Group signature with zk

zk-school: beginner

(3/16, 2023)

정동현

Normal signature

KeyGen → (sk, pk): selects a random secret key sk and corresponding public key pk

Sign(m, sk) → s: given a message m and secret key, outputs a signature s

Verify(m, s, pk) → 1/0: given a message m, a signature s, and a public key pk, verifies if signature is valid

ECDSA

Alice

Bob

Key Generation

PrivKey sk $\in^{R} \{1, \dots, q-1\}$

PubKey pk = sk * G

pk

Sign

$r = (k \cdot G)_x, k \in^{R} \{1, \dots, q-1\}$

$s = k^{-1} \cdot [\mathrm{Hash}(msg) + r \cdot sk]$

Signature = (r, s)

msg

Verification

$u_1 = \mathrm{Hash}(msg) / s$

$u_2 = r / s$

$(x_1, y_1) = u_1 * G + u_2 * pk$

If $x_1 \equiv r \pmod{q}$, (r, s) is valid

[Correctness] If (r, s) is a valid signature,

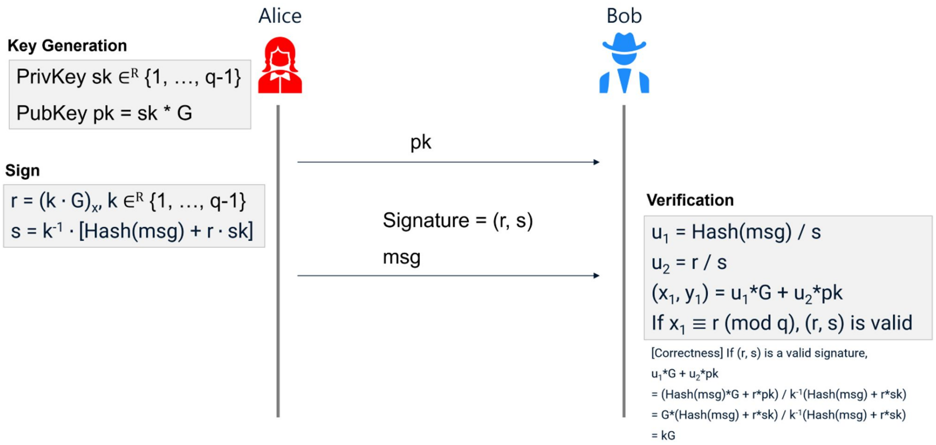
$u_1 * G + u_2 * pk$

$= (\mathrm{Hash}(msg) * G + r * pk) / k^{-1}(\mathrm{Hash}(msg) + r * sk)$

$= G * (\mathrm{Hash}(msg) + r * sk) / k^{-1}(\mathrm{Hash}(msg) + r * sk)$

$= kG$

ECDSA



Keygen

sk, pk(=$sk * G$)

Sign

random k,

$k * G$ = elliptic curve point (x_1, y_1)

r = x_1

s = $k^{-1}[H(m) + r * sk]$

signature(r, s), message(m)

Verify

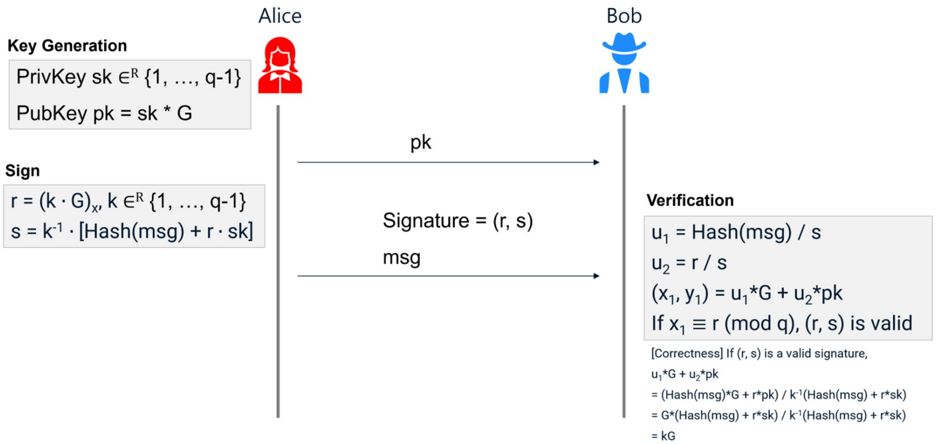
$$u1 = H(m) * s^{-1}$$

$$u2 = r * s^{-1}$$

$$\boxed{r = u1 * G + u2 * pk}$$

check!

ECDSA



Keygen

sk, pk(= $sk * G$)

Sign

random k,

$k * G$ = elliptic curve point (x_1, y_1)

r = x_1

s = $k^{-1}[H(m) + r * sk]$

signature(r, s), message(m)

Verify

$u1 = H(m) * s^{-1}$

$u2 = r * s^{-1}$

$\boxed{r = u1 * G + u2 * pk}$ ← **check!**

$$u1 * G + u2 * pk = s^{-1}(H(m) * G + r * pk)$$

$$s^{-1}(H(m) * G + r * pk) = (H(m) * G + r * sk * G)/k^{-1}(H(m) + r * sk)$$

$$k * G$$

simple signature in zk(keygen)

```
include "circomlib/poseidon.circom";


template SecretToPublic() {
    signal input sk;
    signal output pk;


    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== sk;
    pk <== poseidon.out;
    }
```

simple signature in zk(sign)

```
template Sign() {
    signal input m;
    signal input sk;
    signal input pk;
    // verify prover knows correct sk
    component checker = SecretToPublic();
    checker.sk <== sk;
    pk === checker.pk;

}
    component main { public [ pk, m ] } = Sign();
```

simple signature in zk(sign)

```
template Sign() {
    signal input m;
    signal input sk;
    signal input pk;
    // verify prover knows correct sk
    component checker = SecretToPublic();
    checker.sk <== sk;
    pk === checker.pk;
    signal m_square <==  m*m;
}
    component main { public [ pk, m ] } = Sign();
```

simple signature in zk(verify)

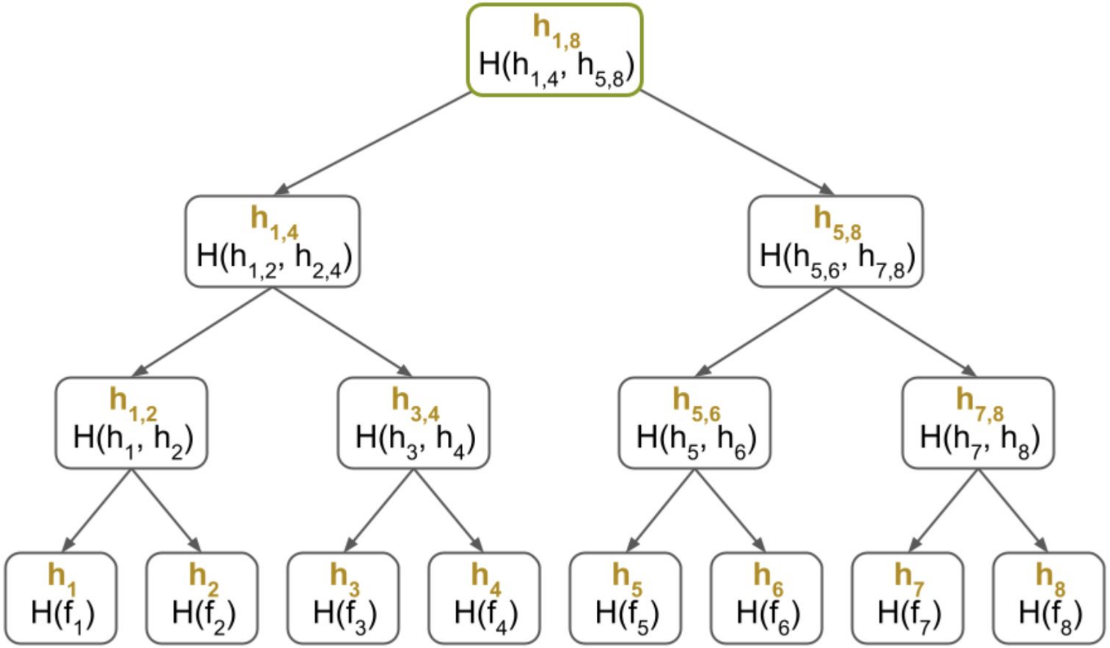
zk snark proof가 마치 signature로써 쓰일 수 있음.

verifier가 public input값 pk와 메시지 m과 signature proof를 통해 해당 proof가 유효한지 T/F 체크로 verify.

Group signature

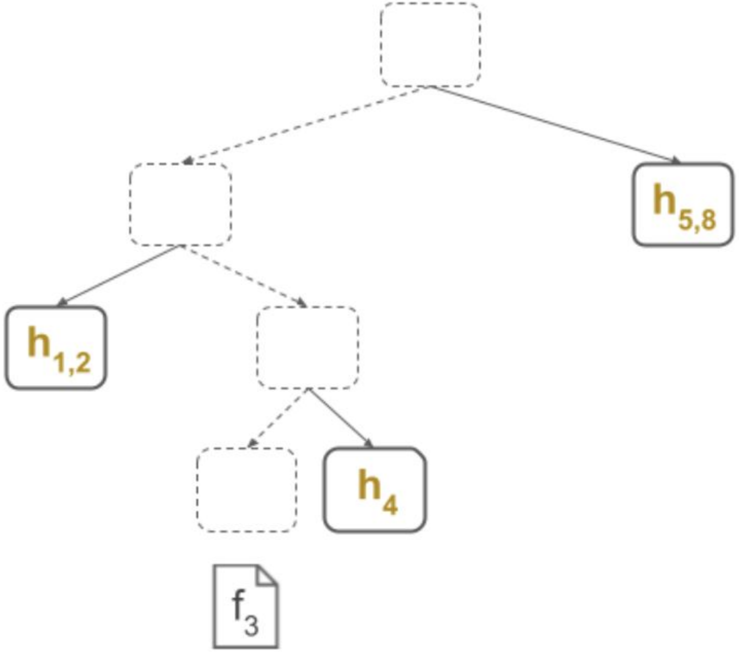
```
template GroupSign(n) {
    signal input m;
    signal input sk;
    signal input pk[n];
    // verify prover knows correct sk
    component checker = SecretToPublic();
    checker.sk <== sk;
// checker.pk is going to correspond your pk
++++++
signal zeroChecker[n+1];
zeroChecker[0] <== 1;
for (var i = 0; i < n; i++){
zeroChecker[i+1] <== zeroChecker[i] * (pk[i] - checker.pk);
}
zeroChecker[n] === 0;    결과값으로는 0이되게된다.
++++++
signal m_square <==  m*m
}
```

Merkle Tree



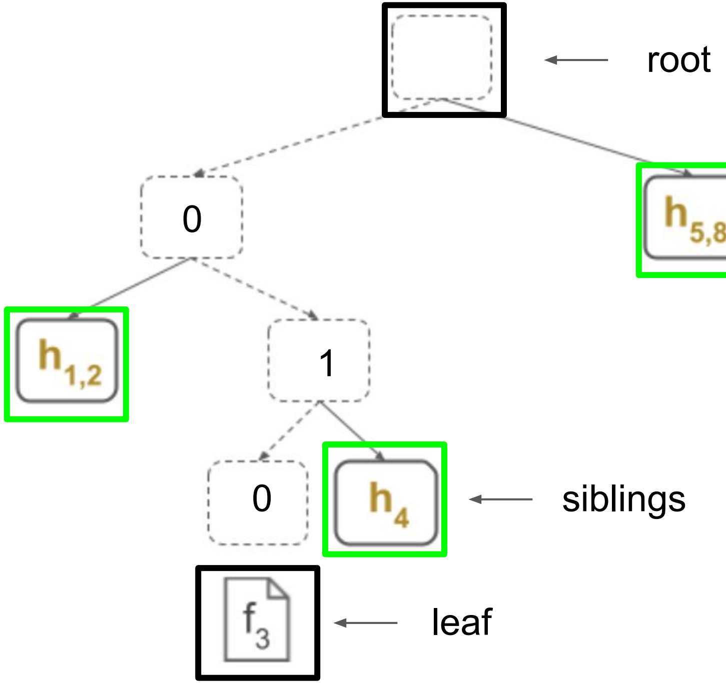
$h_{1,8}$
$H(h_{1,4}, h_{5,8})$
$h_{1,4}$
$H(h_{1,2}, h_{2,4})$
$h_{5,8}$
$H(h_{5,6}, h_{7,8})$
$h_{1,2}$
$H(h_1, h_2)$
$h_{3,4}$
$H(h_3, h_4)$
$h_{5,6}$
$H(h_5, h_6)$
$h_{7,8}$
$H(h_7, h_8)$
h_1
$H(f_1)$
h_2
$H(f_2)$
h_3
$H(f_3)$
h_4
$H(f_4)$
h_5
$H(f_5)$
h_6
$H(f_6)$
h_7
$H(f_7)$
h_8
$H(f_8)$

Merkle Tree proof



Merkle Tree proof

pathIndices
= [0,1,0]



Big size Group signature (merkle Tree)

```
template MerkleTreeInclusionProof(nLevels) {
    signal input leaf;
    signal input pathIndices[nLevels];
    signal input siblings[nLevels];
    signal input root;

    component mux[nLevels];
    component poseidons[nLevels];

    signal hashes[nLevels+1];
    hashes[0] <== leaf;

    for (var i = 0; i < nLevels; i++) {
        mux[i] = DualMux();
        mux[i].in[0] <== hashes[i];
        mux[i].in[1] <== siblings[i];
        mux[i].s <== pathIndices[i];

        poseidons[i] = Poseidon(2);
        poseidons[i].inputs[0] <== mux[i].out[0];
        poseidons[i].inputs[1] <== mux[i].out[1];
        hashes[i+1] <== poseidons[i].out;
    }

    root === hashes[nLevels];
}

component main { public [ leaf, root ] } = MerkleTreeInclusionProof(3);
```

```
// if s == 0 returns [in[0], in[1]]
// if s == 1 returns [in[1], in[0]]
template DualMux() {
    signal input in[2];
    signal input s;
    signal output out[2];

    s * (1 - s) === 0;
    out[0] <== (in[1] - in[0])*s + in[0];
    out[1] <== (in[0] - in[1])*s + in[1];
}
```

EDDSA in zk-snark

ecdsa는 쉬운일은 아님. - https://github.com/0xPARC/circom-ecdsa

bn254(이더리움에 precompiled되어있는)field안에서 구현된 baby-jubjub

해당 커브 구현을 사용하여 eddsa나 커브를 사용하는 pedersen hash를 사용할 수 있음

```
inputs             zk-SNARK (alt_bn128(a.k.a bn254))          output
          +---------------------------------------------+
          |   +---------------------+                   |
      --->|   | EdDSA (Baby Jubjub)|                    |
          |   +---------------------+                   |
      --->|                                             |--->
          |          +------------------------------+   |
      --->|          | Pedersen Hash (Baby Jubjub) |    |
          |          +------------------------------+   |
          +---------------------------------------------+
```