1, 5\}$
- **Private inputs**: $\{2\}$
- **Output (public)**: $\{60\}$
- **Publics**: $\{1, 5, 60\}$

In this execution trace design, input x_1 is private, while inputs x_0 and x_2 are public and the output is also public.

Publics are enforced by constraints.

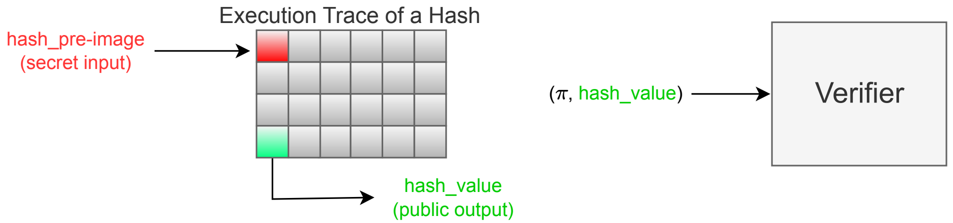
Generating and Verifying Proofs



- With a valid proof π, the verifier is convinced that the execution of the computation in question is correct for the given public inputs.
- π is small and needs a small amount of resources to be validated.
- In our case, currently we use a backend cryptographic system whose final verifier is **FFlonk**.
- **Note**: A smart contract in the L1 execution layer can verify the proof implementing a FFlonk verifier (and therefore validate the computation of a new state from a batch) with $\approx 200K$ gas.

Example of Usage of a Private Input

A typical example of using a private input is to prove the knowledge of the pre-image of a hash without revealing this pre-image value:



Shaping Execution Traces

- In an execution trace, each row is in charge of validating an **zkASM operation** or **part of an operation**.
- For example, suppose that we are given a set of 3 operations `OP1`, `OP2` and `OP3`.
- These operations change the next[1] value of the **A** column.

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

- Consider also that we have an execution trace matrix of 6 columns.

A	B	C	D	E	F

- **Q.** Can we fit this computation inside the matrix? how?

[1]Primes mean next value of some column, e.g. a' means the next value in the **A** column.

Shaping Execution Traces: Strategies

- We can adopt two straightforward strategies:
  a) Increasing the number of columns so that we can fit every summand.
  b) Use the next row in order to fit some of the remaining summands of the operation.

- Using the second approach, we can define an execution matrix in which OP3 uses two rows:

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

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

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

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				

Execution Trace Design Strategies i

Number of rows and columns?

- The maximum number of rows might be fixed by a cryptographic back-end (this is the case of our current back-end).
- The fewer rows used, the faster the prover is.
- In practice, $\#rows = 2^n$ for some natural $n \in \mathbb{N}$.
- More or less columns? Adding columns also increases proving time.

Execution Trace Design Strategies ii

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

OP1: $a' = a + b + c$

OP2: $a' = a + b + c + d + e$

6 columns and 3 rows.

18 cells but 9 of them unused.

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

OP1: $a' = a + b + c$

OP2: $c' = a + b + c + a' + b'$

3 columns and 3 rows.

9 cells and all of them used.

#unused_cells depends on the instructions executed and the execution matrix shape.

Selector Columns

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

OP1: $a + b + c - a' = 0$
OP2: $a + b + c + a' + b' - c' = 0$

- Since we are using constraints with columns (not cells), we need to add **selector columns**.
- Selector columns are used to control wether the constraints apply or not (meaning whether we are performing this or that operation).

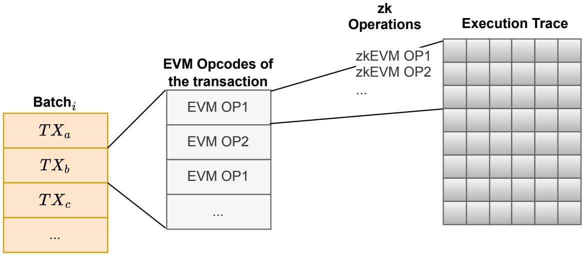
	A	B	C	OP1	OP2
OP1	a_0	b_0	c_0	1	0
OP2	a_1	b_1	c_1	0	1
	a_2	b_2	c_2		

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

The Execution Trace and the zkEVM

- In our current cryptographic backend, we have a shape that is pre-fixed for the execution trace.
- Also, we don't 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.
- 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:

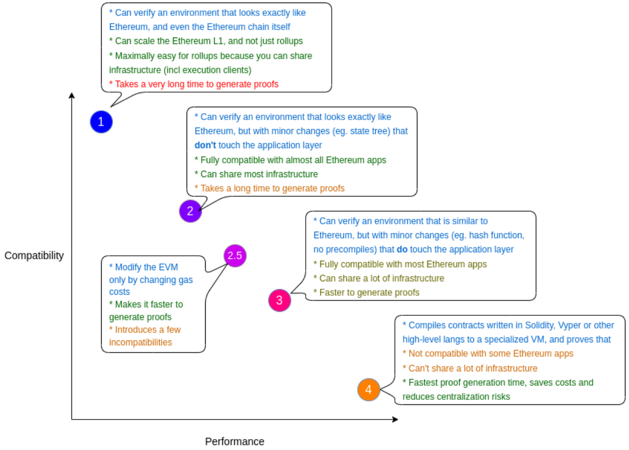
1. **Narrow matrices may easily hit the max row limit**, which is about 2^{23} (fixed by the cryptographic back-end).
2. **Wide matrices might be inefficient** for mixing many different instructions.



zkEVMs Compatibility/Equivalence i

- A layer 2 is EVM compatible or equivalent if it can run EVM byte code without modifying the underlying smart contract logic.
- EVM compatibility allow L2's 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.
- **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 ZK 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 zk, resulting in faster prover times but potentially introducing incompatibilities and limitations.

zkEVMs Compatibility/Equivalence ii


* Can verify an environment that looks exactly like Ethereum, and even the Ethereum chain itself
* Can scale the Ethereum L1, and not just rollups
* Maximally easy for rollups because you can share infrastructure (incl execution clients)
* Takes a very long time to generate proofs
1
* Can verify an environment that looks exactly like Ethereum, but with minor changes (eg. state tree) that **don't** touch the application layer
* Fully compatible with almost all Ethereum apps
* Can share most infrastructure
* Takes a long time to generate proofs
2
Compatibility
* Modify the EVM only by changing gas costs
* Makes it faster to generate proofs
* Introduces a few incompatibilities
2.5
* Can verify an environment that is similar to Ethereum, but with minor changes (eg. hash function, no precompiles) that **do** touch the application layer
* Fully compatible with most Ethereum apps
* Can share a lot of infrastructure
* Faster to generate proofs
3
* Compiles contracts written in Solidity, Vyper or other high-level langs to a specialized VM, and proves that
* Not compatible with some Ethereum apps
* Can't share a lot of infrastructure
* Fastest proof generation time, saves costs and reduces centralization risks
4
Performance

https://vitalik.ca/general/2022/08/04/zkevm.html

Motivational Example: Implementing an EXP Operation

- Let's assume that our cryptographic backend only allows to define constraints with additions and multiplications.
- Let's consider also that we want to implement a exponentiation operation (EXP).
- Then, we can use several rows doing multiplications to implement EXP.
- A portion of the execution trace (that implements $2^5 = 32$) could be:

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

An incomplete (**uncorrected**) set of constraints:

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)

- If we want to implement this operation in the main Execution Trace, note that we are going to spend several rows per EXP operation.
- In fact, a variable number of rows per EXP operation depending on the exponent.

Secondary Execution Trace Matrices and Lookup Arguments

- The previous approach leads to a very complicated set of constraints together with a huge amount of consumed rows by operations, which is quite an unwanted scenario.
- Another approach is to use **tailor-made secondary execution traces for specific operation(s)**:
    - 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.
- So, another approach for our `EXP` operation is to implement it in a secondary execution trace matrix and link the main execution trace with the secondary trace with a **lookup argument**.

Main Trace with Delegated Operation checks

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

- In the main execution trace, each EXP operation occupies just one row.
- Notice that we introduced the **EXP** selector to indicate when the EXP operation is being performed.
- For the first EXP operation, inputs are 2 and 5 and the result is 32.
- The correctness of the result will be validated in a secondary execution matrix.

State Machine (SM) Concept and Lookup Between SMs

- 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.

Main state machine

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

🟨 : Lookup

EXP 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

- Constraints in the secondary SM enforce the correctness of the **EXP** operation.
- In the main SM, we just put the inputs/outputs, that we call *”free”*, in a single row.

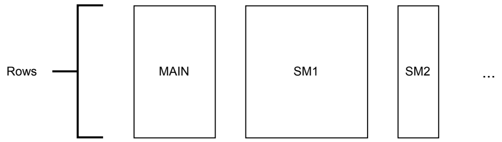
The PIL of the zkEVM

- 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.
- In the PIL language, the state machines (subexecution matrices) are called `namespaces`.
- In the zkevm-proverjs repository, you can find the PIL specification of the zkEVM under the pil directory.

Remarks about the Computation and the Shape of SMs

- The columns of each state machine are defined by the design of its corresponding execution trace.
- Due to limitations of our current cryptographic backend, all the SM must have 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.
- As a result, the number of rows used at each SM depends on the number of operations of each type during the batch execution.

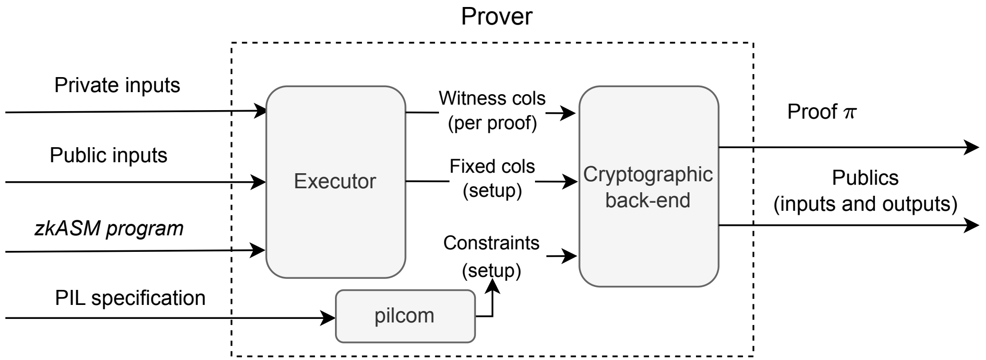
- Since the number of rows is fixed (and the same for all State Machines) we can have *unused* rows.
- But, what is more important is that obviously, **the size of the computation being proved must fit in the execution trace matrices available**.



PIL Evolution: PIL2 (WIP)

- Currently, we are under the development of a new version of PIL called **PIL2**.
- PIL2 is designed to operate with a more powerful cryptographic backend that is able to generate as many subexecution traces as required by the batch processing so that we never run out of rows.
- We are also agreeing with the rest of the ”zk projects” at Polygon a format for the PIL output file called ”pilout”.

Recap of the Prover



Note. The files containing the pre-processed fixed columns and the processed witness columns for the zkEVM are temporary stored in binary files and are quite large (>100Gb).

Recap of the Verifier

- In our previous example:

A	B	C
1	2	
3		4
12	5	
60		

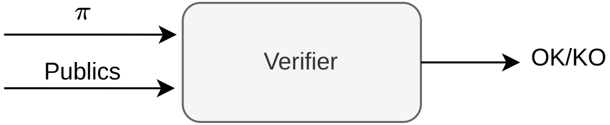
🟩 : publics

🟧 : privates

⬜ : fixed

- **Public inputs**: {1, 5}
- **Private inputs**: {2}
- **Output** (public): {60}
- **Publics**: {1, 5, 60}

- Then the verifier:

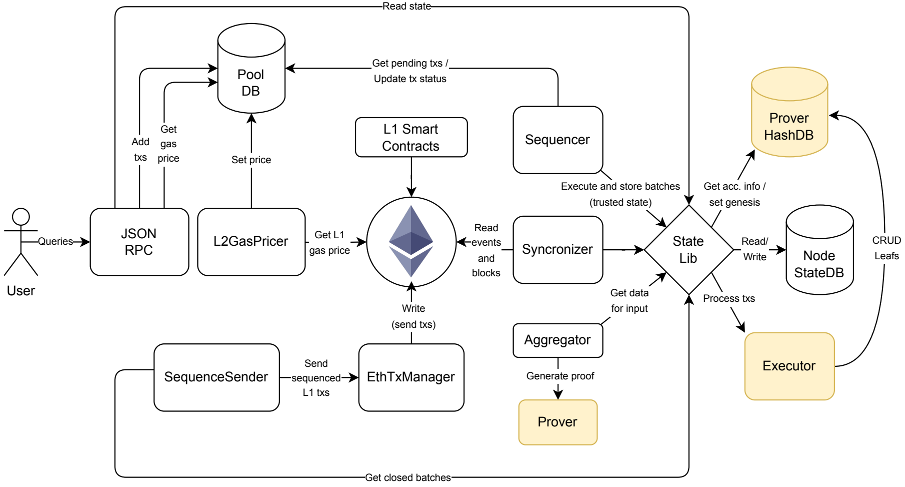


List of Covered Concepts

- Provers. ☑
- Execution trace. ☑
- Witness and fixed columns. ☑
- Executors (general purpose and computation-specific). ☑
- zk Assembly. ☑
- ROM of the zkEVM. ☑
- forkId. ☑
- PIL (Polynomial Identity Language). ☑
- PIL2 (WIP). ☑
- Publics and privates. ☑
- Verifiers (Fflonk). ☑
- Selector columns. ☑
- zkEVM compatibility/equivalence types. ☑
- Secondary execution matrices A.K.A state machines. ☑
- PIL namespaces. ☑
- State machine interconnection with lookups. ☑

Outline

Architecture of the zkEVM



List of To Be Covered Concepts

- L2 state CRUD operations. ☐
- zkEVM binary SMT. ☐
- Storage state machine. ☐
- Revisit the HashDB. ☐

Updating the State

- Recall that the **zkEVM ROM** is **zkASM program** designed to prove the computations for stating correct L2 State transitions given a batch of L2 transactions.
- We might use several zkASM instructions to implement a single zkEVM opcode (in fact, this this happens most of the times).
- Recall that the L2 State:
  1. Is stored as a Merkle Tree.
  2. The root (called **State Root**) of the tree is used as a cryptographic summary of the current state data.
- Hence, the **ROM** needs to have a way to **correctly** perform CRUD (Create, Read, Update and Delete) operations on the Merkle Tree representing the current state.

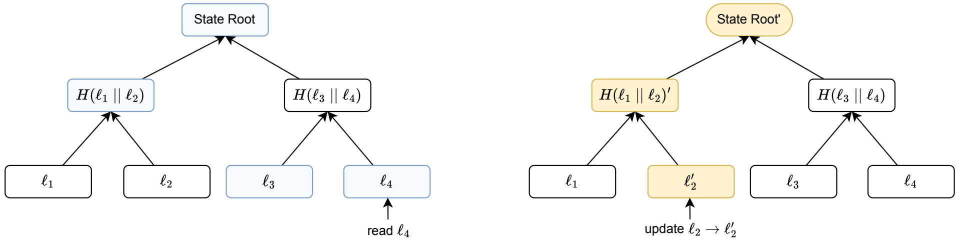
Storage State Machine i

In the zkEVM, we have implemented a secondary State Machine called the **Storage State Machine** that is devoted to generate the execution trace that **proves L2 data state creation, read, update and delete**:

- Each time we need to read a value from the tree it is required to obtain a set of nodes of the tree called the Merkle proof to assure that the read state value is correct.
- Each operation that modifies the tree requires to proof that the tree modification is correctly performed.

After processing the last L2 transaction of the batch, the remaining root will be the new state root.

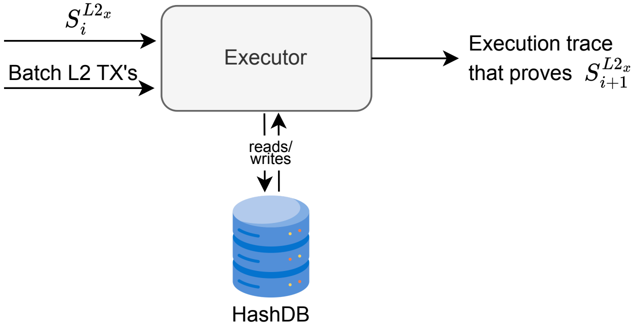
Storage State Machine ii


State Root
$H(\ell_1 \,||\, \ell_2)$
$H(\ell_3 \,||\, \ell_4)$
ℓ_1
ℓ_2
ℓ_3
ℓ_4
read ℓ_4
State Root'
$H(\ell_1 \,||\, \ell_2)'$
$H(\ell_3 \,||\, \ell_4)$
ℓ_1
ℓ_2'
ℓ_3
ℓ_4
update $\ell_2 \to \ell_2'$

- In the previous example, we are reading the fourth leaf (ℓ_4) and updating the second leaf (ℓ_2) of the Merkle tree.
- **Remark.** The Merkle tree of the Polygon zkEVM is a binary Sparse Merkle Tree (SMT) and it differs from the Merkle tree of the L1 EVM, which is a trie, in particular, a Patricia tree.

Storing the State: HashDB

- All the hashes (nodes) of the Merkle tree of the L2 state are stored in a database called **prover HashDB** or **HashDB** for short.
- The executor, when performing operations of the Storage State Machine, needs to read and write the HashDB to create appropriate execution trace that proves the L2 state reads and writes.

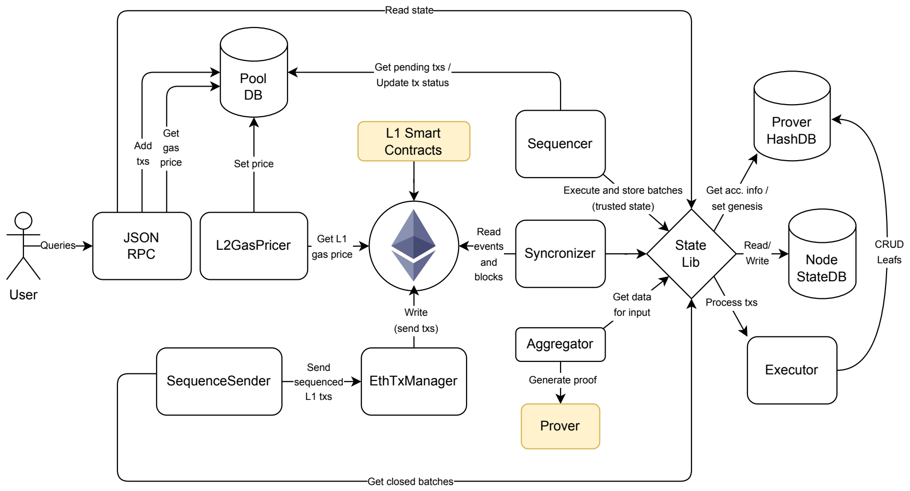


List of Covered Concepts

- L2 state CRUD operations. ☑
- zkEVM binary SMT. ☑
- Storage state machine. ☑
- Revisited the HashDB. ☑

Outline

Architecture of the zkEVM



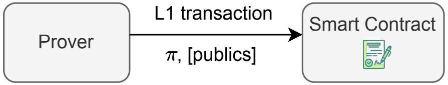
List of To Be Covered Concepts

- zkEVM and Privacy. ☐
- `batchData` public input. ☐
- `currentStateRoot` public input. ☐
- `proverAccount` public input. ☐
- `timestamp` public input. ☐
- `forkId` public input. ☐
- `chainId` public input. ☐
- `newStateRoot` public output. ☐

zkEVM and Privacy

- In our setting, the zkEVM, we are not concerned with privacy, only with **succinct computation verification**.
- In this sense, L2 transactions and L2 state data are public.
- So, let's start with an initial design of the proving system **without private inputs**.

Proof's publics



- Public inputs:
  1. `batchData`: L2 transactions in the batch.
  2. `currentStateRoot`: Current L2 state root.
- More public inputs:
  1. `proverAccount`: to receive rewards.
  2. `timestamp`.
  3. `forkId`: L2 EVM version.
  4. `chainId`: for being able to host multiple L2 networks.
- (Public) outputs:
  1. `newStateRoot`: New L2 state root.

- **Q**: Do we send all publics in the transaction to L1?
  A: Actually **no**, some data will already be in the storage of the smart contract.

Sending Proof's Publics to L1

- In particular the following publics are actually stored in the L1 smart contract:
    - `currentStateRoot`.
    - `forkId`.
    - `chainId`.
- Henceforth, in the **`calldata`** of the transaction sent to the L1's smart contract, we need to include the rest of inputs for the batch verification:
    - **Publics**:
        - `batchData`.
        - `timestamp`.
        - `newStateRoot`.
        - Note that the `proverAccount` is already in the L1 transaction's signature, so we need not provide it explicitly.
    - **The proof** of the correct execution of the L2 transactions within the batch.

List of Covered Concepts

- zkEVM and Privacy. ☑
- `batchData` public input. ☑
- `currentStateRoot` public input. ☑
- `proverAccount` public input. ☑
- `timestamp` public input. ☑
- `forkId` public input. ☑
- `chainId` public input. ☑
- `newStateRoot` public output. ☑