Cryptographic Pairings in a zkEVM

Héctor Masip Ardevol

Polygon zkEVM,
Universitat Politècnica de Catalunya (UPC)

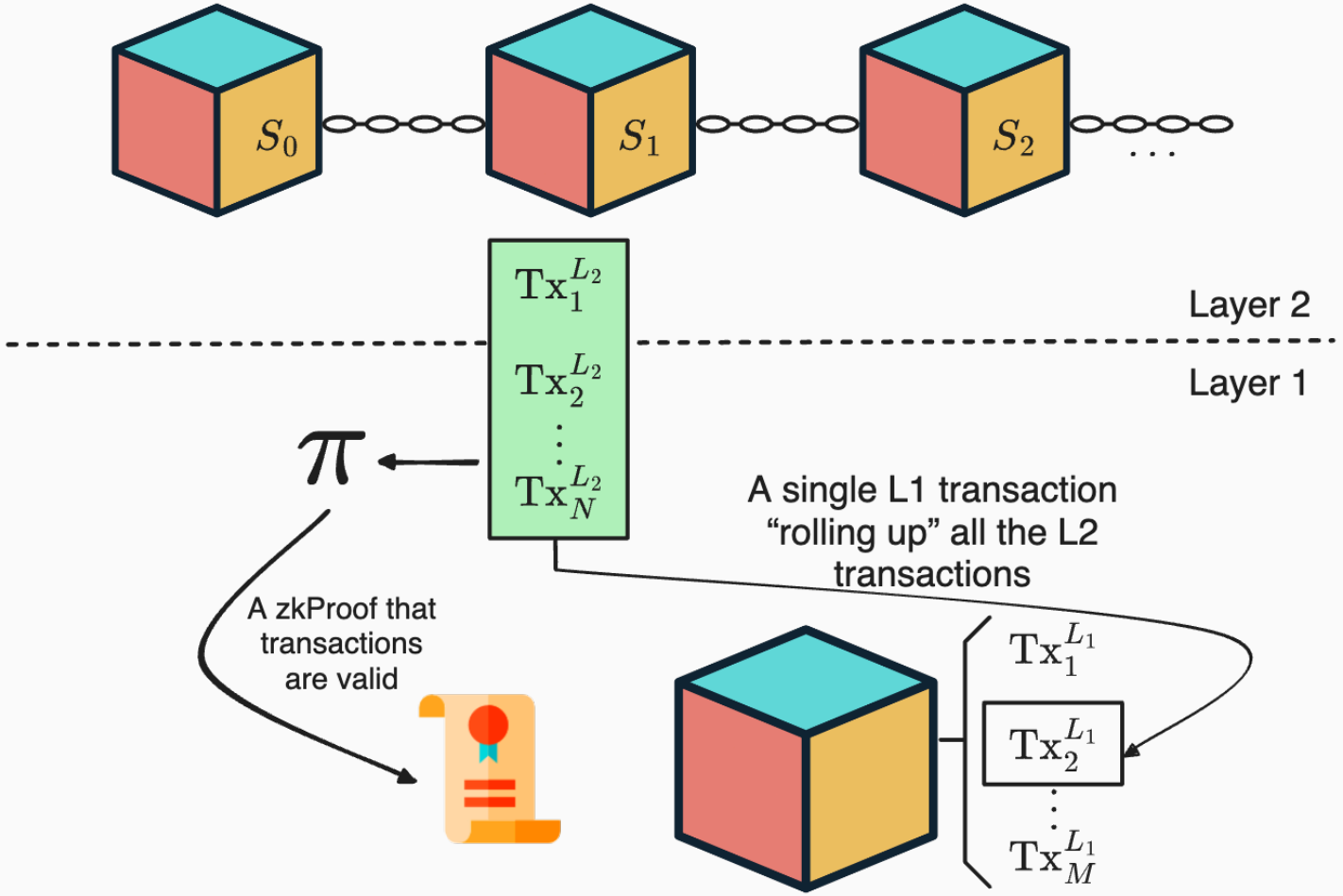
Table of Contents



1. zkEVM: The Scaling Solution
2. Pairings as a Cryptographic Primitive
3. zkEVM & Pairings: The Ultimate Union

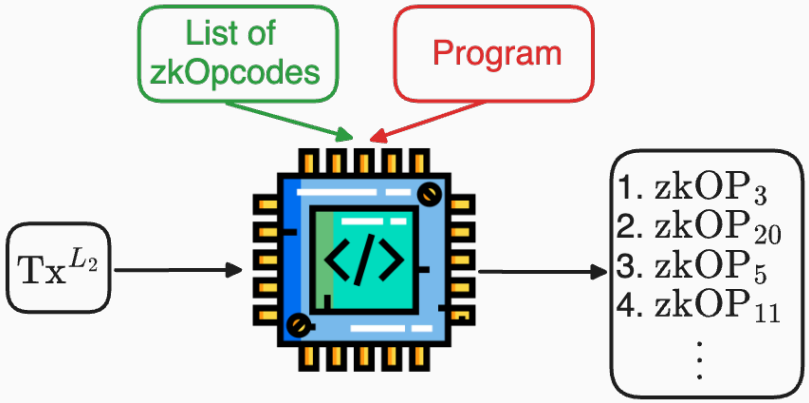
zkEVM: The Scaling Solution

What is a zkRollup?



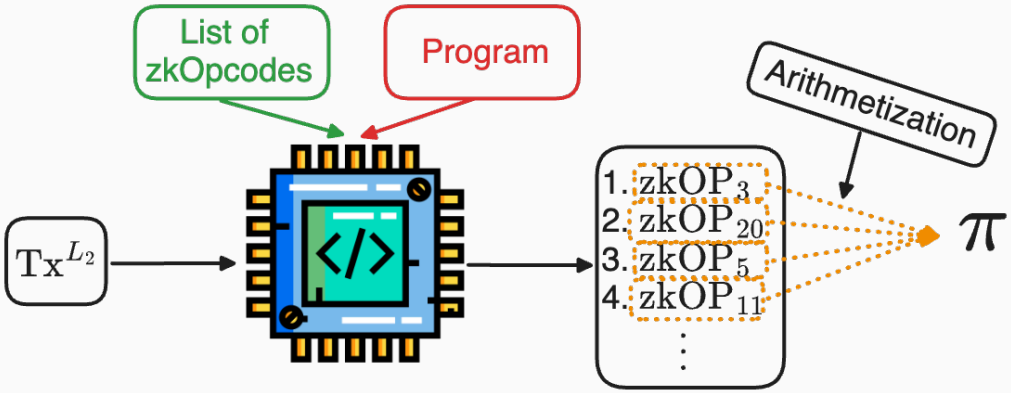
Proving State Transitions: the zkEVM

- zkRollups aim to be *fully EVM-equivalent*, which means that they are fully compatible with existing applications.
- However, there should be some desing differences to make them more amenable to *arithmetization* (e.g. using Poseidon instead of Keccak for storage).
- To get both, it makes sense to introduce a "zk-friendly EVM": **the zkEVM**.

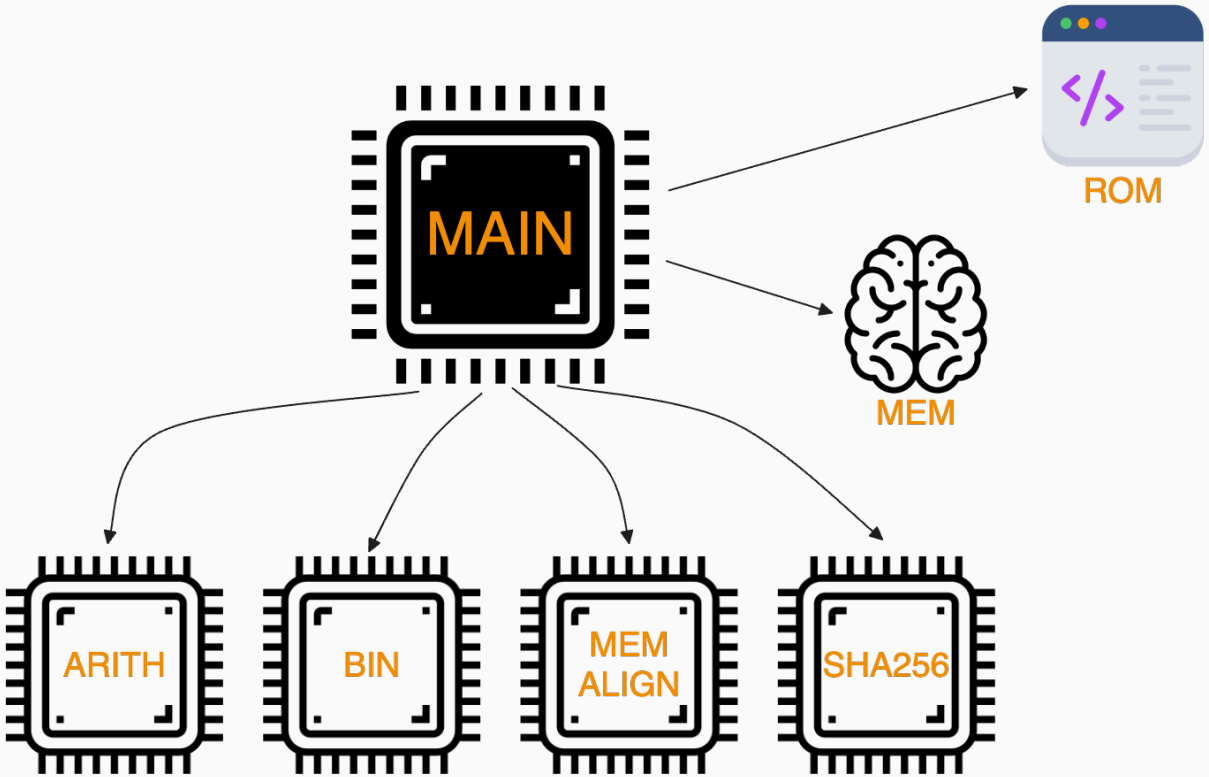


Proving State Transitions: the zkEVM

- zkRollups aim to be *fully EVM-equivalent*, which means that they are fully compatible with existing applications.
- However, there should be some desing differences to make them more amenable to *arithmetization* (e.g. using Poseidon instead of Keccak for storage).
- To get both, it makes sense to introduce a "zk-friendly EVM": **the zkEVM**.



The Rollup's State Transtion has been Performed Correctly



Specifications of the zkEVM

- It only contains 5 generic-purpose registers `A`,`B`,`C`,`D`,`E`.
- Works with unsigned integers of 256 bits, represented by 8 chunks of 32 bits:

$$A = [A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7]$$

- Each chunk is represented by an element in the field of order $p = 2^{64} - 2^{32} + 1$.
- It has some "small" registers like `SP`, `GAS`, `HASHPOS`...
- Very limited set of opcodes: `MLOAD`, `MSTORE`, `SLOAD`, `SSTORE`, `ASSERT`, `HASHX`, `ARITH`, `BIN`, `JMP`, `JMP_CND`, `CALL`, `RETURN`...

The ROM and zkASM

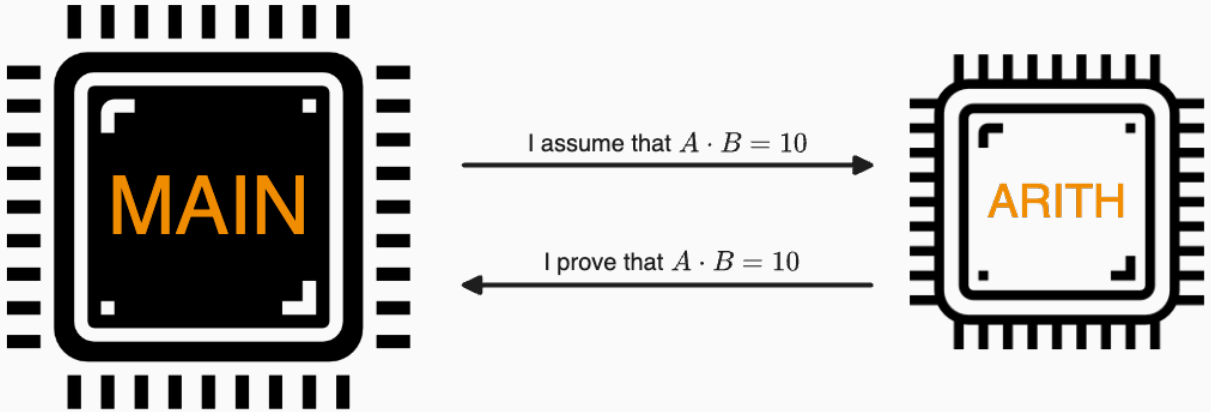
- The ROM is the part of the zkEVM that encodes its logic. It's **program**.
- In particular, EVM's opcodes are broken down into zkEVM's opcodes with **zkASM**:

```
opDUP1:
    $ => A          :MLOAD(SP-1)
    A               :MSTORE(SP++)
```

```
opADD:
    SP - 1 => SP
    $ => A          :MLOAD(SP--)
    $ => B          :MLOAD(SP)
    $               :ADD, MSTORE(SP++)
```

- Hence, **proving each state transition translates to proving each zkEVM's opcode**.

- **Polynomial Identity Language (PIL)** is the part of zkEVM that encodes its arithmetization.



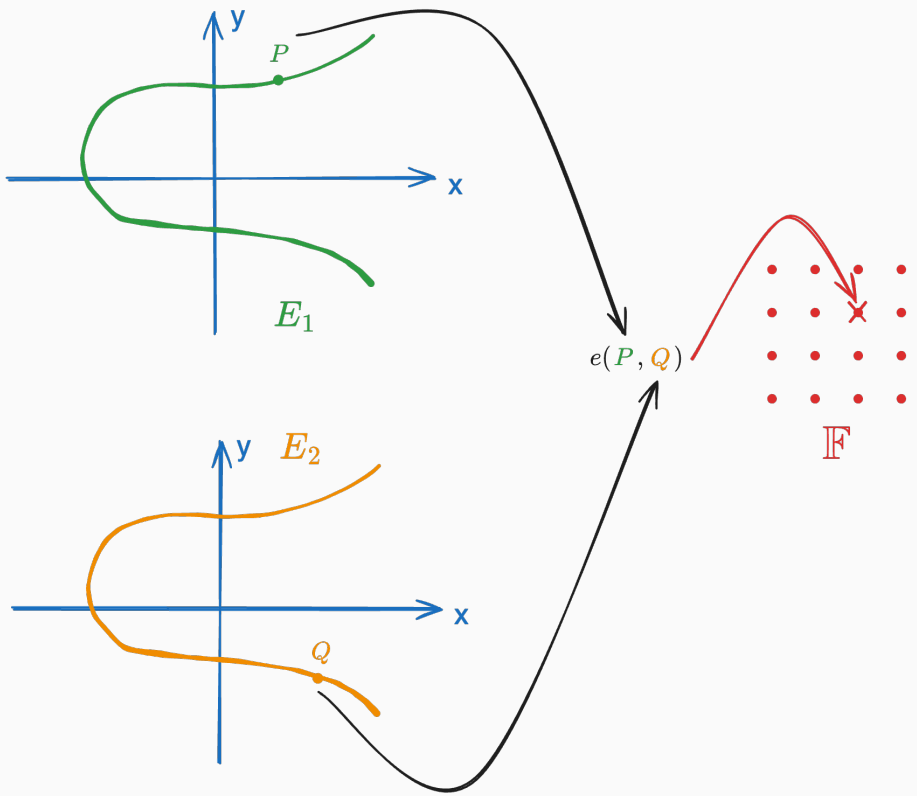

```
{arithEq, A, B, op} is {Arith.selEq, Arith.x, Arith.y, Arith.z}
```

- In particular, zkEVM's opcodes are related to one ore more constraints in the PIL description that ensure its correctness.

Pairings as a Cryptographic Primitive

What is a Pairing?

- A **pairing** is defined as the mapping of two *elliptic curve* points $P \in E_1, Q \in E_2$ to an element in a *finite field* $e(P, Q) \in \mathbb{F}$:



What is a Pairing?

- The signature of this pairing is of the form:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

  where:

  - $\mathbb{G}_1 = E(\mathbb{F}_p)[r]$ is the only subgroup of the r-torsion of E over $\mathbb{F}_p$.
  - $\mathbb{G}_2 \subset E(\mathbb{F}_{p^{12}})[r]$, represented by a **twist** $E'(\mathbb{F}_{p^2})[r]$, is the trace-zero subgroup of the r-torsion of E over $\mathbb{F}_{p^{12}}$.
  - $\mathbb{G}_T = \mu_r$ is the set of r-th roots of unity over $\mathbb{F}_{p^{12}}$.

Pairings are Bilinear Maps

- What makes pairings so useful for cryptography is that they are **bilinear maps**:
  1. $e(P + R, Q) = e(P, Q) \cdot e(R, Q)$
  2. $e(P, Q + R) = e(P, Q) \cdot e(P, R)$
  3. $e([a]P, [b]Q) = e(P, [b]Q)^a = e([a]P, Q)^b = e([b]P, [a]Q) = e(P, Q)^{ab}$

  where P, Q, R are points on the appropriate elliptic curves and $a, b \in \mathbb{Z}$.
- In other words, we can "move around" the coefficients a and b on those two curves while keeping the mapping the same.

Applications of Pairings

- Bilinearity allows “verifying” algebraic relations between a and b:
  1. Checking whether $e([a]P, [b]Q) = e(P, Q) \neq 1$ implies $b \equiv a^{-1} \bmod r$.[1]
  1. BLS digital signature verification:

$$e(\sigma, G_2) = e(H(m), \mathrm{pk})$$

  1. Polynomial opening verification à la KZG:

$$e\left(\llbracket p(x) \rrbracket_1 - \llbracket p(z) \rrbracket_1, \llbracket 1 \rrbracket_2\right) \cdot e\left(-\llbracket q(x) \rrbracket_1, \llbracket x \rrbracket_2 - \llbracket z \rrbracket_2\right) = 1$$

$$\implies \frac{p(x) - p(z)}{x - z} = q(x)$$

[1] Related to the DDH

Applications of Pairings

- Bilinearity allows "verifying" algebraic relations between a and b:
  1. Checking whether $e([a]P, [b]Q) = e(P, Q) \neq 1$ implies $b \equiv a^{-1} \bmod r$.[2]
  2. BLS digital signature verification:

$$e(\sigma, G_2) = e(H(m), \mathrm{pk})$$

  3. Polynomial opening verification à la KZG:

$$e\Big(\llbracket p(x) \rrbracket_1 - \llbracket p(z) \rrbracket_1, \llbracket 1 \rrbracket_2\Big) \cdot e\Big(-\llbracket q(x) \rrbracket_1, \llbracket x \rrbracket_2 - \llbracket z \rrbracket_2\Big) = 1$$

$$\implies \frac{p(x) - p(z)}{x - z} = q(x)$$

[2] Related to the DDH

zkEVM & Pairings: The Ultimate Union

What is Needed to Compute a Pairing

```
def e(P,Q):
    assert(subgroup_check_G1(P))
    assert(subgroup_check_G2(Q))

    # Miller Loop
    R = Q
    f = 1
    for i in range(len(x)-2,-1,-1): # x is the curve parameter
        f = f * f * line(untwist(R),untwist(R),P)
        R = 2 * R
        if (x[i] == 1):
            f = f * line(untwist(R),untwist(Q),P)
            R = R + Q

    return final_exponentiation(f)
```

What is Needed to Compute a Pairing

- Tower of extension fields arithmetic:

  $$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 - \beta), \quad \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi), \quad \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v)$$

- Elliptic curve arithmetic:

  $$E(\mathbb{F}_p) : y^2 = x^3 + b, \quad E'(\mathbb{F}_{p^2}) : y^2 = x^3 + b'$$

- The untwist and twist maps:

  $$\varphi(x', y') = (x'w^2, y'w^3), \quad \varphi^{-1}(x, y) = (x/w^2, y/w^3)$$

What is Needed to Compute a Pairing

- Tower of extension fields arithmetic:

$$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 - \beta), \quad \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi), \quad \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v)$$

- Elliptic curve arithmetic:

$$E(\mathbb{F}_p) : y^2 = x^3 + b, \quad E'(\mathbb{F}_{p^2}) : y^2 = x^3 + b'$$

- The untwist and twist maps:

$$\varphi(x', y') = (x'w^2, y'w^3), \quad \varphi^{-1}(x, y) = (x/w^2, y/w^3)$$

What is Needed to Compute a Pairing

- Tower of extension fields arithmetic:

$$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 - \beta), \quad \mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi), \quad \mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v)$$

- Elliptic curve arithmetic:

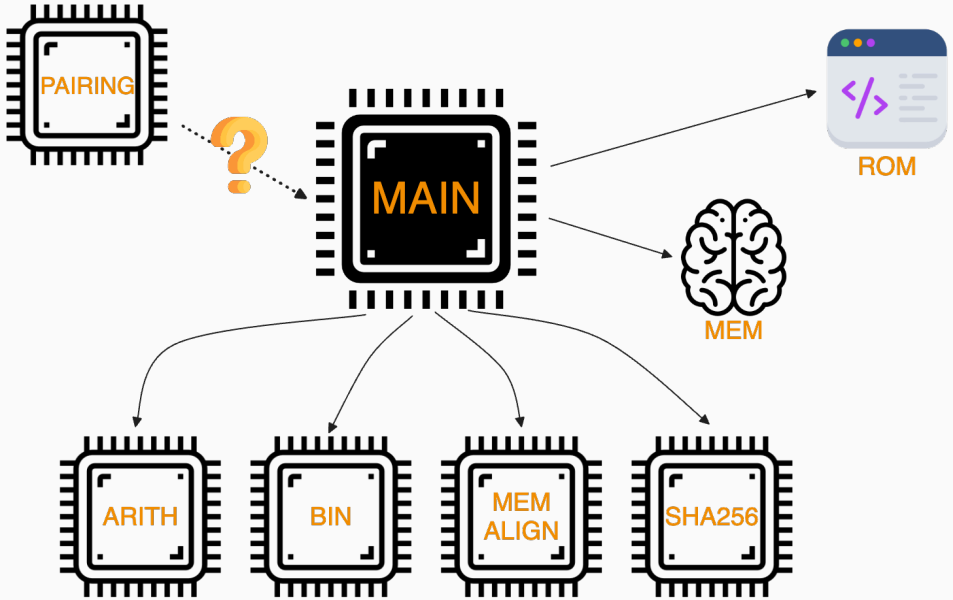
$$E(\mathbb{F}_p) : y^2 = x^3 + b, \quad E'(\mathbb{F}_{p^2}) : y^2 = x^3 + b'$$

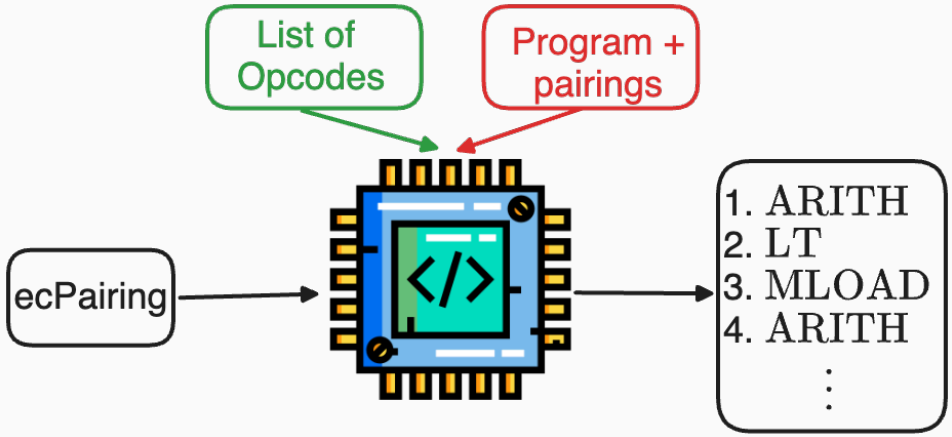
- The untwist and twist maps:

$$\varphi(x', y') = (x'w^2, y'w^3), \quad \varphi^{-1}(x, y) = (x/w^2, y/w^3)$$

Enlarging the zkEVM's PIL or Encoding

- A hard decision should be made:
  - **Precompile**: Enlarging zkEVM's PIL with the pairing arithmetization.
  - **ROM**: Breaking it down into zkEVM's opcodes.





Which one to Choose?

- Finding its PIL's arithmetization is *doable* but *challenging* and (largely) *inefficient*..
- In particular, the augmented proving time would not justify the overhead for something as infrequently used in Ethereum as pairings.
- On the other hand, expanding the zkEVM's program description (ROM) to include the pairing is more *intuitive*, and **the cost of the pairing is only incurred when utilized**.
- For sure, one should face the own's language(s) restrictions. See circom-pairing.

Which one to Choose?

- Finding its PIL's arithmetization is *doable* but *challenging* and (largely) *inefficient*..
- In particular, the augmented proving time would not justify the overhead for something as infrequently used in Ethereum as pairings.
- On the other hand, expanding the zkEVM's program description (ROM) to include the pairing is more *intuitive*, and **the cost of the pairing is only incurred when utilized**.
- For sure, one should face the own's language(s) restrictions. See circom-pairing.

BN254 vs BLS12-381

- Before pairings, we had some arithmetic opcodes:

$$\texttt{ARITH}: A \cdot B + C, \quad \underbrace{\texttt{ARITH_ECADD_DIFFERENT} \quad \texttt{ARITH_ECADD_SAME}}_{\text{Secp256k1 Signature Verification}}$$

- New opcodes have been introduced in order to handle with the curves:

BN254	BLS12-381
▸ ARITH_MOD: $(A \cdot B + C) \bmod D$ ▸ ARITH_BN254_ADDFP2: complex add ▸ ARITH_BN254_SUBFP2: complex sub ▸ ARITH_BN254_MULFP2: complex mul	▸ ARITH_256TO384: chunk conversion ▸ ARITH_384_MOD: $(A \cdot B + C) \bmod D$ ▸ ARITH_BLS12381_ADDFP2: complex add ▸ ARITH_BLS12381_SUBFP2: complex sub ▸ ARITH_BLS12381_MULFP2: complex mul

Finite Field Arithmetic

- Working over the BN254...

```
; A + C in Fp
addFpBN254:
  1 => B ; verify against op
  %BN254_P => D
  ${(A + C) % D}    :ARITH_MOD
```

```
; (A + C) + (B + D)·u in Fp²
addFp2BN254:
  ; verify against E + op·u
  ${(A + C) % P} => E
  ${(B + D) % P}  :ARITH_BN254_ADDFP2
```

- Is exactly the same as the BLS12381...

```
; A + C in Fp
addFpBLS12381:
  1 => B ; verify against op (mod D)
  %BLS12_381_P => D
  ${(A + C) % D}    :ARITH_384_MOD
```

```
; (A + C) + (B + D)·u in Fp²
addFp2BLS12381:
  ; verify against E + op·u
  ${(A + C) % P} => E
  ${(B + D) % P}  :ARITH_BLS12_ADDFP2
```

A More Complex Example

```
; in:    P = (P.x, P.y) ∈ E(Fp)
; out: 2·P = (Q.x, Q.y) ∈ E(Fp)
doublePointFp:
        doublePointFp_P_y => A  :CALL(dblFpBLS12381)
        ; C = 2·P.y

        B => A                  :CALL(invFpBLS12381)
        ; B = 1 / 2·P.y
        B => lambda

        3 => A                  :CALL(mulFpBLS12381)
        ; C = 3/2y

        doublePointFp_P_x => A
        C => B                  :CALL(mulFpBLS12381)
        ; C = 3x/2y

        doublePointFp_P_x => A
        C => B                  :CALL(mulFpBLS12381)
        ; C = lambda = 3x²/2y

        C => A,lambda           :CALL(squareFpBLS12381)
        ; B = lambda²

        B => A
        doublePointFp_P_x => C  :CALL(subFpBLS12381)
        ; C = lambda² - P.x

        C => A
        doublePointFp_P_x => C          :CALL(subFpBLS12381)
        ; C = lambda² - P.x - P.x

        C => doublePointFp_Q_x
        ; Q.x = lambda² - P.x - P.x

        doublePointFp_P_x => A          :CALL(subFpBLS12381)
        ; C = P.x - Q.x

        lambda => A
        C => B                          :CALL(mulFpBLS12381)
        ; C = lambda·(P.x - Q.x)

        C => A
        doublePointFp_P_y => C          :CALL(subFpBLS12381)
        ; C = lambda·(P.x - Q.x) - P.y

        C => doublePointFp_Q_y
```

Applications: Precompileds in the zkEVM

- The inclusion of pairings in a zkEVM allows proving the validity of the `ecAdd`, `ecMul`, `ecPairing`, `point_evaluation` precompiled.
- This has several benefits:
  - Advancing a zkEVM from Type 3 to Type 2 (see Vitalik's).
  - It allows for all types of transactions, enabling full Ethereum scalability.
  - Other projects can act as if you where Ethereum: L3s, Polygon's AggLayer,... even **L2's verifying their own proofs!**

Thanks for listening!