Proof of Identity

Albert Vučinović, University North
Miroslav Jerković, University of Zagreb

June 26, 2022

1 Notation

(K, K^{-1})	A pair of public K and the corresponding private key K^{-1}.
$K(value)$	*value* encrypted with a public key K.
$K^{-1}(value)$	*value* encrypted with a private key K^{-1}.
H	Hash function, e.g. some zksnark friedly one
K_{ID}	State CA certified public key of e.g. eID
K_P	Person key, public key used to access *Identity Blockchain* as an unknown Person.
Nin	A personal National Identification Number issued by state.

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

2 Registering K_P (Person key)

1. Register K_{ID} on *Identity Blockchain*, by calling

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

   The Person tells the *Identity Blockchain* what personal identification document will they be using.

   What is the meaning behind $registerK_{ID}$ function?

   - Sybil resistance. To prevent a Person from registering more than one K_P using different identification documents, which are issued for the same Nin.
   - Loss of K_{ID}. If a Person looses previously registered K_{ID} (e.g. by loosing the eID), they can register a new one.
   - Identity theft damage control. If a Person overwrites the previous K_{ID} with a new one, the identity thief will loose access to services that check the whole identity chain.

   Entities who know K_{ID} (e.g. CA that issued it) can know that the Person is using *Identity Blockchain*, and what identification document the Person is using.

2. Register K_P on *Identity Blockchain*, by calling

$$registerK_P(NinK_{ID}, NinK_{ID}K_P, proof)$$

   where:

   (a) $NinK_{ID} = H(Nin, K_{ID}^{-1}("registerK_P"))$

   (b) $NinK_{ID}K_P = H(Nin, K_{ID}^{-1}("registerK_P"), K_P^{-1}("registerK_P"))$

3 On chain code

```
//pseudo Solidity
//Key is eID private key, or PrivateeIDKey("")
//PersonKey is Blockchain key used by a Person
contract IdentitiesManager {
  //all the mappings are mappings where uint256 is a hash!
  mapping(uint256 => uint256) public PinKUsed;
  mapping(uint256 => uint256) public PinKPUsed;
  mapping(uint256 => uint256) public PinKPLastConfirmed;
  mapping(uint256 => bool) public PInvalid;
  mapping(uint256 => bool) public KInvalid;
  //TODO:CA fields!

  //K_1=K_1("registerK")
  //Pin=H(Pin)
  function registerK public(
    uint256 Pin,
    uint256 K_1,
    bytes proofPinOwnsK
  ){
    uint256 previousKey = PinKUsed[Pin];
    if(previouseKey == K_1)
      return;
    bool isOwner = zksnarkverify(VK, [
        PIN,
        K_1
      ],
      proofPINOwnsK);
    if(isOwner && ! KInvalid[K_1]){
      if(previousKey!=0)
        KInvalid[previousKey]=true;
      PinKUsed[Pin]=K_1;
    }
  }

  //PinK=H(Pin, K_1("registerP")) which is != H(Pin, K_1("registerK"))
  //PinK can not be connected to Pin, unless one knows K_1
  function registerP public(
    uint256 PinK,
    uint256 PinKP,
    bytes proof
  ){
    uint256 previousKey = PinKPUsed[PinK]
    if(previousKey == PinK)
      return; //already registered

    //zkp(
      //K!invalid
        //?Need KInvalid to be Zero Knowledge Set (ZKS), and to get roots
      //CA!invalid
      //CA -> K -> Pin
      //PinK=H(Pin, K_1("registerP"))
      //PinKP=H(Pin, K_1("registerP"), P_1("registerP"))
    //)
    bool ok = zksnarkverify(VK, [
    ], proof);

    if(ok && ! PInvalid[PinKP]){
      if(previousKey != 0)
        PInvalid[previousKey]=true;
      PinKPUsed[PinK]=PinKP;
      PinKPLastConfirmed[PinK]=block.number;
    }

  }

}

contract PersonAccount{
}
```

4 zkp code

```
//pseudo zokrates zk-SNARK
import "hashes/sha256/512bitPacked" as sha256packed
import "ecc/babyjubjubParams.code" as context
import "ecc/proofOfOwnership.code" as proofOfOwnership

def proofPINOwnsK(
  field[2] PINHash,
  field[2] K_1Hash,
  field[2]

  private field[?] CACert){
}

def proof_of_being_a_person(
  //publically known arguments
  field[2] PINKeyUsedHash,
  field[2] PINKeyPersonKeyIssuedHash,
  field[2] PersonKeyInvalidHash,
  field[2] KeyInvalidHash,

  //BC state
  field BC_PINUsed,//BC State
  field BC_PINKeyUsed,//BC State
  field BC_PINKeyPersonKeyIssued,//BC State
  field BC_PersonKeyInvalid,//BC State
  field BC_KeyInvalid,//BC State
  ?field BC_private_key_challenge?

  //CA keys!
  field[?][?] CAKeys,//? BC State
  ...

  //private data
  //15360 bit RSA key is equivalent to 256-bit symmetric keys
  //2048 bit RSA key is equivalent to 112 bit symmetric keys
  //eID has 2048 bit RSA
  private field PIN,//max 254 bits, using only 128 = 16 bytes
  private field K_private_bc_new,//ECC private key
  private field[?] K_new,//public RSA key from eID
  private field[?] K_1_new,//private RSA key from eID, actually signed message
  //K_1_new("PeopleBC person proof")
  private field K_private_bc_old,//ECC private key, old
  private field[?] K_1_old)//K_1_old("PeopleBC person proof")?
  //?private field[2] K_bc_new,//ECC public key ?no need for key pair verification?
  //?private field[2] K_bc_old,//ECC public key ?no need for key pair verification
  private field[?] CAcert,
  private field CAindex
{
  //actual checking code here
  field[2] hash_PIN = sha256packed([0,0,0,PIN])
  assert(hash_PIN == PINHash);//proving that we know the real PIN, unimportant

  //proving ownership of newly registered key,
  //no real need to prove ownership of bc key
  //we actually need to prove ownereship of K_1_new,
  //and its connection to PIN via CA
  context=context()//babyjubjubParams context
  proofOfOwnership(K_bc_new, K_private_bc_new, context)==1

  //prove that you own the K_new
  //pseudo
  RSADecrypt(K_new, K_1_new) == "PeopleBC Person Proof";//?+BC_private_key_challenge;

  //proove that Key is connected to CA
  //pseudo
  checkCASignature(CAKey[CAindex], CACert, K_new) == 1;

  //proove that K_new is PIN certificate
  //pseudo
  extractPIN(CACert)==PIN;

  //end

  //end..
}
```

5 Problems

Problem 1. *The rate of adding identities, the problem being frequent changes of Merkle Tree Roots, and how long it takes to generate zkp proof, and block generation time.*

Problem 2. *Someone stole K^{-1} before we registered on Identity Blockchain. The Thief registered K and P, and who knows what else.*

Solution: □

Problem 3. *Someone stole K^{-1} after we registered on Identity Blockchain.*

Problem 4. *Trying to use unregistered Key.*

Solution: □

Problem 5. *Trying to register the same PIN with multiple Keys (e.g. two physical ids).*

Problem 6. *Using P without checking the whole $CA \to K \to P$ chain.*

- *Using P to register as a mining key.*
- *Invalidating P.*
- *Using P_{new} as a mining key.*

Solution: Look at the solution to Problem 7.

The Mining Registry can accept new Ps after an expiration period (which can also be e.g. 1 day). □

Problem 7. *A combination of Problem 2 and Problem 6. Someone steals K, registeres P and registeres P for mining. The problem is bigger, because we can't invalidate P if we don't know which P it is.*

Solution: We make P renewable. We write the last block number it was renewed on to the blockchain. Mining Registry can choose to accept P which is not older than some time period (for the Registry a good period would seem to be 1 day). When paying the miners, the Registry also requires proof of P's age.

K is easely invalidated, because there is an explicit $PIN \to K^{-1}$ mapping on the blockchain, so it doesn't need to be renewable in the sence to proove it is still valid. □

Problem 8. *Only P compromised.*