ZKP Crash Course

Zero-Knowledge proofs

Prove correctness of an argument, without revealing the witnesses

What are zero-knowledge proofs?

Two balls and the colour-blind friend

Can we prove/verify ***arbitrary*** computations?

zk-SNARK

Zero-**K**nowledge
Succinct
Non-interactive
ARgument of
Knowledge

Zcash

tornado

zkSync

STARKNET

How is a problem represented?

```
int f(x) {
   int y = x * x * x;
   return x + y + 5;
}
```

Prove: *I know x, which f(x) = 35*

Transform the problem to R1CS form

(R1CS = Rank-1 Constraint System)

```
int f(x) {
  int y = x * x * x;
  return x + y + 5;
}
```

```
1-   x             *   x   -   sym1          =   0
2-   sym1          *   x   -   y             =   0
3-   y             *   1   -   (x + sym2)    =   0
4-   (sym2 + 5)    *   1   -   ~out          =   0
```

$$v.A * v.B - v.C = 0$$

```
(sym2 + 5)  *   1   -   ~out        =   0
```

v	A		B		C	
ONE	1	5	1	1	1	0
x	3	0	3	0	3	0
~out	35	0	35	0	35	1
sym1	9	0	9	0	9	0
y	27	0	27	0	27	0
sym2	30	1	30	0	30	0

$$35 * 1 - 35 = 0$$

Gate #4

$$A : (5, 0, 0, 0, 0, 1)$$
$$B : (1, 0, 0, 0, 0, 0)$$
$$C : (0, 0, 1, 0, 0, 0)$$

```
(sym2 + 5)  *   1   -   ~out          =   0
```

R1CS Representation

(**6** variables, **4** gates)

$A:$

$(0, 1, 0, 0, 0, 0)$

$(0, 0, 0, 1, 0, 0)$

$(0, 1, 0, 0, 1, 0)$

$(5, 0, 0, 0, 0, 1)$

$B:$

$(0, 1, 0, 0, 0, 0)$

$(0, 1, 0, 0, 0, 0)$

$(1, 0, 0, 0, 0, 0)$

$(1, 0, 0, 0, 0, 0)$

$C:$

$(0, 0, 0, 1, 0, 0)$

$(0, 0, 0, 0, 1, 0)$

$(0, 0, 0, 0, 0, 1)$

$(0, 0, 1, 0, 0, 0)$

```
int f(x) {
  int y = x * x * x;
  return x + y + 5;
}
```

What if?

```
int f(x) {
   int y = x * x * x;
   if(x > 10) {
      y = y * 2;
   }
   return x + y + 5;
}
```

Side note:

$$A, B \in \{0, 1\}$$

R1CS: $A * (1 - A) + 0 = 0$

$$A \wedge B = A * B$$

$$A \vee B = A + B - A * B$$

$$\neg A = 1 - A$$

Thus we can build conditional R1CS code!

Variables are numbers in $\mathbb{F}_p$

$$0 \leq r < P$$

$$r + s = (r + s) \mod P$$

$$r * s = (r * s) \mod P$$

$$-r = P - r$$

$$(r + s) + t = r + (s + t)$$

$$(r * s) * t = r * (s * t)$$

$$r * (s + t) = r * s + r * t$$

$$2^{-1} \mod 7 = 4$$

Multiplication Gate

$$(a) * (b) - (c) = 0$$

Addition Gate

$$(a+b)*(ONE)-(c)=0$$

Or:

$$(c)*(ONE)-(a+b)=0$$

Division Gate

$$(b) * (c) - (a) = 0$$

Boolean Restriction Gate

$$(b) * (ONE - b) - (0) = 0$$

$$\begin{aligned}
&A, B \in \{0, 1\} \\
&A \wedge B = A * B \\
&A \vee B = A + B - A * B \\
&\neg A = 1 - A
\end{aligned}$$

Bit Decomposition Gate

(+ Range check)

1 + bits constraints

$$(b_0 2^0 + b_1 2^1 + \cdots + b_{63} 2^{63}) * (ONE) - (a) = 0$$

$$\begin{aligned} & b_0 * (1 - b_0) = 0 \\ & b_1 * (1 - b_1) = 0 \\ & \vdots \\ & b_{63} * (1 - b_{63}) = 0 \end{aligned}$$

Is-Zero check gate (Naive method)

```
is_zero = 1 if a == 0 else 0
```

- Decompose the number to 255 bits (1 constraint)
- Apply OR operation on all the bits (~255 constraints)

Is-Zero check gate

(Smart method - 2 constraints)

$$IsZero = -a_{inv} * a + 1$$
$$IsZero * a = 0$$

Now, if `a` is not zero, then `is_zero` has no choice but to be zero in order to satisfy the second constraint. If `is_zero` is 0, then `a_inv` should be set to inverse of a in order to satisfy the first constraint. inverse of `a` exists, since `a` is not zero. If `a` is zero, the first constraint is reduces to `is_zero == 1`.

Equality check gate

$$Equals(a, b) = IsZero(a - b)$$

Ternary gate

```
c = s ? a : b
```

Ternary gate

Naive method (2 constraints)

c = s ? a : b

$$c = s * a + (1 - s) * b$$

$$\begin{aligned} &tmp = s * a \\ &c = tmp + (1 - s) * b \end{aligned}$$

Ternary gate

Smart method (1 constraint)

c = s ? a : b

$$(a - b) * s = a - c$$

Comparison gate (64-bit number)

```
c = a < b
c = (a - b) < 0
```

- Check if both A and B are 64-bit (Range check)
- Calculate two’s complement of B: **B_neg = 2^64 - B**
- Add A and B_neg: **Sub = A + B_neg**
- Decompose Sum into 65 bits: **SubBits = DecomposeBits(Sub, 65)**
- **c = SubBits[64]**

Hash function?

$$H(a) = a * a + 21894798 + 328a + \dots$$

Not secure :(

Hash functions must be:

- Collision resistant
- Preimage resistant

SHA-256?

Very expensive :(

- Works on bits
- Needs thousands of constraints

Poseidon hash - ZK friendly hash function

Research paper

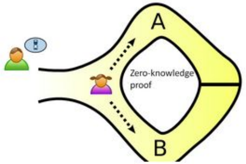
PDF

Reference implementation

Sage with test vectors

Encryption with Poseidon

PDF

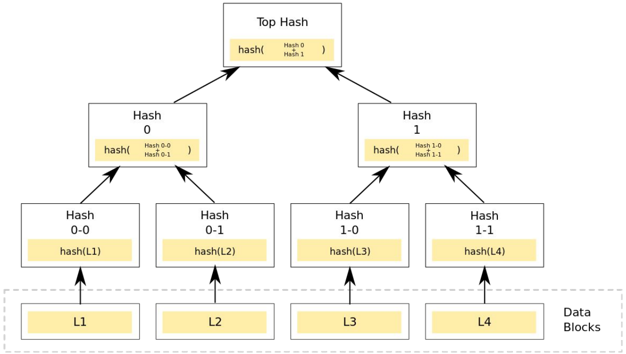


Poseidon in Plonk

PDF

TornadoCash

- Pick a secret ***s*** (E.g. $s \in \mathbb{F}_p$)
- Calculate a Commitment and a Nullifier:
  - **Commitment**: Poseidon(s)
  - **Nullifier**: Poseidon(s | 1234)
- Users will deposit their Commitment values into a PUBLIC merkle tree



- The recipient will provide a proof that he knows the corresponding nullifier of a commitment in the merkle tree.
- The nullifier becomes unusable after the withdrawal

TornadoCash

Digital Signature?

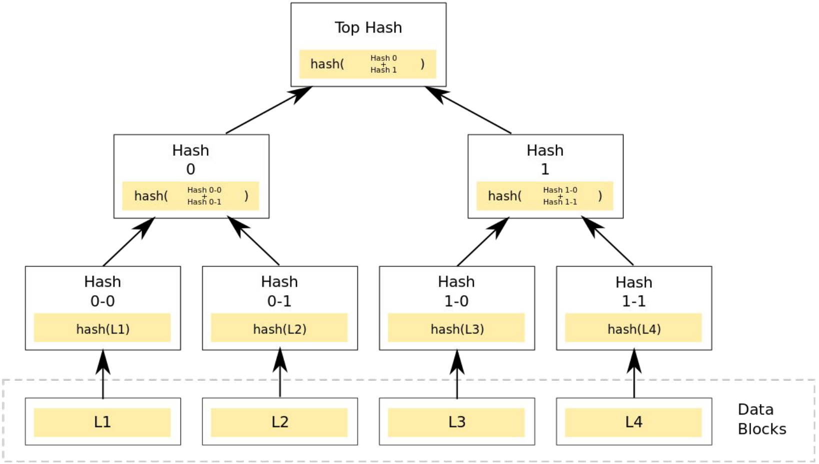
- Elliptic curve cryptography operates on points on elliptic curve defined on prime fields.
- If the prime-field of the DSA is equal with the prime-field of the circuit, then E(C/D)DSA verification can be efficiently done on the circuit! E.g. **JubJub**
- Poseidon can be used as the hash-function of the DSA

zkRollup

- Imagine an account merkle tree…

- Each account consists of 4 ***Fp*** numbers:
    - Address point X
    - Address point Y
    - Balance
    - Nonce

- Leaves are ***Poseidon(AddrX, AddrY, Balance, Nonce)***



Single Transition

- Reveal source account by providing merkle proof to the source account

- Check if **Amount < SrcBalance**

- Check if **EdDSA_Verify(Poseidon(Tx), TxSig)**

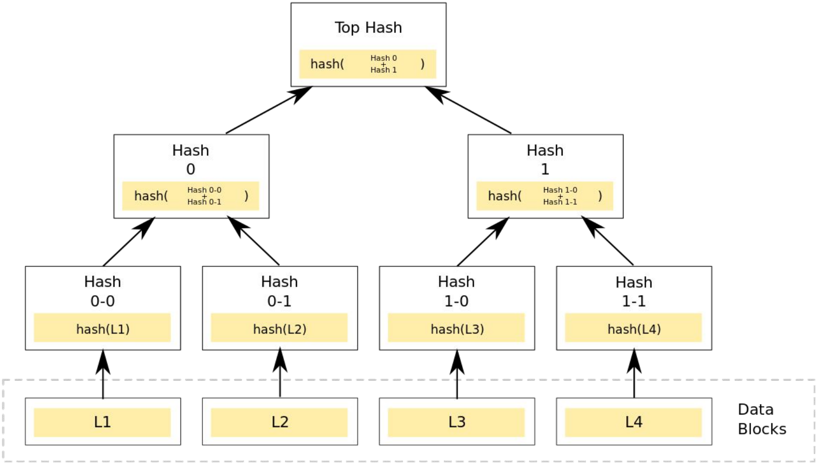
- Decrease the source account's balance, increase the source account's nonce, and recalculate the account hash:
  Poseidon(SrcAddrX, SrcAddrY, SrcBalance - Amount, SrcNonce + 1)

- Re-apply the same source merkle proof to the new account hash, to get an **intermediary** merkle-root!

- Reveal the destination account by providing merkle proof to the destination account AFTER applying the changes of source account

- Calculate new account hash of destination:
  Poseidon(DstAddrX, DstAddrY, DstBalance + Amount, DstNonce)

- Re-apply the same destination merkle proof to the new account hash, to get the **final** merkle root!



Transition Batch

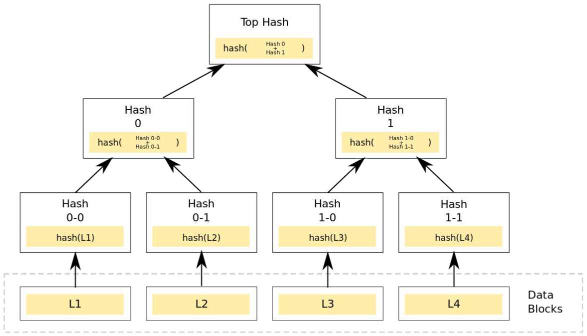
- Apply a sequence of 1024 transitions
- Then we will effectively process 1024 txs in a single low-size proof, which can be verified in constant-time.
- What if there aren’t 1024 txs?
    - Some transitions can be null, in that case, they won’t change the merkle-root
    - **CurrMerkleRoot = TransitionIsNull ? CurrMerkleRoot : NewMerkleRoot**

zkVM

- Imagine a zkRollup of VM instructions…
- Machine state is merkle-root of its RAM and its CPU registers
- Program is stored as a list of opcodes in RAM, and there is a Program-Counter (PC) register

Sparse Merkle Tree

- Merkle Tree with a fixed size of **2^n**
- Can contain billions of leaves
- Is sparse
- Each leaf has a “Default value” and default-values of lower depths are also precalculated
- ***Defaults = [0, Poseidon(0, 0), Poseidon(Poseidon(0, 0), Poseidon(0, 0)), …]***



Sparse Merkle Trees are RAMs

- SMTs are Random Access Memories with log(n) access time-complexity