Identity Blockchain - Proof of Identity

Albert Vučinović, University North
Miroslav Jerković, University of Zagreb

July 23, 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 *National Identification Number*
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 *Identity Blockchain*

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

3 Registering identity key (K_P)

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

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

Pseudo-solidity code

```
mapping(uint => uint) public ninKidUsed;
mapping(uint => uint) public ninKidKpUsed;
mapping(uint => bool) public kidInvalid;

function register(
  uint hNin, // hNin = H(Nin)
  uint hNinKid,
  uint hNinKidKp,
  bool invalidate, //true if someone stole your previous id
  uint[2] memory proofPointA,
  uint[2][2] memory proofPointsB,
  uint[2] memory proofPointC,
) public {
  //Setting public arguments
  uint[] memory inputValues = new uint[](3);
  inputValues[0]=hNin;
  inputValues[1]=hNinKid;
  inputValues[2]=hNinKidKp;


  require(verifyProof(
    proofPointA,
    proofPointsB,
    proofPointC,
    inputValues
  ));


  uint previousHNinKid = ninKidUsed[hNin];
  if (previousHNinKid != hNinKid and invalidate == true) {
    //If they are different, we need to invalidate old Kid
    kidInvalid[previousHNinKid] = true;
  }
  require(!kidInvalid[hNinKid]);


  ninKidUsed[hNin] = hNinKid;

  ninKidKpUsed[hNinKid] = hNinKidKp;

}
```

Pseudo-circom code

Using https://github.com/zkp-application/circom-rsa-verify for now, MIT license. Which probably doesn't work, we should check (sha256?).
The next line is used on a file of shape:
—–BEGIN PUBLIC KEY—–

...
——END PUBLIC KEY——
——BEGIN CERTIFICATE——
...
——END CERTIFICATE——
We assume:
modulus = Subject Modulus
exp = Subject Exponent
openssl x509 -in Identification.pem -text -noout

Statements we need to check:

- CA is an accepted CA by *Identity Blockchain*
  - A set representation exists on blockchain which contains CA public keys (CA-public-keys).
  - Verify inclusion of the CA in cert in CA-public-keys.
  - A set representation exists on blockchain which has all the revoked CA public keys (Revoked-CA-public-keys).
  - Verify exclusion of the CA in cert from Revoked-CA-public-keys.
  - Verify CA validity with respect to time (maybe blockchain)
- Cert CA signed
  - the Kid
  - x509 Subject that contains Nin in the right place in the subject
  - verify Validity of x509 cert with respect to time
- Kid is not on revoked certs list by CA (blockchain hosted list)
- Kid decrypts vKid into clear text "registerKid".

```
pragma circom 2.0.0;

include "./utils/rsa_verify.circom"
include "./circomlib/circuits/sha256/sha256.circom"
include "./circomlib/bitify.circom"

//hashNumberOfWords = 4 for sha256
//rsaNumberOfWords = 32 for 2048 rsa
//we use words of 64 bits
template NinOwnsKid(
    rsaNumberOfWords,
    hashNumberOfWords,
    lengthInBitsOfSignedMessage,
    lengthOfNinInBits,
    bitsInAWord
){
  //TODO: What happens to uint256 when converted to signal

  //public:
  signal input hNin;
  signal input vKid;
```

```

//private:
signal input Nin;
//Kid, CA signature
singal input exp[rsaNumberOfWords];
signal input sign[rsaNumberOfWords];
signal input modulus[rsaNumberOfWords];

//we need some inputs to verify what is signed
//and that Nin in the right place in the signed message
signal input messageToBeSigned[lengthInBitsOfSignedMessage];
signal input Nin[lengthOfNinInBits];

//TODO: Insert Nin into messageToBeSigned in the right place
//or create constraint on the corresponding bits


component sha = Sha256(lengthInBitsOfSignedMessage);
for (int i = 0; i<lengthInBitsOfSignedMessage; i++) {
  sha.in[i] <== messageToBeSigned[i]
}

var hashed[hashNumberOfWords];//4x64 bit
var hashedBits[hashNumberOfWords*bitsInAWord];
for (int i=0; i<hashNumberOfWords*bitsInAWord;i++){
  hashedBits[i] <== sha.out[i];
}
component b2n = Bits2Num(64);
//TODO:Can we use a component multiple times?
//Will all the constraints be generated
for (int i=0;i<hashNumberOfWords;i++){
  for (int j=0;j<bitsInAWord;j++){
    b2n.in[j] <== hashedBits[i*64+j];
  }
  hashed[i] <== b2n.out;
}


//lets assume sha256WithRSAEncryption
//verify CA signature of Kid
component rsa = RsaVerifyPkcs1v15(bitsInAWord, rsaNumberOfWords, 17, 4);
for (var i = 0; i < rsaNumberOfWords; i++){
  rsa.exp[i] <== exp[i];
  rsa.sign[i] <== sign[i];
  rsa.modulus[i] <== modulus[i];
}
for (var i = 0; i<hashNumberOfWords; i++){
  rsa.hashed[i] <== hashed[i];
}

//TODO: verify Kid(vKid)=="registerKid"

//Check CA on CA approved list
//On blockchain mapping (uint256 => bool) public CAValid


```

```
  //Take Kid
  //Take CA



}
component main { public [hNin,vKid]} = NinOwnsKid();
```

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 if someone tried to register with two different ids, we couldn't tell if Nin was already used. On the other hand, the choice of arguments Nin and $K_{ID}^{-1}("registerK_{ID}")$ would be almost the same as $hNin$ and $K_{ID}^{-1}("registerK_{ID}")$, because you can brute force all the possible Nins and find the corresponding $hNins$.

4 Registering K_P

Register K_P on *Identity Blockchain* by calling

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

where:

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

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

Pseudo-solidity code

```
mapping(uint256 => uint256) public ninKidKpUsed;
mapping(uint256 => bool) public kpInvalid;
mapping(uint256 => uint256) public ninKidKpLastConfirmed;

function registerKp(
  uint256 ninKid,
  uint256 ninKidKp,
  bytes proof
) public {
  uint256 previousKey = ninKidKpUsed[ninKid];
  if (previousKey == ninKid) return; // already registered
  bool ok = zksnarkverify(VK, [ninKid, ninKidKp], proof); // VK = verification key
  if (ok && !kpInvalid[ninKidKp]) {
    if (previousKey != 0) kpInvalid[previousKey] = true;
    ninKidKpUsed[ninKid] = ninKidKp;
    ninKidKpLastConfirmed[ninKid] = block.number;
  }
}
```

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.

Anonimity
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 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: We want to connect Nin with K_P in a unique, but untraceable way. If we only used $NinK_{ID}K_P$ then if someone used multiple K_Ps we wouldn't be able to prove the connection to Nin (i.e. they could register a lot of K_Ps). Hashing connects arguments, and makes them opaque without the knowledge of arguments (which in this case include private keys), so the arguments are opaque without the knowledge of private keys.

Pseudo-circom code Statements we need to check:

- Check first step (Register K_{ID}) still valid

```
//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

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.