

# Identity Blockchain - Proof of Identity

Albert Vučinović, University North  
Miroslav Jerković, University of Zagreb

June 28, 2022

## Abstract

*Identity Blockchain* uses state-certified electronic identities (eIDs) to create blockchain identities and a consensus protocol called Proof of Identity (PoI). PoI is "green" and enables many applications that are impossible to implement without verified identity on the blockchain, e.g., direct democracy and universal basic income, encrypted messaging, etc. *Identity Blockchain* preserves identity anonymity by using zk-SNARKs and an anonymization protocol for identity onboarding.

## 1 Introduction

TODO

## 2 Notation

$(K, K^{-1})$	Pair of public key $K$ and the corresponding private key $K^{-1}$
$K(value)$	String $value$ encrypted with a public key $K$
$K^{-1}(value)$	String $value$ encrypted with a private key $K^{-1}$
$H$	zk-SNARK-friendly hash function
$Nin$	Unique state-issued personal <i>National Identification Number</i>
$CA$	Certificate Authority
$eID$	Electronic identification document
$K_{ID}$	CA-certified public key of eID
$K_P$	Person key, unique public key used to access <i>Identity Blockchain</i>

Sometimes we write  $K$  for  $(K, K^{-1})$ .

### 3 Registering $K_{ID}$

Register  $K_{ID}$  on *Identity Blockchain* by calling

$$registerK_{ID}(H(Nin), K_{ID}^{-1}("registerK_{ID}"), proofNinOwnsKid).$$

Pseudo-solidity code

```
1  mapping(uint256 => uint256) public ninKidUsed;
2  mapping(uint256 => bool) public kidInvalid;
3
4  function registerKid(
5      uint256 hNin, // hNin = H(Nin)
6      uint256 vKid, // vKid = KidPrivate("registerKid")
7      bytes proofNinOwnsKid
8  ) public {
9      uint256 previousKey = ninKidUsed[hNin];
10     if (previousKey == vKid) return; // already registered
11     bool isOwner = zksnarkverify(VK, [hNin, vKid],
12         proofNinOwnsKid); // VK = verification key
13     if (isOwner && !kidInvalid[vKid]) {
14         if (previousKey != 0) kidInvalid[previousKey] = true;
15         ninKidUsed[hNin] = vKid;
16     }
```

Features

#### **Sybil resistance**

A Person is prevented from registering more than one  $K_P$ , e.g. by using different identification documents issued for the same  $Nin$ .

#### **Fix for the loss of eID**

A Person who has lost a previously registered  $K_{ID}$ , e.g. due to loss of eID, can overwrite it by registering a new  $K_{ID}$ .

#### **Identity theft damage control**

If a Person overwrites the previous  $K_{ID}$  with a new one, the identity Thief loses access to services that verify the entire identity chain.

Remarks

- i) An entity that knows  $K_{ID}$ , e.g. the CA who certified the corresponding eID, can know that the Person has registered  $K_{ID}$  on *Identity Blockchain*.
- ii) Reasoning for the choice of arguments: if  $H(Nin, K_{ID}^{-1}("registerK_{ID}"))$  was used as an argument, then TODO. On the other hand, the choice of arguments  $Nin$  and  $K_{ID}^{-1}("registerK_{ID}")$  would TODO.

## 4 Registering $K_P$

Register  $K_P$  on *Identity Blockchain* by calling

$$\text{register}K_P(\text{Nin}K_{ID}, \text{Nin}K_{ID}K_P, \text{proof}),$$

where:

$$\text{Nin}K_{ID} = H(\text{Nin}, K_{ID}^{-1}(\text{"register}K_P\text{"}))$$

$$\text{Nin}K_{ID}K_P = H(\text{Nin}, K_{ID}^{-1}(\text{"register}K_P\text{"}), K_P^{-1}(\text{"register}K_P\text{"})).$$

Pseudo-solidity code

```
1  mapping(uint256 => uint256) public ninKidKpUsed;
2  mapping(uint256 => bool) public kpInvalid;
3  mapping(uint256 => uint256) public ninKidKpLastConfirmed;
4
5  function registerKp(
6      uint256 ninKid,
7      uint256 ninKidKp,
8      bytes proof
9  ) public {
10     uint256 previousKey = ninKidKpUsed[ninKid];
11     if (previousKey == ninKid) return; // already registered
12     bool ok = zksnarkverify(VK, [ninKid, ninKidKp], proof); //
        VK = verification key
13     if (ok && !kpInvalid[ninKidKp]) {
14         if (previousKey != 0) kpInvalid[previousKey] = true;
15         ninKidKpUsed[ninKid] = ninKidKp;
16         ninKidKpLastConfirmed[ninKid] = block.number;
17     }
18 }
```

Features

### Sybil resistance

$K_P$  is a unique public key bound to the Person on the *Identity Blockchain*. By design, the Person cannot have two valid  $K_P$  keys at the same time.

TODO: Explanation/proof.

### Anonymity

No one who has a database with all public keys  $K_{ID}$  or/and public keys  $K_P$  can tell, from the function call, which  $\text{Nin}$ ,  $K_{ID}$  or  $K_P$  was used. This is true under the assumption that CA, which certified the eID, does not have access to  $K_{ID}^{-1}$ , i.e. that  $K_{ID}$  was securely generated on the eID.

TODO: Explanation/proof.

Remarks

- i) Reasoning behind the choice of arguments: TODO

## 5 zkp code

```
1 //pseudo zokrates zk-SNARK
2 import "hashes/sha256/512bitPacked" as sha256packed
3 import "ecc/babyjubjubParams.code" as context
4 import "ecc/proofOfOwnership.code" as proofOfOwnership
5
6 def proofPINOwnsK(
7     field[2] PINHash,
8     field[2] K_1Hash,
9     field[2]
10
11     private field[?] CACert){
12 }
13
14 def proof_of_being_a_person(
15     //publically known arguments
16     field[2] PINKeyUsedHash,
17     field[2] PINKeyPersonKeyIssuedHash,
18     field[2] PersonKeyInvalidHash,
19     field[2] KeyInvalidHash,
20
21     //BC state
22     field BC_PINUsed, //BC State
23     field BC_PINKeyUsed, //BC State
24     field BC_PINKeyPersonKeyIssued, //BC State
25     field BC_PersonKeyInvalid, //BC State
26     field BC_KeyInvalid, //BC State
27     ?field BC_private_key_challenge?
28
29     //CA keys!
30     field[?][?] CAKeys, //? BC State
31     ...
32
33     //private data
34     //15360 bit RSA key is equivalent to 256-bit symmetric keys
35     //2048 bit RSA key is equivalent to 112 bit symmetric keys
36     //eID has 2048 bit RSA
37     private field PIN, //max 254 bits, using only 128 = 16 bytes
38     private field K_private_bc_new, //ECC private key
39     private field[?] K_new, //public RSA key from eID
40     private field[?] K_1_new, //private RSA key from eID, actually
41         signed message
42     //K_1_new("PeopleBC person proof")
43     private field K_private_bc_old, //ECC private key, old
44     private field[?] K_1_old, //K_1_old("PeopleBC person proof")?
45     //?private field[2] K_bc_new, //ECC public key ?no need for key
46         pair verification?
47     //?private field[2] K_bc_old, //ECC public key ?no need for key
48         pair verification
49     private field[?] CACert,
50     private field CAindex
51 {
52     //actual checking code here
53     field[2] hash_PIN = sha256packed([0,0,0,PIN])
54     assert(hash_PIN == PINHash); //proving that we know the real PIN,
55         unimportant
```

```

52
53 //proving ownership of newly registered key,
54 //no real need to prove ownership of bc key
55 //we actually need to prove ownership of K_1_new,
56 //and its connection to PIN via CA
57 context=context()//babyjubjubParams context
58 proofOfOwnership(K_bc_new, K_private_bc_new, context)==1
59
60 //prove that you own the K_new
61 //pseudo
62 RSADecrypt(K_new, K_1_new) == "PeopleBC Person Proof";//?+
    BC_private_key_challenge;
63
64 //prove that Key is connected to CA
65 //pseudo
66 checkCASignature(CAKey[CAindex], CACert, K_new) == 1;
67
68 //prove that K_new is PIN certificate
69 //pseudo
70 extractPIN(CACert)==PIN;
71
72 //end
73
74 //end..
75 }

```

## 6 Problems

- P1** The rate of addition of identities, the problems of frequent changes of Merkle Tree Roots, and how long it takes to generate zkp proof, and block generation time.
- P2** The Thief stole  $K_{ID}^{-1}$  before we registered on *Identity Blockchain* and then registered  $K_{ID}$  and  $K_P$ .
- P3** The Thief stole  $K_{ID}^{-1}$  after we registered on *Identity Blockchain*.
- P4** Trying to use unregistered Key.
- P5** Trying to register the same *Nin* with multiple Keys (e.g. two physical ids).
- P6** Using  $K_P$  without checking the whole  $CA \rightarrow K_{ID} \rightarrow K_P$  chain.
- i) Using  $K_P$  to register as a mining key.
  - ii) Invalidating  $K_P$ .
  - iii) Using new  $K_P$  as a mining key.

### Solution

Look at the solution to **P7**.

The Mining Registry can accept new  $K_P$  after an expiration period (which can be e.g. one day).

- P7** A combination of **P2** and **P6**. The Thief steals  $K_{ID}$ , registers  $K_P$  and registers  $K_P$  for mining. The problem is bigger, because we can't invalidate  $K_P$  if we don't know which  $K_P$  it is.

### Solution

We make  $K_P$  renewable. We write the last block number it was renewed on to the *Identity Blockchain*. Mining Registry can choose to accept  $K_P$  that is not older than some time period (for the Registry a good period would seem to be one day). When paying the miners, the Registry also requires proof of  $K_P$ .

- P8** Only  $K_P$  is compromised.