# POSEIDON

Zero-knowledge friendly hash function

# WHY IS IT NEEDED?

- In private transactions, we:
  - Sign a hash $h = H(k, m)$, where $k$ is a secret.
  - Add $h$ to the Merkle tree.
  - Then we have to prove that:
    - $h \in T$
    - $h = H(k, m)$
- Doing that with traditional hashing functions is very expensive.

# OTHER PROBLEMS

- Traditional hashing functions are often optimized for specific CPU architectures. ZK projects often use virtual architectures (e.g. zkEVM, WASM).

# WHAT DO WE NEED?

- Work natively with *GF(p)* (prime field) objects.
- Consideration of circuit properties.
  - Degrees.
  - Size of circuit.

# STATE

An array of the given width which initially consists of the following prime field elements:

- Domain tag - used for differentiating between multiple hasher instances in the same program. By default, 0.
- Inputs to be hashed.
- Optional padding.

# STATE

- Width has 1 more element than inputs (including padding).
- If we declare width as $w$, number of inputs (w/o padding) as $n$, and number of padding elements as $m$, then:
  - $w = 1 + n + m$

# STATE

| 0 | 1 | ... | 1 + n - 1 | 1 + n | ... | 1 + n + m |
|---|---|---|---|---|---|---|
| Domain tag | Input 0 | ... | Input n | Padding 0 | ... | Padding m |

# DOMAIN TAG

- By default, 0.
- Can be set to a different value if we want to differentiate between multiple hasher instances in the same program.

# POSEIDON HASHER WITH STATE

```
use ark_ff::PrimeField;

pub struct Poseidon<F: PrimeField> {
    params: PoseidonParameters<F>, // we will get to them late
    domain_tag: F,
    state: Vec<F>,
}
```

# INITIALIZATION (OF HASHER)

```
pub fn new(params: PoseidonParameters<F>, domain_tag: Option<F
    let domain_tag = domain_tag.unwrap_or_else(F::zero);
    let width = params.width;
    Self {
        domain_tag,
        params,
        state: Vec::with_capacity(width),
    }
}
```

# INITIAL STATE (WHEN STARTING A SINGLE HASHING OPERATION)

```
fn hash(&mut self, inputs: &[F]) -> F {
    assert!(inputs.len() == self.params.width - 1);

    self.state.push(self.domain_tag);
    for input in inputs {
        self.state.push(*input);
    }

    [...]
}
```

# ADD ROUND CONSTANTS

- Denoted by *ARC*.
- In each round, constants which are added to all elements of the state:
  - $state = state \oplus ARC$

# ADD ROUND CONSTANTS

```
pub struct PoseidonParameters<F: PrimeField> {
    pub ark: Vec<F>, // size: rounds * width
    [...]
}

fn apply_ark(&mut self, round: usize) {
    self.state.iter_mut().enumerate().for_each(|(i, a)| {
        let c = self.params.ark[round * self.params.width + i]
        *a += c;
    });
}
```

# MDS MATRIX

- Multi-dimensional matrix.
- In each round, the state is multiplied by the MDS matrix.
  - $state = state \times MDS$

# MDS MATRIX

```
pub struct PoseidonParameters<F: PrimeField> {
    [...]
    pub mds: Vec<Vec<F>>,
    [...]
}

fn apply_mds(&mut self) {
    self.state = self
        .state
        .iter()
        .enumerate()
        .map(|(i, _)| {
            self.state
                .iter()
                .enumerate()
```

# S-BOX

α-power S-box:

- *S-box(x) = x^α*
  - $\alpha \geq 3$
  - $gcd(\alpha, p - 1) = 1$
- Circom uses α = 5.

# FULL VS PARTIAL S-BOX

- Full S-box is applied to all elements of the state.
- Partial S-box is applied to the first element of the state.

# FULL S-BOX

```
fn apply_sbox_full(&mut self) {
    self.state.iter_mut().for_each(|a| {
        *a = a.pow([self.params.alpha]);
    });
}
```

# PARTIAL S-BOX

```
fn apply_sbox_partial(&mut self) {
    self.state[0] = self.state[0].pow([self.params.alpha]);
}
```

# POSEIDON FULL ROUND

- Applying *ARC*.
- Applying full *S-box*.
- Applying *MDS*.

# POSEIDON FULL ROUND

- $state = state \oplus ARC$
- $state = S\text{-}box(state)$
- $state = state \times MDS$

# POSEIDON FULL ROUND

```
fn full_round(&mut self, round: usize) {
    self.apply_ark(round);
    self.apply_sbox_full();
    self.apply_mds();
}
```

# POSEIDON PARTIAL ROUND

- Applying *ARC*.
- Applying partial *S-box*.
- Applying *MDS*.

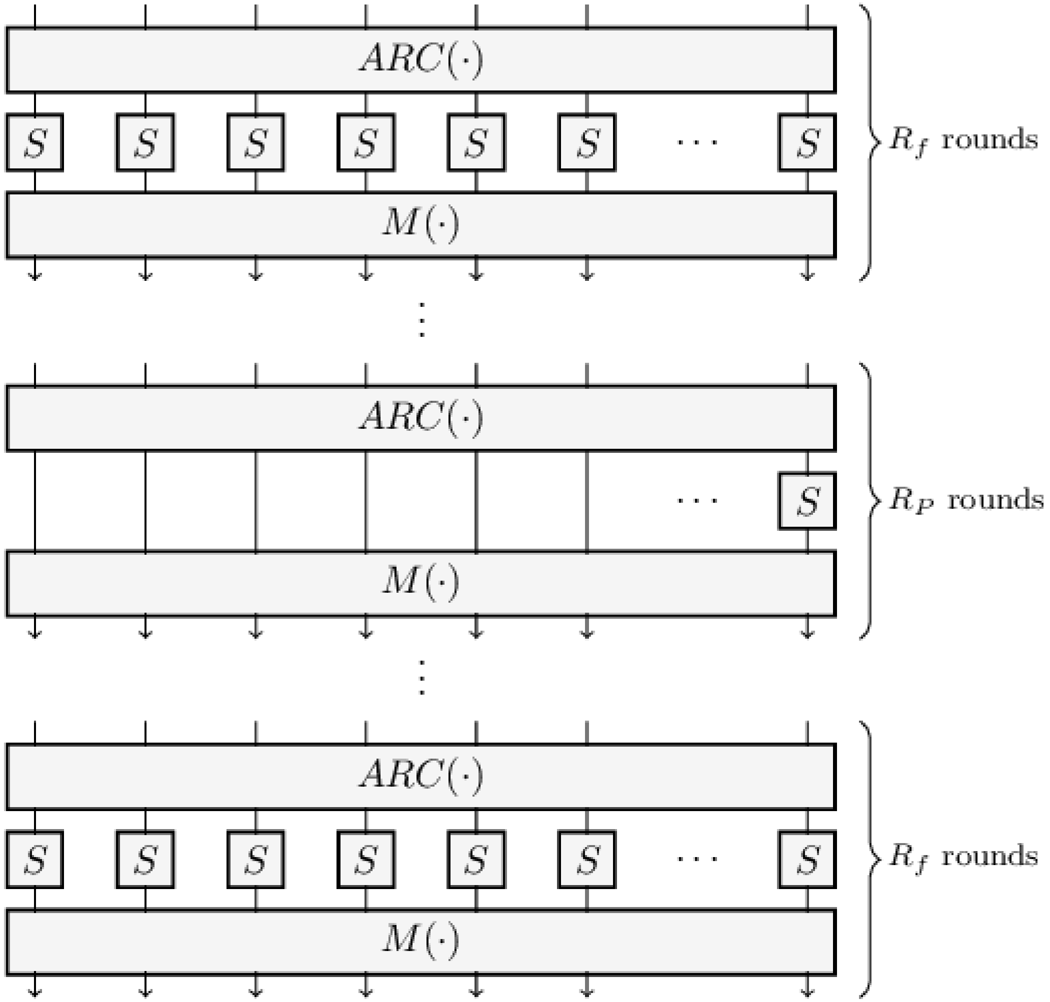
# POSEIDON PARTIAL ROUND

- $state = state \oplus ARC$
- $state[0] = S\text{-}box\text{-}partial(state[0])$
- $state = state \times MDS$

# POSEIDON PARTIAL ROUND

```
fn partial_round(&mut self, round: usize) {
    self.apply_ark(round);
    self.apply_sbox_partial();
    self.apply_mds();
}
```

# POSEIDON PERMUTATION



# POSEIDON HASH

- Numbers of rounds:
  - *F* full rounds.
  - *P* partial rounds.
- Algorithm:
  - *F/2* full rounds.
  - *P* partial rounds.
  - *F/2* full rounds.
  - Return the 1st element of the state.

# POSEIDON HASH

```
for r ∈ [0, F/2):
    state = state ⊕ ARC
    state = S-box(state)
    state = state × MDS
for r ∈ [F/2, F/2 + P):
    state = state ⊕ ARC
    state[0] = S-box-partial(state[0])
    state = state × MDS
for r ∈ [F/2 + P, F):
    state = state ⊕ ARC
    state = S-box(state)
    state = state × MDS
return state[0]
```

# POSEIDON HASH

```
let all_rounds = self.params.full_rounds + self.params.partial
let half_rounds = self.params.full_rounds / 2;

for round in 0..half_rounds {
    self.full_round(round);
}
for round in half_rounds..half_rounds + self.params.partial_ro
    self.partial_round(round);
}
for round in half_rounds + self.params.partial_rounds..all_rou
    self.full_round(round);
}

return self.state[0];
```

# SOURCES

- Filecoin Spec: Poseidon
- Poseidon whitepaper