For background see https://vitalik.ca/general/2019/04/03/collusion.html

Suppose that we have an application where we need collusion resistance, but we also need the blockchain's guarantees (mainly correct execution and censorship resistance). Voting is a prime candidate for this use case: collusion resistance is essential for the reasons discussed in the linked article, guarantees of correct execution is needed to guard against attacks on the vote tallying mechanism, and preventing censorship of votes is needed to prevent attacks involving blocking votes from voters. We can make a system that provides the collusion resistance guarantee with a centralized trust model (if Bob is honest we have collusion resistance, if Bob is dishonest we don't), and also provides the blockchain's guarantees unconditionally (ie. Bob can't cause the other guarantees to break by being dishonest).

Setup

We assume that we have a registry

R

that contains a list of public keys

K_1 ... K_n

. It's assumed that R

is a smart contract that has some procedure for admitting keys into this registry, with the social norm that participants in the mechanism should only act to support admitting keys if they verify two things:

1. The account belongs to a legitimate participant (eg. is a unique human, is a member of some community as measured in some formalized way such as citizenship of a country or a sufficiently high score on a forum, holds some minimum balance of tokens…)
2. The account holder personally controls the key (ie. they have the ability to print it out on demand if they really wanted to)

Each user is also expected to put down a deposit; if anyone publishes a signature of their own address with the private key, they can steal the deposit and cause the account to be removed from the list (this feature is there to heavily discourage giving any third party access to the key).

We assume that there is an operator

with a private key k_{\omega}

and a corresponding public key K_{\omega}

.

We assume that there is a mechanism

M

, which we define as a function action^n \rightarrow Outputs

, where the input is the action taken by each of the n

participants and the output is some output as defined by the mechanism. For example, a simple voting mechanism would be the function that returns the action

that appears the most times in the input.

Execution

At time T_{start}

, the operator begins with an internal state S_{start} = {i: (key=K_i, action=\emptyset)}

for i \in 1...n

. Anyone can compute this internal state; the registry contract itself could do this.

Between times T_{start}

and T_{end}

, anyone has the right to publish messages into an on-chain registry (eg. set the to

address to be a smart contract that saves them into a linked hash list) that are encrypted with the key k

. There are two types of messages:

1. Actions: the intended behavior for a user is to send the encryption enc(msg=(i, sign(msg=action, key=k_i)), pubkey=K_{\omega})

where k_i

is the user's current private key and i

is the user's index in R

.

1. Key changes: the intended behavior for a user is to send the encryption enc(msg=(i, sign(msg=NewK_i, key=k_i)), pubkey=K_{\omega})

, where NewK_i

is the user's desired new public key and k_i

is the user's current private key.

We assume an encryption function where the user can provide a salt to make an arbitrarily large number of possible encryptions for any given value.

The operator's job is to process each message in the order the messages appear on chain as follows:

- Decrypt the message. If decryption fails, or if the resulting object fails to deserialize into one of the two categories, skip over it and do nothing.
- Verify the internal signature using state[i].key
- If the message is an action, set state[i].action = action

, if the message is a new-key, set state[i].key = NewK_i

After time T_{end}

, the operator must publish the output M(state[1].action .... state[n].action)

, and a ZK-SNARK proving that the given output is the correct result of doing the above processing on the messages that were published.

Collusion resistance argument

Suppose a user wants to prove that they took some action A

. They can always simply point to the transaction enc(msg=(i, sign(msg=A, key=k_i)), pubkey=K_{\omega})

on chain, and provide a zero-knowledge-proof that the transaction is the encrypted version of the data containing A

. However, they have no way of proving that they did not send an earlier

transaction that switched the key to some NewK_i

, thereby turning this action into a no-op. The new key then could have been used to take some other action.

The user could give someone else access to the key, and thereby give them the ability to race the user to get a new-key message in first. However, this (i) only has a 50% success rate, and (ii) would allow the other user the ability to take away their deposit.

Problems this does not solve

- A key-selling attack where the recipient is inside trusted hardware or a trustworthy multisig
- An attack where the original key is inside trusted hardware that prevents key changes except to keys known by an attacker

The former could be mitigated by deliberately complicated signature schemes that are not trusted hardware / multisig friendly, though the verification of such a scheme would need to be succinct enough to be ZKP-friendly. The latter could be

solved with an “in-person zero knowledge proof protocol”, eg. the user derives x

and y

where x+y=k_i

, publishes X = x * G

and Y = y * G

, and shows the verifier two envelopes containing x

and y

; the verifier opens one, checks that the published Y

is correct, and checks that X + Y = K_i

.

Future work

- See if there are ways to turn the trust guarantee for collusion resistance into an M of N guarantee without requiring full-on multi-party computation of signature verification, key replacement and the mechanism function
- MPC and trusted hardware-resistant signing functions.