Disclaimer: This is a very experimental project, and the components described here is strongly unrecommended to use in production without further research.

Thanks to Oleg Taraskin and Kobi Gurkan for links.

Fast elliptic point compression for the subgroup

Public and private keys are used at twisted Edwards snark-friendly curve babyJubJub

a x^2 + y^2 = 1 + d x^2 y^2,

where all equations are carried out at field F_p

, where p

is point order for BN254. a = 168700

, d = 168696

.

Curve order for babyJubJub is 8q

, where q

is 251bit prime number.

The addition rule for the curve is

(x_1, y_1) + (x_2, y_2) = \frac{x_1 y_2 + y_1 x_2}{1+d x_1 x_2 y_1 y_2} + \frac{y_1 y_2 - a x_1 x_2}{1-d x_1 x_2 y_1 y_2}.

This mean that

(x, y) + (x, -y) = (0, -1)

.

(0, -1)

is not in F_q

subgroup and points (x, y)

and (x, -y)

cannot be at subgroup both.

It is enough to store only first coordinate x

to keep the public key and unpack the point when we need.

Here is implementation for point decompression. We use it ateddsa and ecvrf. This approach allows to use much lesser constraints to keep and hash the points.

Nullifier computation via ECVRF.

Each UTXO has a unique deterministic commitment to protect from double-spends. If this commitment is already published onchain, the contract reject the transaction.

It is important that nullifier is depended from the private key of the owner. That means that UTXO creator does not know the nullifier if he is not the owner and nobody can track when the UTXO is spent.

The naive way to compute the nullifier is using

N := hash(sk, utxoId)

, where sk

is the private key and utxoId

is a unique id for the UTXO.

But in this case, we need to keep sk

and the same device as zkSNARK prover. Snarks computation is a heavy thing, so, it is not working for hardware wallets.

Another way is by using ECVRF. Here is the construction inside our solution:

Keys

Y = x G

, where G

is generator point, Y

is public key and x

is a private key.

Proof

H := Hash(Y, alpha) Gamma := x H k := nonce(x, H) c := hash(H, Gamma, k G, k H) s := (k+c x) mod q return {Gamma, c, s}

Where nonce is determenistic pseudo-random function (we use one based on blake512 here), Hash

is elliptic curve point hash function, hash

is scalar hash function, gamma

is deterministic verifiable commitment.

We use Pedersen hash for both cases.

In original paper increment and try, the method is used to hash value into a point. We cannot use probability methods for snarks. And Shallue-Woestijne-Ulas point picker is useful for further works based on MiMC or Poseidon hashes.

Verify

H := Hash(Y, alpha) U := s G - c Y V := s H - c Gamma c == hash(H, Gamma, U, V)

Circom implementation is here and js implementation is here.

Shallue-Woestijne-Ulas point picker

The solution is based on Shallue and Woestijne result, that exists x_1

, x_2

, x_3

, x_4

, and

f(x_1)f(x_2)f(x_3) = x_4^2

, where f

is polynomial in Weierstrass representation.

That means that at least one of f(x_1)

, f(x_2)

, f(x_3)

is full square.

Ulas found that it is working for

X_1=u

X_2=−\frac{b}{a} \left(1 +\frac{1}{t^ 4g(u)^2+t^2g(u)}\right)

X_3=t^2g(u) X_2

We select u = 5

, so, g(X_1)

is not full square. Also, t^ 4g(u)^2+t^2g(u)

never equals zero for nonzero t

.

So, we need to check inside the snark only f(X_2)

, f(X_3)

, find y

in Weierstrass representation and process Wierstrass to Edwards point transformation.

Circom implementation is here and js implementation is here.

Bibliography

1. S. Goldberg, L. Reyzin, D. Papadopoulos, J. Vcelak “Verifiable Random Functions (VRFs)”
2. Maciej Ulas “Rational points on certain hyperelliptic curves over finite fields”

UPD

In current ecvrf implementation, Pedersen hash is used. This function is not a random oracle and must be replaced in production. Noticed by Kobi Gurkan.