Identity Blockchain - Proof of Identity

Albert Vučinović, University North
Miroslav Jerković, University of Zagreb

July 24, 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_H	Human key, unique public key to access *Identity Blockchain*

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

3 Registering K_{ID}

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

$$registerK_{ID}(H(Nin), K_{ID}^{-1}(\text{"}registerK_{ID}\text{"}), invalidateK_{ID}, verifyProofArguments).$$

Pseudo-solidity code

```
pragma solidity ^0.8.0;

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

function registerKid(
  uint hNin, // hNin = H(Nin)
  uint vKid, // vKid = KidPrivate("registerKid")
  bool invalidateKid,
  uint[2] memory proofPointA,
  uint[2][2] memory proofPointB,
  uint[2] memory proofPointC
) public {
  // Setting public arguments
  uint[] memory inputValues = new uint[](2);
  inputValues[0] = hNin;
  inputValues[1] = vKid;

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

  uint previousVKid = ninKidUsed[hNin];
  if (previousVKid == vKid) return; // already registered
  if (previousVKid != vKid && invalidateKid == true) {
    // If they are different, invalidate old Kid
    kidInvalid[previousVKid] = true;
  }
  require(!kidInvalid[vKid]);

  ninKidUsed[hNin] = vKid;
}
```

Features

Sybil resistance
A Person is prevented from registering more than one K_H, e.g. by using different eIDs 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 $H(Nin)$ and $K_{ID}^{-1}("registerK_{ID}")$:

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

Pseudo-circom code

Statements we need to check:

1) CA is accepted by *Identity Blockchain* (assumption: the set representations of *CA-public-keys* and *revoked-CA-public-keys* lists exist on *Identity Blockchain*):

   i) inclusion of the CA in cert in *CA-public-keys*.

   ii) exclusion of the CA in cert from *revoked-CA-public-keys*.

   Verify CA validity with respect to time (maybe on *Identity Blockchain*)!

2) Cert CA signed:

   i) K_{ID}

   ii) x509 Subject that contains Nin in the right place in the subject

   Verify validity of x509 certificate with respect to time!

3) K_{ID} is not revoked by CA (*Identity Blockchain* hosted list)

4) K_{ID} decrypts $K_{ID}^{-1}("registerK_{ID}")$ into clear text "registerKid".

For now, we're using the MIT-licensed Circom RSA signature verify, 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

```
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
  signal 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();
```

4 Registering K_H

Register K_H on *Identity Blockchain* by calling

$$registerK_H(HNinK_{ID}, HNinK_{ID}K_H, K_H, verifyProofArguments),$$

where:

$$HNinK_{ID} = H(Nin, K_{ID}^{-1}("registerK_H"))$$

$$HNinK_{ID}K_H = H(Nin, K_{ID}^{-1}("registerK_H"), K_H^{-1}("registerK_H")).$$

Pseudo-solidity code

```
mapping(uint => uint) public ninKidKhUsed;
mapping(uint => bool) public isPerson;

function registerKh(
  uint hNinKid,
  uint hNinKidKh,
  uint Kh,
  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] = hNinKid;
  inputValues[1] = hNinKidKh;
  inputValues[2] = Kh;

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

  ninKidKhUsed[hNinKid] = hNinKidKh;
  isPerson[Kh] = true;
}
```

Features

Sybil resistance
K_H is a unique public key bound to the Person on the *Identity Blockchain*. By design, the Person cannot have two valid K_H 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_H can tell, from the function call, which Nin, K_{ID} or K_H 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_H in a unique, but untraceable way. If we only used $HNinK_{ID}K_H$ then if someone used multiple K_Hs we wouldn't be able to prove the connection to Nin (i.e. they could register a lot of K_Hs). 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.

<u>Pseudo-circom code</u>

Statements we need to check:

1) $registerK_{ID}$ is still valid
2) $HNinK_{ID}$ is valid
3) $HNinK_{ID}K_H$ is valid
4) K_H is valid
5) TODO: K_H invalidation!?

```
pragma circom 2.0.0;
```

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_H.

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_H without checking the whole $CA \rightarrow K_{ID} \rightarrow K_H$ chain.

i) Using K_H to register as a mining key.
ii) Invalidating K_H.
iii) Using new K_H as a mining key.

<u>Solution</u>

Look at the solution to **P7**.

The Mining Registry can accept new K_H 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_H and registers K_H for mining. The problem is bigger, because we can't invalidate K_H if we don't know which K_H it is.

<u>Solution</u>

We make K_H renewable. We write the last block number it was renewed on to the *Identity Blockchain*. Mining Registry can choose to accept K_H 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_H.

P8 Only K_H is compromised.