Anonymization in MACI

Thanks to @vbuterin for suggesting this idea, and @barryWhiteHat for collaborating.

It's an MPC-less alternative to Adding anonymization to MACI.

Introduction to MACI

We assume a MACI system as described here:

1. Registry R

with registered public keys $K_1, ..., K_n$

that belong to users.

1. Operator O

with with a private key k_w

and public key K_w

.

1. Mechanism $M: action^n \rightarrow Outputs$

The operator O

manages internally a state $S = \{i : (key = K_i, action = \emptyset)\}$

for $i \in 1...n$

. That is, the state has the current public key for each user and the current action the user has chosen.

The system works as follows:

At time T_{start}

, the operator O

has state $S_{start} = \{i : (key = K_i, action = \emptyset)\}$

for $i \in 1...n$

.

Between times T_{start}

and T_{end}

, users publish messages encrypted with the operator's key K_w

.

Users are allowed two types of messages:

1. M_{action}
2. where users wish to change the current action associated their state. Specifically, they publish $enc(msg = (i, sig = sign(msg = action, key = k_i)), pubkey = K_w)$

. The key k_i

is the user's private key and i

is their index in the registry R

.

1. M_{key_change}
2. where users wish to change the current key associated with their state. Specifically, the publish enc(msg = (i, sig =

sign(msg = NewK_i, key=k_i)), pubkey = K_w)

. The key k_i

is the user’s private key, i

is their index in the registry R

and NewK_i

is their new public key.

The operator processes messages in the order they have been published as following:

1. On invalid messages - decryption fails, unknown type or badly formatted message - do nothing.
2. Check the signature inside the message verifies, i.e. verify(sig, msg, state[i].key) == true

. This means that the user’s key in the state matches the key signing the message. If true:

- If the message is of type M_{action}

, set S[i].action = action

.

- If the message is of type M_{key_change}

, set S[i].key = NewK_i

.

1. If the message is of type M_{action}

, set S[i].action = action

.

1. If the message is of type M_{key_change}

, set S[i].key = NewK_i

.

The operator doesn’t publish anything until time T_{end}

, where they then run the mechanism M(state[1].action, ..., state[n].action)

and publish both the output of the mechanism and a zkSNARK proving:

1. Processing happened on the all the published messages in-order.
2. Each processed message was either invalid or the signature didn’t verify - causing no changes in the state, or the message was one of M_{action}

or M_{key_change}

and the appropriate update was applied.

Anonymity problem

Everything is hidden on-chain - only ciphertexts are published by users. The operator, though, sees all the actions taken by each of the keys, as they have to update the state and generate the proof of correctness at the end.

Ideally, we’d like a situation where the operator is responsible only for anti-collusion, and doesn’t know which user took what action.

Solution - Re-randomization

ElGamal Encryption

Given a group G

of order q

and generator g

, we have the following functions:

- KeyGen: $() \rightarrow (x, g^x)$
- generate a private key and its corresponding public key. x

is an integer. g^x

is the public key.

- Encrypt: $(pk, message) \rightarrow (c_1, c_2)$
- encrypt a message under the public key pk

, producing a ciphertext (c_1, c_2)

. Encryption is done by choosing a random integer y

and outputing $(c_1 = g^y, c_2 = m \cdot pk^y)$

.

- Decrypt: $(x, (c_1, c_2)) \rightarrow message$
- decrypt a ciphertext (c_1, c_2)

using the private key x

, producing a message m

. Decryption is done by computing $m := (c_1^x)^{-1} \cdot c_2$

.

We additionally define a re-randomization function:

- ReRandomize: $(c_1, c_2) \rightarrow (d_1, d_2)$
- randomizes an existing ciphertext such that it's still decryptable under the original public key it was encrypted for. Re-randomization is done by choosing a random integer z

and outputing $(d_1=g^z \cdot c_1, d_2 = pk^z \cdot c_2)$

. This essentially produces a ciphertext as if the random integer $z+y$

was chosen, as $(d_1 = g^z \cdot g^y = g^{z+y}, d_2 = pk^z \cdot m \cdot pk^y = pk^{z+y} \cdot m)$

.

Protocol

Let H

be a cryptographic hash function.

The operator publishes an ElGamal public key E_w

with private key e_w

.

The operator manages the following two sets:

1. withdrawn_set
2. Encrypted states for all the keys that have been deactivated, using the message described below. This set has elementes of the form (K_i, enc_active_i)

, where enc_active_i

is an encryption of either ACTIVE

or INACTIVE

under the operator's public key E_w

. This set is public.

1. nullifiers
2. Nullifiers for new keys that were activated from previously deactivated keys, using the message described below. This set is private to the operator.

We add another field to the state of each user - active

, which marks whether the key is active or not. Newly registered keys have S[i].active = true

.

Add two more message types:

1. M_{deactivate_key}
2. where users wish to deactivate their current active key. Specifically, they publish enc(msg = (i, sig = sign(msg = \emptyset, key = k_i)), pubkey = K_w)

. If the request was valid - the signature verifies, the public key corresponds to the current key of the user and S[i].active = true

, the operator adds (K_i, Encrypt(K_w, ACTIVE))

. Otherwise, the operator adds (K_i, Encrypt(K_w, INACTIVE))

.

.

1. M_{new_key_from_deactivated}
2. where users wish to register a new key, given that they deactivated a key before.

First, they generate a SNARK proof \pi

showing that:

- An element (K_i, (c_1, c_2))

exists in the withdrawn_set

.

- They know the private key of K_i

.

- They output (d_1, d_2)

, which is ReRandomize(c_1, c_2)

.

- They output a nullifier H(k_i)

.

- A hash of the following is a public input:
- Commitment to the current state of withdrawn_set

.

- (d_1, d_2)

.

- H(k_i)

.

- Commitment to the current state of withdrawn_set

.

- (d_1, d_2)

.

- H(k_i)

.

Then, they publish enc(msg = ((d_1, d_2), H(k_i), \pi, NewK_i), pubkey = K_w)

, where \pi

is validated against the current withdrawn_set

state commitment.

The operator decrypts (d_1, d_2)

into is_active

.

The operator adds the new key NewK_i

if the proof verifies, the nullifier doesn’t already exist in nullifiers

and sets S[i].active = is_active

. The operator also adds H(k_i)

to nullifiers

. If anything fails, the operator still adds the key but with S[i].active = false

.

1. An element (K_i, (c_1, c_2))

exists in the withdrawn_set

.

1. They know the private key of K_i

.

1. They output (d_1, d_2)

, which is ReRandomize(c_1, c_2)

.

1. They output a nullifier H(k_i)

.

1. A hash of the following is a public input:
2. Commitment to the current state of withdrawn_set

.

- (d_1, d_2)

.

- H(k_i)

.

1. Commitment to the current state of withdrawn_set

.

1. (d_1, d_2)

.

1. H(k_i)

.

Short analysis

Data on-chain:

1. The withdrawn_set

.

1. Proof for every M_{new_key_from_deactivated}

and its public input.

Efficiency challenge - either proving non-membership in the nullifier set is linear, or updating it is linear. This affects proving time, though it is still practical.