Shuffling Public Keys
(A Peer-to-peer Voting Algorithm)

Santi J. Vives
jotasapiens.com/research

Abstract: A peer-to-peer, permisionless cryptographic voting system that relies only on the existence of generic digital signatures and encryption.

1. Introduction

Democratic elecions are a a vital decising-making method in modern societies. Despite its importance, elections hold a vulnerability that can bring democracies to a halt. An election cooses between a set of candidates, one which will became an authority of some form. In turn elected authorities (either directly or) indirectly) are in charge of the next elections. That depence on an authority makes it possible for people to elect a malicious authority that will later refuse to organize future elections, impede elections, or comit fraud.

This paper is the result of work aimed to tackle this problem from a cryptographic point of view. Through the sections in the paper, I introduce a voting algortihm that works on a peer-to -peer basis, without relying on any authority to organize the election itself nor count the votes. The algorithm still requires authorities to inscribe voters, creating a public record that pairs persons with their public keys. Once the process starts and keys are known, peocple can cast their votes in a peer-to-peer and permisionless way at any point in time, making the right to vote and voice an opinion harder to take away. In cases where a malicious authority is elected, unwilling to organize a new election, people would still be able to cast their votes as long as their hold their private keys, and are able to comunicate over any public network.

Stating the problem

Imagine a group of persons who want to vote, persons who own cryptographic coins and want to spend them without disclosing who made each payment, or persons that want to say something anonymously. These three situations, though they may appear superficially different, can be described as a same underlying mathematical problem:

A set of persons want to produce a set of shuffled information, in such a way that anyone can verify that each piece of information belongs to one of them, but nobody can tell to which in particular.

We start we a set of keys k_0, k_1, k_2..., each belonging to one person, and they want to generate of shuffled messages m_0, m_1, m_2...
Digital signatures would solve half the problem, allowing to authenticate those messages. But we would like to achieve this in a way that is unfeasible to match the authentication of

each message with any of the original k_n keys in particular.

We will see that this problem can be solved in a distributed maner, relying sole in very simple assumtions: the existence of a generic signature scheme, generic public encryption, any pseudorandom funcion and hash, and any public network to comunicate with each other.

To solve the problem, we will use a divide-and-conquer approach. We will deal with the a simplest case first. Later, we'll break the general problem into many instances of the simplest case.

The work is organized as follows:

Section 2: The problem is solved for the simplest form of 2 participants.

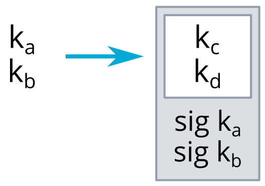
Section 3: Extends the solution to any 2^x number of participants. At this points, only the case of honest participants is considered .
Section 4: Solves the problem of routing secret information through the network.
Section 5: Extends the algorithm to any number of participants.
Section 6: Solves the problem of dishonest participants.
Appendix A: Solves problems specific to voting (casting votes, counting them).
Appendix B: Describes some possible optimizations.

2. Butterfly

We will solve the problem using a divide and conquer strategy, breaking it into many smaller instances. For that, let's start with the simplest case of having only two public keys to shuffle.

Image there is one public key k_a that belongs to Albert, and one public key k_b that belongs to Barbara. They want to generate a shuffled pair of keys k_c, k_d. Except for them, no one should be able to tell which owns each key in the new pair.

Imagine Albert and Barbara have a way of exchanging secret information. Each Albert and Barbara generate a new key. They share their secret keys and put them in random order, creating a shuffled pair of keys k_c, k_d. They can now make the new pair public.

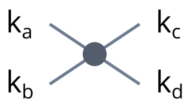


Next, it is time to authenticate the new keys. Albert signs the pair using his old key k_a. Barbara does the same: she signs the pair using k_b.

And that is all. Anyone can verify the new keys are authentic, since the new pair is signed by both Albert and Barbara. But to an outsider there is no way of telling which of the two owns each.

We will refer to this process of shuffling two key as a butterfly, borrowing terminology from the FFT [1], despite we are dealing with a different problem here: shuffling information rather than moving to the frequency domain.

Generally, a butterfly takes two input keys k_a and k_b, and outputs a shuffled pair k_c, k_d



2.1 Implementation

Albert and Barbara have two public keys (k_a and k_b) and they want to produced a pair of shuffled keys k_c, k_d.

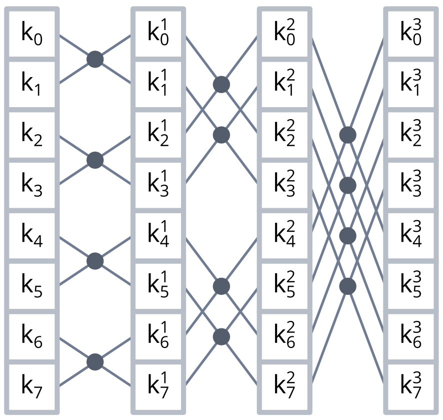
- **Albert** generates a pair of encryption/decryption keys *e, d*. He signs his encryption key *e* with his old signature k_a and makes it public.
- **Barbara** verifies the signature to ensure *e* belongs to Albert.
- Next **Barbara** generates a new verification key k_c, she encrypts it with *e*, signs the encrypted message with her old signature k_b, and publishes the encrypted message.
- **Albert** verifies Barbara's signature, and decrypts the message with his own key *d*. Albert now knows Barbara's new verification key k_c.
- **Albert** generates a new verification key k_d of his own, and keeps it secret.
- Now, **Albert** takes Barbara's new verification key k_c and his own new verification key k_d. He forms a message by appending both keys in random order to form the shuffled pair $k_d \,||\, k_c$. He signs the shuffled pair with his old signature k_a and makes it public.
- **Barbara** verifies Albert's signature, and checks that her new key k_c is in the pair. She signs the pair with her old signature k_b and publishes the signature.

We started with two keys k_a, k_b, and ended with a new pair of public keys k_c k_d. Everyone can verify the new pair is valid, since its signed by the two older keys. But an outsider there is no way of telling who (Barbara or Albert) owns each key.

3. Shuffle all

It's time to solve the larger problem of many voters. That can be done by redefining the larger case as multiple iterations of the simplest case.

In this section we will work with any number of voters of the form 2^x, and consider only what is observed by an outsider. We will study more general cases later on (sections 5, 6)



To begin shuffling, let's take the keys $k_0 ... k_7$ in pairs, and apply a butterfly to each of those pairs. In the diagram the keys we started with are at the leftmost column, and the new keys are in the next.

For this first step, we get the butterflies input pairs:
$(k_0, k_1), (k_2, k_3), (k_4, k_5), (k_6, k_7)$

Keys are now shuffled in groups of two. For the second step, we will take those groups of size 2 in pairs and shuffle them together: the zeroth key of one group with the zeroth of the other, the first key of one group with the first of the other.

In our example, we take keys in groups of size 2:
$(k_0^1, k_1^1), (k_2^1, k_3^1), (k_4^1, k_5^1), (k_6^1, k_7^1)$

And from those groups we get the butterflies input pairs:
$(k_0^1, k_2^1), (k_1^1, k_3^1), (k_4^1, k_6^1), (k_5^1, k_7^1)$

As usual, each butterfly takes two input keys k_a^1, k_b^1 and produces a new pair k_a^2, k_b^2 signed by the input keys.

> Let's not worry for now about how nodes communicate to compute the butterflies. We will study a general method in detail the next section.

After this second step, each resulting key can come from either of two groups, and each key within that group can come from either of two positions. Meaning that each new key is 1 out of 4 shuffled keys.

Now keys are shuffled in groups of four. For the third step we will take the groups of size 4 in pairs and shuffle them together. Since our example has only 8 keys, this means shuffling one half of the keys with the other.

We will apply butterflies as usual: the zeroth key of a group with the zeroth of the other, the first with the first, second with second, third with third.

In our example, we take keys in groups of size 4:
$(k_0^2, k_1^2, k_2^2, k_3^2), (k_4^2, k_5^2, k_6^2, k_7^2)$

And from those groups we get the butterflies input pairs:
$(k_0^2, k_4^2), (k_1^2, k_5^2), (k_2^2, k_6^2), (k_3^2, k_7^2)$

At this point all 8 keys are fully shuffled. Each key obtained from the third step is 1 out of 2 distinct keys from the second. In turn, each of those keys is 1 out of 2 distinct keys from the first, and each of those is 1 of 2 distinct keys from the start. In other words, after 3 steps each key is 1 out of 2^3=8 shuffled keys.

Clearly, we can extend the algorithm to larger power-of-two numbers of keys iterating this same procedure: at each step *s* make groups of 2^{s-1} consecutive keys, and pick the corresponding keys of consecutive groups in pairs to apply butterflies.

Since each step doubles the number of keys shuffled together, we can shuffle *n* keys in $\log_2(n)$ steps.

3.1 Probabilities

We have a series of keys $k_0 \ldots k_7$, and persons $a_0 \ldots a_7$. For any given vote, we will compute the probabilities that it belongs to each of the persons.

Before shuffling starts, we know who owns each key. We know for example that the zeroth key belongs with certainty to the zeroth person, and does not belong to the others.

If we define a vector pk_x as a list of probabilities that key k_x belongs to each of the eight persons, for the zeroth key before shuffling we have:

$$pk_0 = [\ \mathbf{1}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ]$$

That same way, for all keys we have:

$$\begin{aligned}
pk_0 &= [\ \mathbf{1}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ] \\
pk_1 &= [\ 0\ \ \mathbf{1}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ] \\
pk_2 &= [\ 0\ \ 0\ \ \mathbf{1}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ] \\
pk_3 &= [\ 0\ \ 0\ \ 0\ \ \mathbf{1}\ \ 0\ \ 0\ \ 0\ \ 0\ ] \\
pk_4 &= [\ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1}\ \ 0\ \ 0\ \ 0\ ] \\
pk_5 &= [\ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1}\ \ 0\ \ 0\ ] \\
pk_6 &= [\ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1}\ \ 0\ ] \\
pk_7 &= [\ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1}\ ]
\end{aligned}$$

Let's compute now the probabilities after a butterfly. The butterfly takes two keys k_a, k_b as input and gives as a result two new keys k'_a, k'_b. Since for an output key there is half the chance that it corresponds to k_a and half the chance that it corresponds to k_b, we have:

$$\begin{aligned}
pk'_a &= \tfrac{1}{2}\, pk_a + \tfrac{1}{2}\, pk_b \\
pk'_b &= \tfrac{1}{2}\, pk_a + \tfrac{1}{2}\, pk_b
\end{aligned}$$

In the first step, butterflies shuffle even and odd keys: (k_0 with k_1), (k_2 with k_3), (k_4 with k_5), (k_6 with k_7). Using the equations above we get:

$$\begin{array}{l}
pk_0^1 = [\,\mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_1^1 = [\,\mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_2^1 = [\,0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_3^1 = [\,0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_4^1 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\,] \\
pk_5^1 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\ \ 0\ \ 0\,] \\
pk_6^1 = [\,0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\,] \\
pk_7^1 = [\,0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/2}\ \ \mathbf{1/2}\,]
\end{array}$$

Next, the second step shuffles keys that are two positions apart: (k_0^1 with k_2^1), (k_1^1 with k_3^1), (k_4^1 with k_6^1), (k_5^1 with k_7^1). We get:

$$\begin{array}{l}
pk_0^2 = [\,\mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_1^2 = [\,\mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_2^2 = [\,\mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_3^2 = [\,\mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ 0\ \ 0\ \ 0\ \ 0\,] \\
pk_4^2 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\,] \\
pk_5^2 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\,] \\
pk_6^2 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\,] \\
pk_7^2 = [\,0\ \ 0\ \ 0\ \ 0\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\ \ \mathbf{1/4}\,]
\end{array}$$

Finally, the last step shuffles the first half with the second: (k_0^2 with k_4^2), (k_1^2 with k_5^2), (k_2^2 with k_6^2), (k_3^2 with k_7^2), making the probabilities the same for any key and any person:

$$\begin{array}{l}
pk_0^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_1^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_2^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_3^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_4^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_5^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_6^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,] \\
pk_7^3 = [\,\mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\ \ \mathbf{1/8}\,]
\end{array}$$

Shuffling any number 2^x of keys result in the same probability value distributed across all positions. In other words, it is just as likely that any key belongs to any person.

3.2 Finding butterfly pairs

When we have a key we want to shuffle, we need to pair it up with other keys to apply butterflies. We need to compute the positions of those other keys.

We already know that a set of *n* keys require $\log_2(n)$ shuffle steps.

And we know that at step *s* keys are combined in groups of size 2^{s-1}, meaning our key k_a^s lies within group $\lfloor a/2^{s-1} \rfloor$.

If our key is in an even group, our butterfly pair is in the next (odd) group. Instead, if it is in an odd group, our pair is in the previous (even) group.

We also know the other key is placed at the same position within its group. That is, the key is 2^{s-1} positions apart from ours.

Summing up, at step *s* we have a key k_a^s, and we need to find its butterfly pair k_b^s. The value of *b* is:

$$b = \begin{cases} a + 2^{s-1} & \text{if } \lfloor a/2^{s-1} \rfloor \text{ is even} \\ a - 2^{s-1} & \text{if } \lfloor a/2^{s-1} \rfloor \text{ is odd} \end{cases}$$

We can write this as (Python) code:

4. Routing

So far we have learned how to shuffle keys. Still, shuffling requires a method to secretly exchange keys and signatures. We need to devise that method.

Imagine an attacker able to observe all internet connections and all information flowing through the network. If keys and signatures are broadcasted in the clear, the attacker could observe those messages coming from an ip, and link persons with their shuffled keys.

We need a confidential way to broadcast information that is robust even against a see-it-all attacker.

4.1 Background: onion routing

A common method to send anonymous information over a network is *onion routing* [2]. We will describe it briefly. Imagine a set of persons *a, b, c, d*. Each has a known encryption key any other can use to send a confidential message. Then, imagine that *a* wants to send an anonymous message to *d*. Node *a* will chose a path at random, in this example $a \to b \to c \to d$. Node *a* will then encrypt the message using the keys from the path in reverse order (*d, c, b*), and append instructions for each node, so that each one knows who to send the message next.

$$a \rightarrow b \rightarrow c \rightarrow d$$

Node *a* encrypts the message and sends it through the path:

- Node *a* starts by sending the message to *b*.
- Node *b* then removes a layer of encryption and finds instructions to send it to *c*.
- Node *c* receives the message, removes another encryption layer, and finds instructions to send it to *d*.
- Node *d* removes the final encryption layer to discover the original message.

Supposedly, node *b* only knowns an encrypted message traveled from *a* to *c*, node *c* that an encrypted message traveled from *b* to *d*, while node *d* is the only (apart from *a*) that knows the unencrypted message and only knows it came through *c*.

The problem

Consider the see-it-all attacker in this scenario. The attacker can observe an encrypted message going from *a* to *b*, followed by a message from *b* to *c*, and then by message from *c* to *d*. By relating the messages and their timing, the attacker can deanonymize the full path to discover it was *a* who send a message to *d* [3].

4.2 Shuffled routing

This section introduces a variant of onion routing to solve this problem. The aim is to introduce ambiguity against a see-it-all attacker, and internal randomness to prevent malicious participants from deciding upon the paths taken by others.

Imagine two persons Albert and Barbara (*a* and *b* in the diagram) that want to send some messages. In onion routing each would chose a random path. Let's make this the shortest possible for this purpose. In the illustration Albert picks the path $a \rightarrow c \rightarrow e$ and Barbara the path $b \rightarrow d \rightarrow f$

$$\begin{tikzcd}
a \arrow[r] & c \arrow[r] & e \\
b \arrow[r] & d \arrow[r] & f
\end{tikzcd}$$

In the shuffled variant, both paths share and intermediate node *c* (Constantine), predefined to the network. There are two options only: either Albert communicates to *e* and Barbara to *f*, or destinations are swapped.

$$\begin{tikzcd}
a \arrow[dr] & & e \\
 & c \arrow[ur] \arrow[dr] & \\
b \arrow[ur] & & f
\end{tikzcd}$$

The order of the paths is known to Albert, Barbara and Constantine. The channel is in charge of routing messages, and demonstrating the order is truly random. Here by random we mean unpredictable to everyone, including the three of them.

First the three collaborate to discover the random path, as we will see next. Once the path is settled, Albert and Barbara send their messages the same way they would in onion routing. Imagine, for example the paths are:

$a \rightarrow c \rightarrow f$
$b \rightarrow c \rightarrow e$

Constantine waits for Albert's and Barbara's encrypted messages. Once both arrive, he removes the outer layer of encryption, places the messages in order, and publishes them: one message encrypted for *e*, the other for *f*.

The channel introduces ambiguity against the see-it-all attacker. An observer cannot tell whether Albert communicates to *e* and Barbara to *f*, or destinations are swapped [4].

The randomness in the path adds protection against internal attackers. This is important since the path of one sender is dependent on the path of the other. Making paths random (not chosen at will) ensures that no participant can decide upon the path of others.

By including many channels we can introduce ambiguity and randomness to the entire network. But first, there are condition to consider in more detail: randomization, timing, encryption, size, and message authentication.

Randomization

That random order in the path is generated from two numbers: one picked by Albert, the other by Barbara. They commit to their picks, before revealing them. The channel feeds the numbers to a pseudorandom function (PRF) to discover the order.

1. Albert picks a random number *ra*. Barbara picks *rb*.
2. They commit to those number without revealing them. As a commitment, Albert and Barbara send a proof of the numbers to the channel (for example, a hash). The channel makes the commitments public.
3. As soon as each sees the other commitment, they can disclose *ra* and *rb* to the channel.
4. The channel hashes *ra* and *rb* and compares the result against the commitments.

At this point we have two random numbers *ra* and *rb*, each generated independently of the other.

5. The channel feeds *ra* and *rb* to the PRF to generate a new pair, *ra'* and *rb'*:
   ra' = PRF (ra || rb || 0)
   rb' = PRF (ra || rb || 1)
6. If *ra'* is smaller, Albert takes the top outputs. If *rb'* is the smallest one, Barbara takes it instead:
   - **ra' < rb':** top to top, bottom to bottom
   - **ra' > rb':** top to bottom, bottom to top

7. The channel proves the order by disclosing *ra* and *rb*:
   - *ra* encrypted for Barbara,
   - *rb* encrypted for Albert.

Timing

If one message would travel first (say from *a* to *e*) and the other second (*b* to *f*) their timing would reveal their paths. To avoid revealing any information from their timings, the channel

- receives a message (from either *a* or *b*),
- waits for the other,
- shuffles them, and only then publishes the two together.

Encryption

Given a public key it is not possible to decrypt a message, yet one can take any message and encrypt it. If encryption is done naively, one could take the decrypted messages the channel outputs and re-encrypt them to reveal the shuffling order. For the channel to be secure, it must be impossible to recreate the same encrypted messages.

To encrypt a message *g* for someone *c* we encrypt it with a random key *r*, then encrypt *r* using *c*'s public key. We send both encrypted *g* and encrypted *r*.

Then *c* decrypts *r*, uses *r* to decrypt the message *g*, and discards *r*. Since *r* is unknown, reencrypting the same way is not possible.

Size

Additionally, messages must not be distinguishable by their sizes. All communications must be routed in packets of a fixed size.

Message authentication

To avoid messages from being replaced they need to be authenticated.

After committing to the shuffling order, the first messages sent by nodes *a* and *b* will be a pair of shuffled public keys . The keys must used to authenticate every message that follows.

From that point, each Albert and Barbara send a series of messages:
g_0, g_1, g_2, g_3...
h_0, h_1, h_2, h_3...

For each message, they will include its index, sign the message (with k_g or k_h), and encrypt it for the channel.

The channel will broadcast pairs of messages with the correct matching indexes, one signed by k_g, the other by k_h, and the pair signed by the channel.

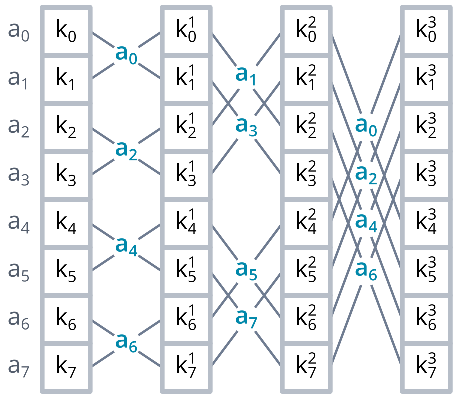
As an additional step, nodes could avoid the chance of messages reaching one destination and not the other:

- The two receiving nodes (*e* and *f*) can communicate with each other, so that they can share the messages as soon as any of them receives it.
- Messages can be included in a decentralized public database, making them available to any node to see.

4.3 Implementation

Now we have the basics, it is time to combine the shuffle algorithm with routing.

We will assign a channel to each butterfly and put voters in charge, so that paths are hidden from the see-it-all attacker. The number of channels needed for each step is half the number of voters. We assign even voters to odd steps (steps 1 and 3), and odd voters to even steps (step 2).


a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7
a_0 a_2 a_4 a_6
k^1_0 k^1_1 k^1_2 k^1_3 k^1_4 k^1_5 k^1_6 k^1_7
a_1 a_3 a_5 a_7
k^2_0 k^2_1 k^2_2 k^2_3 k^2_4 k^2_5 k^2_6 k^2_7
a_0 a_2 a_4 a_6
k^3_0 k^3_1 k^3_2 k^3_3 k^3_4 k^3_5 k^3_6 k^3_7

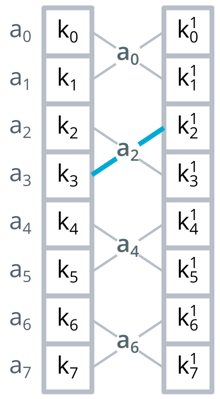
Start

To start, voters must broadcast the encryption keys needed for routing.

Each person a_x generates an encryption key e_x, signs it with key k_x, and makes it public. Every time a message goes through a channel handled by a_x it will be encrypted with e_x.

Step 1

Let's image we are voter a_3. In this first step we shuffle our key k_3 with key k_2, owned by a_2. Here, voter a_2 is also in charge of the channel.



Remember routing a butterfly involves four tasks:

- committing to random numbers,
- disclosing the random numbers to find out the shuffling order,
- broadcasting the new keys (public key k' and its associated encryption key e' to receive messages, with e' signed by k'),
- and signing the pair of shuffled keys.

We will use the notation "m ⊗ e,k" to denote a message *m* encrypted with *e* and signed with *k*. In the example, we broadcast messages *m* signed with our key k_3 and encrypted for a_2 (using e_2). We have:

$m \otimes e_2, k_3$

We start by send a message m with the commitment. Voter a_2 generates the other commitment. Since she is the channel, she removes the outer layer of encryption and makes both public.

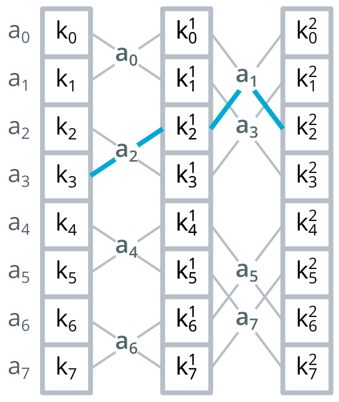
As soon as we see the commitments, we can send a message with the random number and our new key. Channel a_2 broadcasts the pair of shuffled keys in the clear. The random numbers that determine the shuffling order, in contrast, must be encrypted. She broadcast her random number encrypted for us to see, using e_2.

All left is to complete the tasks is sending a message with the signature of the shuffled pair.

Step 2

Once every voter finishes the first step, we can begin with the next. Our last key k_2^2 ended at the top position in the previous butterfly, and is now headed for a_1.

At this step, we shuffle k_2^1 with k_0^1. Our messages must go through two channels: a_2, and a_0.



We encrypt messages n reverse order, so that channel a_0 removes the outer layer of encryption first, and a_2 later. Encrypting and signing a message *m* we get:

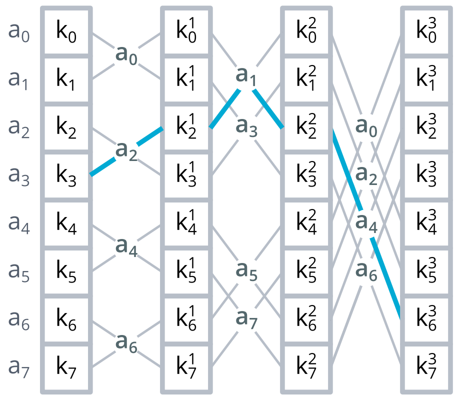
$(m \otimes e_1,k_2^1) \otimes e_2,k_3$

As before, we route the messages to complete the four tasks in the butterfly.

> Notice that at this step we (a_3) are in charge of a channel: that means we are in charge of removing the outer layer of encryption, shuffling information and sending it its way.

Step 3

At step 2, our last key k_2^2 ended at the bottom of the butterfly in a path that leads to channel a_4.



For the third and last step we need to route our messages through three channels: a_2, a_1, and a_4. If we encrypt and sign our messages *m* in reverse order, we get:

$((m \otimes e_4,k_2^2) \otimes e_1,k_2^1) \otimes e_2,k_3$

We are now ready to route the messages for our last butterfly.

And that is all. We started with key k_3 and discovered a routing path that leads to the shuffled key k_6^3.

5. Arbitrary number of voters

We have only considered the number of voters to be a power of 2. Now we will learn what happens for an arbitrary number.

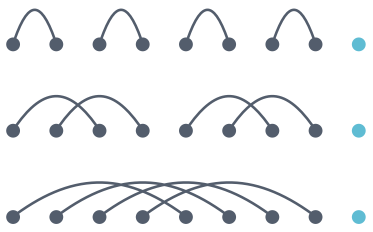
In order to perform all butterflies in the first step, *n* must must be a multiple of 2 (see section 3). The second steps requires *n* to be a multiple of 4. For the third, *n* must be a multiple of 8. Generally, all butterflies pairs exist if *n* is a multiple of 2^{step}. If not, at least one of the keys would find that its pair is out of range.

For an arbitrary number of voters *n*, we will compute pairs as usual, and implement all butterflies that are possible: that is, whenever a pair exists within the range *0...n-1*. For example, for n=7 we get $\lceil \log_2 n \rceil = 3$ steps, and the following butterfly pairs:

step 1: (0,1), (2,3), (4,5) - 6 remains, since its pair (7) is out of range
step 2: (0,2), (1,3), (4,6) - 5 remains
step 3: (0,4), (1,5), (2,6) - 3 remains

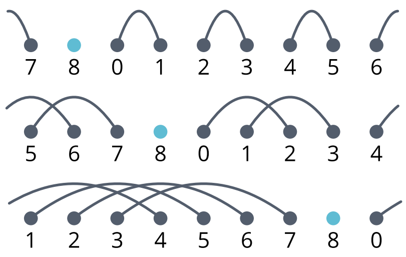
Since some keys might not participate in all of the steps, they would end up shuffled only in part. These keys will have a probability a bit higher than ideal, but we will see in the computed probabilities below that the error is tolerable.

There is still a scenario we need to consider carefully. For some *n* (eg, $n=2^x+1$) it is possible for a key to remain unshuffled on most step. The illustration shows an example with *n=9*. Here, rows from top to bottom indicate steps, and lines connect butterfly pairs. SInce 9 is neither a power of 2, 4, nor 8, the last key remains entirely unshuffled.

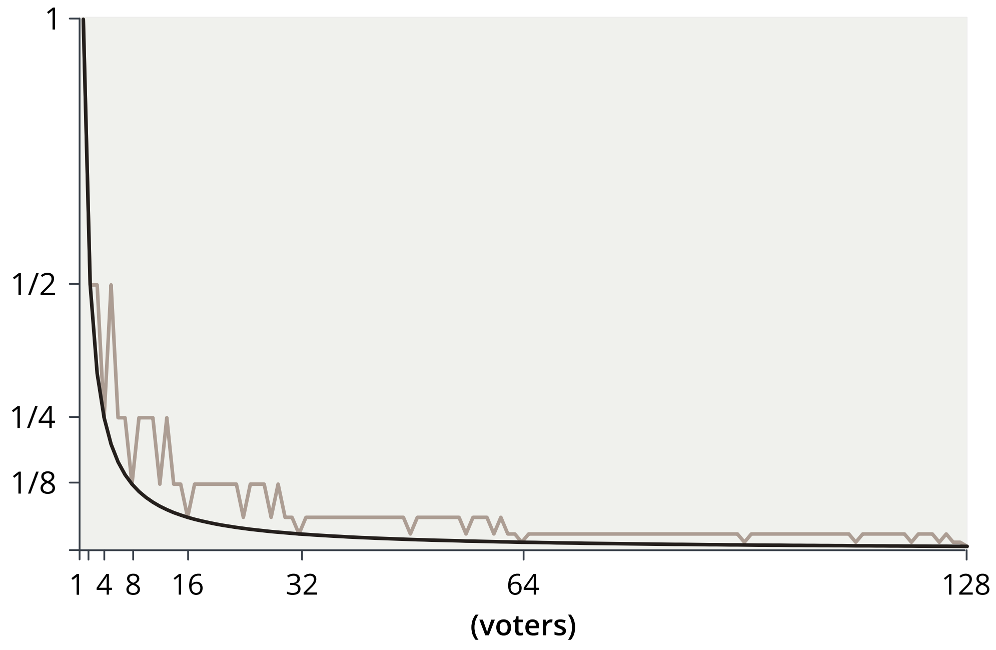


We want to prevent situations like this. To achieve that, we will shift the positions while computing pairs, differently for each step. We will shift keys *2, 4, 8...*2^{step} positions, so that those keys that remain unshuffled are located at different groups of butterflies across steps.

Generally, while for a number of voters *n* at a given step, each position *i* will be shifted to $(i + 2^{step}) \bmod n$



The following plot shows the worst-case probabilities after shifting and shuffling for *n=1...128* (see below a link to the code used to compute the probabilities). The dark curve shows the ideal probability *1/n* (the smallest possible) that a given key belongs to a given voter. The lighter curve shows the worst-case (highest) computed probabilities. The lighter curve only matches the ideal value when *n* is of the form *2^x*. For other cases the curve is not an exact match, but it approximates the ideal curve with a decreasing error as *n* gets larger.



We will learn next that we can improve the approximation as much as we want by running multiple rounds of the shuffling algorithm.

6. Rounds

So far we have consider the perspective of an outsider, and assumed all participants to be honest. It is time now to consider what information voters know, and that some of them might be malicious.

We'll see that those cases results in a shuffle that is imperfect that we can improve as much as we like by running many rounds of the algorithm.

6.1 Missing butterflies

Imagine a voter that wants to compute the probability that keys belong to each other voter. Probabilities after a butterfly are already known from section 3.1, but that equation applies to an outsider only. We now need to compute probabilities considering the point of view of those taking part in the shuffle.

Three persons participate in a butterfly: the owners of two the keys, and the channel. All three know the order within the butterfly. That means, for them the probabilities remain the same.

From that principles, the probabilities can be computed for what a voter knows. Like everybody else, a voter starts knowing the owner of each key with certainty, and can compute the probability vectors after a butterfly as follows:

- If the voter is neither the a owner of an input key nor the channel, the equation in section 3.1 applies.
- If the voter either owns one of the two keys or is the channel, probabilities are left unchanged.

For example, voter *a0* can obtain the following probabilities after a full shuffle:

$$
\begin{array}{lcl}
pk_0^3 & = & [\ \mathbf{1}\quad 0\quad 0\quad 0\quad 0\quad 0\quad 0\quad 0\ ] \\
pk_1^3 & = & [\ 0\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ] \\
pk_2^3 & = & [\ 0\quad 0\quad \mathbf{1/4}\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ] \\
pk_3^3 & = & [\ 0\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ] \\
pk_4^3 & = & [\ 0\quad 0\quad 0\quad 0\quad \mathbf{1/4}\quad \mathbf{1/4}\quad \mathbf{1/4}\quad \mathbf{1/4}\ ] \\
pk_5^3 & = & [\ 0\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ] \\
pk_6^3 & = & [\ 0\quad 0\quad \mathbf{1/4}\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ] \\
pk_7^3 & = & [\ 0\quad \mathbf{1/4}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\quad \mathbf{1/8}\ ]
\end{array}
$$

A voter knows in all cases her own key with certainty, and will of course know with certainty her key does not belong to any other (colored). And ideally, that voter should know nothing about other keys. That means, for *n* other voters she should relate any other key to a any other voter with probability 1/*n*.

Instead, we can observe that the voter is certain that some keys do not belong to some voters, which in turn means some (key, voter) pairs have higher probabilities. Ideally,the voter should link a key to a voters with probability 1/7≈0.14, but we see that in the worst cases the voter can relate some with a higher probability of 1/4=0.25.

This same situation applies to malicious participants as a more general case.

Let's suppose a scenario with a group of *m* attackers, and *n* honest participants. Attackers act like a single front, sharing with each other information they should keep secret.

As a second scenario, imagine attackers don't comply with their tasks. Imagine that their is a timeout for each butterfly. If an attacker is one of the three persons in a butterfly, that butterfly never gets computed, and needs to be bypassed.

Since in both scenarios attackers are not truly participating in the shuffle, the probabilities are updated according to the equation (3.1) if neither of the participants of a butterfly is an attacker, and remain unchanged otherwise.

Then, for a butterfly taking two keys with probability vectors *pa* and *pb*, involving two owners *a* and *b*, and a channel *c*, the output probabilities are:

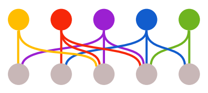
$$
pk'_a,\ pk'_b = \begin{cases} \frac{1}{2}pk_a+\frac{1}{2}pk_b\ ,\ \frac{1}{2}pk_a+\frac{1}{2}pk_b & \text{if neither } a,\ b, \text{ nor } c \text{ is an attacker} \\ pk_a\ ,\ pk_b & \text{if either } a,\ b, \text{ or } c \text{ is an attacker} \end{cases}
$$

This scenarios will result in probabilities deviating from what is ideal. We will see next how probabilities improve by running multiple rounds.

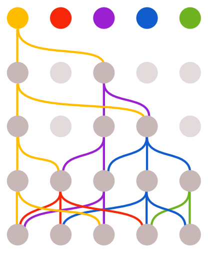
6.2 Probabilities after many shuffles

Suppose you are holding a deck of cards in your hands and you shuffle it. If the process were perfectly random, every card would be just as likely to end up at any position. But that is unlikely to be the case in practice. The shuffle is an imperfect process, only random to a certain degree. Still, you can make the order more and more random by shuffling multiple times [5][6].

Let's build a model to understand what goes on during multiple rounds of shuffling. In this model, at each round, a card can either stay in place or move to a few other random positions. Let's also say that those random positions vary between rounds.



In the example, the yellow card can stay at the first position or move to the third position after a round.



Multiple rounds can be though as a random walk where a card moves to one out of a set of possible places. In the example, it takes four rounds for the path to reach every possible outcome.

Since at each round a card can either stay or move to some random positions, the possible outcomes can only grow with more rounds. Therefore, making every outcome possible is a matter of performing a sufficient number of rounds.

Implementation

Let's design the rounds of the algorithm to behave like the model. The two already agree in the sense that a key (or card) can either stay in place or move to random positions in a single shuffle. The next requirement is for those positions to differ across rounds, which is not met yet.

The final positions in the algorithm come from two sources: the butterflies, that are already ensured to be random (4.2); and the positions of the keys at the start. To make sure the final positions vary between rounds, we can randomize initial positions before a round starts. This ensures butterflies are not performed between the same pairs of voters every time.

At the beginning of each round, voters commit to secret random r_n values as in 4.2. Once all commitments are public, they reveal their values. To compute the ordering, we hash all of the values to obtain a value *h*.

h_n = hash (r_0 || r_1 || ...)

For each key k_n we compute a new value
r'_n = hash (h || r)

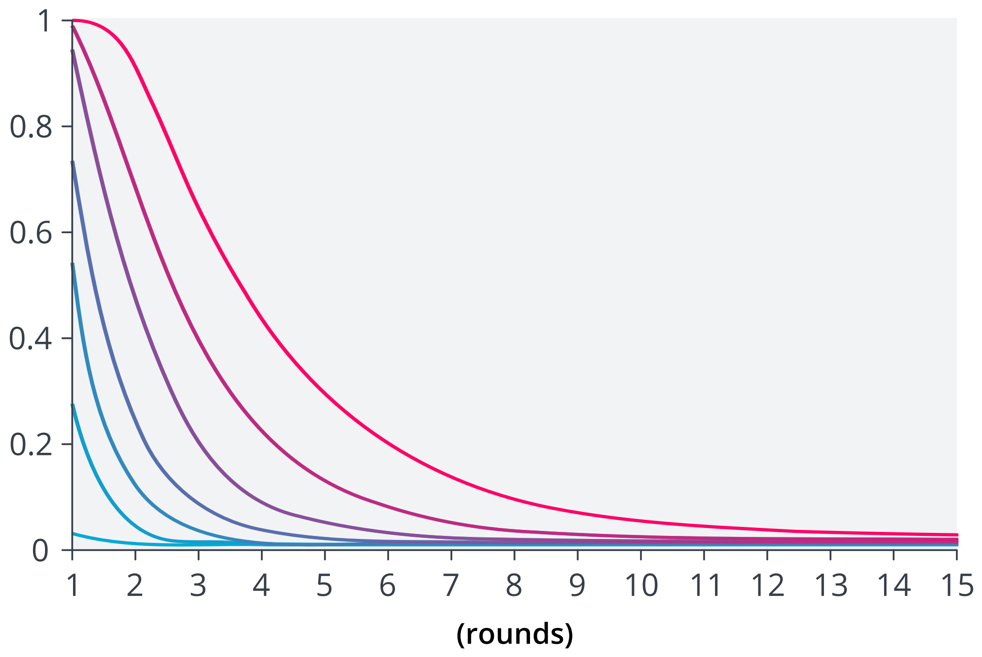
We place all keys ordered according to r'$_n$, and perform a new shuffle round. With each new round, we reorder the keys and shuffle them again. The paths in each round must extend the paths of the ones before. This is important. Otherwise, restarting from scratch would reconnect keys with the persons who own them, undoing all previous shuffles.

Calculations

The probabilities after multiple rounds are easier to approximate empirically, using the equations in 6.1.

Let's say there are *n*+*m* voters, *n* honest ones and *m* attackers. Between the voters, *m* random ones are picked to become attackers, then probabilities are computed as usual.

Attackers will always know the *m* keys they own, and shouldn't know the owner of any honest key. What we are interested in is the worst-case probability. That is, the best match (highest probability) they can make between one of the *n* honest voters and on of their *n* keys.



The plot shows the worst expected probabilities for *n*+*m*=100 total voters, and different percentages of attackers: 0% (cyan), 10%, 20%, 30%, 40%, and 60% (magenta). A link to the code used in this calculation is available below.

Lower is better: a probability of 1 means fully unshuffled, while a probability of 1/*n* means fully shuffled.

As expected, all probabilities tend to the ideal value *1/n*. It just takes more rounds as the number of attackers gets large relative to the number of honest voters.

Appendix A: voting

At this point keys are shuffled and we know how to send information through the network. It is time to cast our vote. Let's see how.

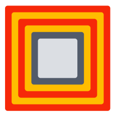
As a thought experiment, imagine we write our vote, sign it with our shuffle key, and send it. The vote travels though the channels. The last channel is the first to see the vote decrypted. Imagine a malicious person in charge: he could reveal the vote if he agrees, and withhold it if he dislikes it.

We need a way of casting and counting votes that does not allow anyone to censor them by looking at their content.

In this section, we'll solve the problem with a game of boxes. We will describe the rules of the game first, and its cryptographic implementation later.

A.1 Boxes

In the game there are yellow and red boxes, and two candidates, named Yellow and Red. Only Yellow knows how to open yellow boxes. And only Red knows how to open red boxes. A vote consists of an unknown number of boxes, one inside the other, alternating colors.



Imaging the box outside is red. Red starts, and opens the box to find a yellow box inside. It is Yellow's turn now, who open that yellow box to find a red one inside. The two candidates keep taking turns, opening red and yellow boxes. At some point, one of them finds a box of neither of the two colors, impossible to open. If a candidate fails to open the box on its turn, the other wins.

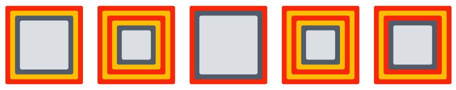
By looking at a box from the outside there is no way of telling who would win the vote. Both candidates must collaborate to discover. A box can lead to two possible outcomes. In the first, the candidate opens it and continues playing with a 50% chance of winning and a 50% chance of losing. In the second, the box is not opened, either because the candidate can't (loses) or won't (quits), resulting in the other winning. It is only after taking turns until the end of the game that we discover the winner.

Now that we understand the rules, we know how to generate a vote. Let's say we prefer Yellow over Red. We know that in order to win, Yellow must be the last to open a box. We create an unopenable box, and wrap it with a yellow box. Then, we put the yellow box inside a red one, into a yellow one after that. We keep alternating colors a random number of times, on the only condition that the last box is of a predetermined color (let's say red) common to all votes.

Once the vote is ready, we sign it with our shuffled key, and broadcast it via shuffled routing.

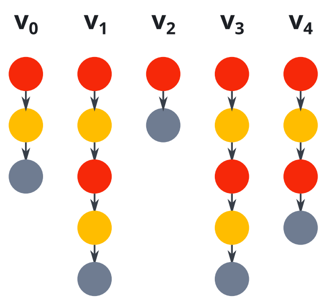
Election

Let's go through an election to visualize the game at work. Voters (5 in the example) pick their favorite candidate (Red or Yellow) and broadcast their votes.



From the outside, all boxes are of a same predefined color (red in the example). Red starts opening boxes, shows the content of the 5 boxes and signs it. It is Yellow's turn now.

Yellow is able to open 4 only, losing the other. Yellow shows the content of those 4 boxes, signed. The two keep taking turns opening some, losing some, until there are no more left to open.



red opens 5, **loses 0**

yellow opens 4, **loses 1**

red opens 3, **loses 1**

yellow opens 2, **loses 1**

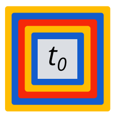
red opens 0, **loses 2**

Through the game, Red has lost 3 boxes while Yellow lost 2. Put in other words, Yellow won 3 boxes and Red 2, making Yellow the winner of the election.

A.2 Boxes, and multiple candidates

Moving from two to more candidates things get slightly different. In case of three candidates or more, one of them losing or quiting does not reveal which of the remaining is the winner. Winning, losing and quitting demand some disambiguation.

Let's start by constructing a vote. Imagine there are three candidates to choose from: Blue, Red, and Yellow. And we want Yellow to win.



In this case, we will state explicitly the winner of our vote. We will generate a message (*t* in the illustration) stating that we pick Yellow, and put the message inside a yellow box. We want the winner to be the last to open the box. Otherwise, if one candidate opens a box and discovers the result picking other as the winner, he might refuse to show that result.

Next, we will put that yellow box inside a box of a random color, then inside another random color, on the condition that we don't pick the same color consecutively. We will repeat this process a random number of times, making sure each color is picked at list once. Will not enforce any particular color for the outer box.

When a candidate opens a box, it can result in two possibilities:

- the candidate finds the result and wins that vote,
- or the candidate finds another box and continues playing with the same chance of winning as any other candidate.

As another possibility, a candidate might not open the box. That could happen for two reasons:

- the candidate can open it but refuses (quits),
- or the candidate cannot open it because the box is unopenable (the vote is invalid and counts as null).

We need to disambiguate these last two possibilities. To do so, let's describe what creating and opening boxes means in more detail.

Let's imagine someone wants to create a red box, only openable by Red. He creates a red lock, and a red key that opens that lock. Then he locks the box, encrypts the red key with Red's public encryption key (so that only Red knows it), and signs the box (so that anyone knows he created it).

Every time Red gets a valid red box, she can open it with the red key. In cases where the box is invalid (unopenable), Red must prove it. Remember no one knows the red key, since it is encrypted. But if Red reveals the red key, we can in fact check whether it's the same she received, and see for ourselves if the vote is invalid: we cannot decrypt the message, but if we receive the unencrypted message we can re-encrypt it and check if it matches.

To sum up, when faced with of box, a candidate must either open the box and show its content, or prove it is unopenable by disclosing the key. Failing to comply implies quitting the election.

Election

Let's see an election in action. There are too many voting systems to consider, but we will study one example. The same principles can be applied to implement other voting systems.

Let's define an example with five voters $v_0...v_4$ and three candidates Red, Blue, and Yellow. In the election, each voter assigns a score to each of the candidates:

- **+1** for the preferred candidates,
- **-1** for the disliked candidates,
- and *0** if indifferent about a candidate.

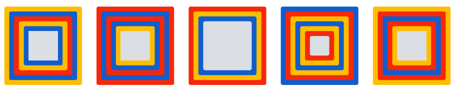
The candidate that is preferred by the most voters (the one with the higher score) wins the election.

Let's imagine we are voter v_2, we like Yellow and Blue, and we dislike Red. We will secretly write our preferences in a tuple: Blue, Red, Yellow = (+1, -1, +1)

In the same way, every other voters will secretly score the candidates:

- $\boldsymbol{v_0}$ (+1, +1, 0)
- $\boldsymbol{v_1}$ (0, -1, +1)
- $\boldsymbol{v_2}$ (+1, -1, +1)
- $\boldsymbol{v_3}$ (0, +1, -1)
- $\boldsymbol{v_4}$ (-1, 1, 0)

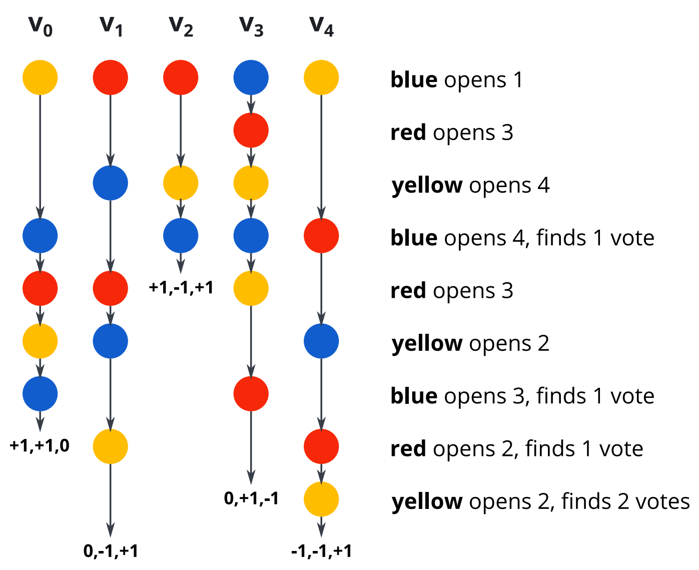
To create the vote, we will wrap the tuple with a box. We will pick a color corresponding to a candidate with the highest score. Since we gave a +1 to both Blue and Yellow, we can pick either. In the example, we wrap the result in a blue box. We place that box inside boxes of random colors as usual.



Once all voters broadcast their votes, candidates can start taking turns opening them.

In case one candidate quits (refuses to either open a box or prove it is unopenable) ththeat candidate is removed from the election, along with any other candidate that no longer holds a change to win. Once removed, all voters cast their votes again between the remaining candidates. Until one wins.

To open boxes, candidates take turns in any predefined color sequence. Let that be Blue, Red, Yellow, B, R, Y, B, R, Y...



- Blue starts. There is only one blue box. Blue opens it, and finds a red box inside.
- It's Red's turn now, and there are 3 red boxes. Red opens them and finds 2 yellow, 1 blue.
- Yellow comes next. There are 4 yellow boxes now. Inside, 3 turn up blue, the other red.
- It's Blue's turn again, and there are 4 blue boxes. After opening them, Blue finds 1 vote (+1,-1,+1), 2 red boxes, and 1 yellow.

  ...

Blue, Red, and Yellow keep taking turns until all tuples are found. Eventually, the reveal the five votes containing the scores for Blue, Red, Yellow: (+1,+1,0), (0,-1,+1), (+1,-1,+1), (0,+1,-1), and (-1,-1,+1).

	🔵	🔴	🟡
v_0	+1	+1	0
v_1	0	-1	+1
v_2	+1	-1	+1
v_3	0	+1	-1
v_4	-1	-1	+1
	1	**-1**	**2**

Now that we have all votes, we can sum the scores each candidate received:

- **Blue** adds up to 1 point.
- **Red** adds up to -1 points.
- **Yellow** adds up to 2 points.

That makes Yellow the preferred candidate and the winner of the election.

A.3 Implementation

Now we have established the rules of the game, let's move to a cryptographic implementation.

Building boxes

A vote consists of a set of boxes one inside the other. As candidates open those boxes, they need to show what it is inside and prove the content is authentic. We can implement this using digital signatures.

Let the outer box be a public key b_0. And let be the content of the box be a message signed by that key, so that anyone can verify its authenticity. Since the content of a box is typically another box, that signed message needs to be another public key b_x. We have, starting from the outside, the set of boxes:

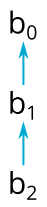
- **box 0**: public key b_0
- **box 1**: public key b_1 signed by b_0
- **box 2**: public key b_2 signed by b_1
- ...

An efficient way to authenticate messages known in advance is by using hash functions. A hash is a type of one-way function: if we feed the function with an input value *x* we can compute the output value *y*, but if we only know the output value *y* it is impossible to compute a valid input *x* (the cost and time of the computations exceed what is physically possible).

Suppose we want to authenticate a message *x*. We will feed it to the hash function to obtain a value *y*:
y = hash (x)

The value *y* will be the public key. Every time someone wants to authenticate message *x*, he will feed it to the hash function and check that the result matches *y*,

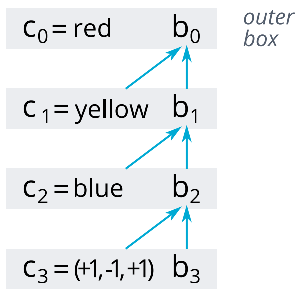
Moving to the boxes: we generate the inner box by picking a big number b_x at random (b_2 in the illustration), and apply the hash function multiple times to obtain boxes b_{x-1}, b_{x-2}... to the outer box b_0. Each value b_x behaves both as a signed message proving it is the content of the box outside, and as the public key of the box inside.



$$\begin{tikzcd} b_0 \\ b_1 \arrow[u] \\ b_2 \arrow[u] \end{tikzcd}$$

Colors

To serve as votes boxes must have colors. We will append a value c_x indicating its color to each b_x value. In the inner box, that value will be the actual result of the vote. We will feed all values of a box (*c, b*) to the hash function to obtain the next *b* value.

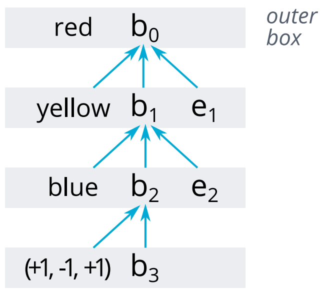


In turn, each box is authenticated by feeding all its values (*c, b*) to the hash function and checking that the result matches the *b* value of the box outside (the one above in the diagram).

Secrecy

The colors and the result must remain secret, and be revealed one at a time as boxes are opened. That means each box must only be read by a specific candidate, and even that candidate shloud only know its content after the box outside is revealed.

First of all, we will append to each intermediate box a random big number e_r, and feed it to the hash function together with the rest of the values.



We will later use each e_x to encrypt values in the box inside (c_{x+1}, b_{x+1}, e_{x+1}...). Value e_x will be used as a symmetric key (meaning the same key is used to encrypt and decrypt). That way, the content of each box can only be decrypted once the box outside is revealed.

> Notice we might drop *e* values and use *b* values as encryption keys instead. We introduced *e* to keep the content of each box secret until the outer box is revealed, and that condition is satisfied by encryption alone. But in doing so the hashes and encryption would not longer be independent, and their properties would need to be considered carefully.

Finally, each box is targeted to a specific candidate. Candidates have public encryption keys, and we need to encrypt each box using the corresponding key.

In our example, the outer box (c_0=*red*, b_0) is red, meaning that its content (box a_1=*yellow*, b_1, e_1) must be encrypted with Red's key.

Inside the red box is a yellow one, meaning that the next box (c_2=*blue*, b_2, e_2) must be encrypted with Yellow's key.

And since that last box (c_2=*blue*, b_2, e_2) is blue, the box inside holding the result (c_3=*[+1,-*

1,+1], b_3) must be encrypted with Blue's key.

Generally, the color a_x of a box determines the target candidate of the box inside (c_{x+1}, b_{x+1}, e_{x+1}).

> To ensure all information in a box remains encrypted till the outer box is revealed, it is better to encrypt the box with the candidate's key first, and with the *e* value after.

Padding

By looking at the vote, it shouldn't be possible to tell the number of boxes are inside. We need to append a random padding to the boxes values, so that all votes have a constant size.

Authentication

Finally, the full vote must be signed by the shuffled public key of the voter.

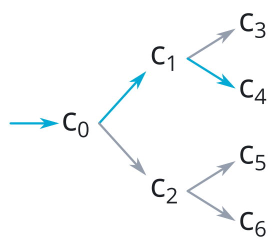
A.4 Vote, and vote again

In short, at every election two sets of keys should be shuffled: one to cast current votes, and other to prepare the next election.

Appendix B: speedup

In this sections, we will focus in optimizing computations. Rather than one step at a time, we can compute multiple steps together by precomputing the different possible routes a message could take.

Let's draw a tree with the possible paths a key can take from the start, a tree of any depth we like. In the illustration, values c_n denote the channels in charge of each butterfly.



The voter routes information of the first butterfly to channel c_0. The output of that butterfly fork into two possible paths, which fork again.

Rather than waiting for each butterfly to reveal the next path, we can route one different value for each possible path. And as each butterfly completes, the channels can discard values from the paths not taken.

Implementation

Let's recall the tasks needed for each butterfly:

1. Commit to a random value that helps choose the shuffling order.

2. Reveal that random value.
3. Send a new public key.
4. Send a signature for the pair of shuffled keys.

Task 1, part a: sending the message

The first task is the commitment. We need to commit to an unique random value m_n for each channel c_n. We also include encryption keys in this first messages, so that channels can communicate things to us.

Let e_n be the public key to encrypt a message for c_n: We start by the last messages, encrypting each for its corresponding channel:

- $m_3 \otimes e_3$
- $m_4 \otimes e_4$
- $m_5 \otimes e_5$
- $m_6 \otimes e_6$

The first pair is routed through c_1, and the other thorough c_2, ao we append the messages meant for c_1 and c_2 and encrypt:

- $(m_1 \,||\, m_3 \otimes e_3 \,||\, m_4 \otimes e_4) \otimes e_1$
- $(m_2 \,||\, m_5 \otimes e_5 \,||\, m_6 \otimes e_6) \otimes e_2$

Finally, the whole is routed through channel c_0. So we append the message and encrypt it. We complete the full packet by adding our signature. The message is now ready:

- $(m_0 \,||$
  $(m_1 \,||\, m_3 \otimes e_3 \,||\, m_4 \otimes e_4) \otimes e_1 \,||$
  $(m_2 \,||\, m_5 \otimes e_5 \,||\, m_6 \otimes e_6) \otimes e_2) \otimes e_1 k_n$

Task 1, part b.: routing the message

Once we send the message, routing starts at channel c_0. Channel c_0 receives two two packets with messages, ours and a one from other voter. After receiving them, c_0

- verifies and drops the signatures,
- removes the outer layer of encryption on each packet,
- publish the two commitments sign only by c_0,
- and sends the rest of the packets (signed also by c_0) to the next channels.

Every other channel does the same until messages are fully delivered.

Tasks 2 and 3

We can now move to the next tasks: revealing the random shuffle value and sending keys come next. The two can be done at once.

We generate a different key for each possible path, append the shuffle value, and prepare the message as before.

The first channel c_0 receives two messages, one is ours. The channels:

- checks the random values against the commitments and computes the shuffling

order,
- publish the shuffle values (each encrypted for the owner of the other),
- shuffles the pair of keys and makes it public,
- discards the unused branches,
- and forwards the used branches (in the shuffled order) to the next channel.

Every next channel does the same. At the end, each channel publishes the encrypted random values, and makes public the pair of shuffled keys.

Task 4

All left is to sign the pairs of shuffled keys. First we verify our keys are there, and we verify the shuffling order. Next, we route the signatures. We already know the full path, so this last message is send for one path only.

If still are steps remaining, we repeat the tasks. We can merge the following task 1 with this one.

Conclusion

If a voting system relies on a central authority, democracy can elect an attacker who stops democracy. A malicious authority can alter results, and even prevent people from participating in future elections.

I have introduced a voting algorithm that is distributed, and does not rely on a central authority to perform the election itself nor count the votes. An authority is still required at the beginning, to inscribe voters and their public keys. But once that process has started, people can emit votes in any election in a purely peer-to-peer basis, thus eliminating many ways in which an authority could impede future elections.

The algorithm relies on minimal assumptions: its security depends only on the existence of secure digital signatures, a secure encryption function, a pseudorandom function, and any public network for communication. As long of persons hold their private keys, and can communicate over any network, it is possible for them to cast their votes at any time without permission.

This small set of dependencies provides resilience on the long term. In the event that a particular signature or encryption function gets broken, it is only a matter of replacing that broken function with a new, secure one.

The algorithm is robust, in the sense that the presence of attackers only results in votes being less than perfectly shuffled, while that imperfect shuffle can be improved simply by running more round of the algorithm.

Source code

Python code shloud be available at:
https://jotasapiens.com/research
https://jota.tuxfamily.org

shuffle_v1.zip

SHA256: 0f38ca952150fe0c44422eb6e4aaef8c8f3e0be291edf686019566b68b87a3c7

References

[1] J. Cooley, J. Tukey, "An algortihm for the machine calculations of complex Fourier series," 1965.
[2] D. Goldschlag, M. Reed, P. Syverson, "Onion routing for anonymous and private internet connections," 1999.
[3] P. Syverson, "Sleeping dogs lie on a bed of onions but wake when mixed," 2011.
[4] D. Chaun, "Untraceable electronic mail, return addresses, and digital pseudonyms," 1981.
[5] D. Bayer, P. Diaconis, "Trailing the dovetail shuffle to its fair," 1992.
[6] B. Mann, "How many times should you shuffle a deck of cards?," 1994.