# ElGamal Encrypted Ballot - The Implementation

Yaheli, Guy, Eden, Simon, Shira

## Table of Contents



## About

This document outlines the structure and logic behind the ElGamal Voting Ballot implementation. It's designed to explain the cryptographic principles at play and offer clear insights into how the system was built from the ground up.

# Connection

The project relies on **socket connections** to enable communication between different components of the voting system. At the center of this setup is the **Bulletin Board**, which acts as a shared communication hub.

- The **Bulletin Board** starts by opening a socket server, allowing other participants to connect.
- Then, each role in the system connects to it in a specific order:
    1. **Admin** – responsible for setting up the election parameters and public keys.
    2. **Voter** – submits encrypted ballots through the socket.
    3. **Mixers** – shuffle and re-encrypt the ballots to preserve anonymity.
    4. **Verifiers** – check the integrity of the process and validate the final outcome.

This connection sequence ensures that each phase of the election is handled in the correct order, and that all data flows securely through a single, consistent channel.

# Encryption

The encryption uses **ElGamal encryption.**

ElGamal encryption is a public-key encryption based on the mathematics of cyclic groups and discrete logarithms.

## Group Class

This class sets up the **mathematical environment** for ElGamal encryption.

It defines a **cyclic group** with a large prime modulus and a generator.

- **Prime modulus** ($\boldsymbol{p}$): A large prime number that defines the size of the group.
- **Generator** ($\boldsymbol{g}$): An element of the group that can generate all other elements through exponentiation.
- **Group operations**: Includes modular multiplication, exponentiation, and inverse—all essential for encryption and decryption.

This class ensures that all cryptographic operations happen within a secure and consistent group structure.

## Encrypting

Encryption in ElGamal uses the **public key** and a randomly chosen number to produce a ciphertext.

- **Public key ($\boldsymbol{pk = g^x \bmod p}$)**: Derived from a secret key $x$, known only to the decryptor (The Admin).
- **Random ($\boldsymbol{r}$)**: A fresh random number used for each encryption to ensure uniqueness.
- **Ciphertext**: A pair of values $(C_1,\ C_2) = (g^r \bmod p,\ m \cdot pk^r \bmod p)$, where $m$ is the plaintext message

This ensures that even if the same message is encrypted twice, the ciphertext will look completely different due to the random $r$.

## Decrypting

Decryption uses the **private key** to recover the original message from the ciphertext.

- Given $(c_1,\ c_2)$ and private key $x$, compute:
- $s = {c_1}^x \bmod p$ (shared secret)
- $m = c_2 \cdot s^{-1} \bmod p$

This reverses the encryption process and retrieves the original message $m$.

# Mixing

The **Mixer** plays a crucial role in preserving voter anonymity. After votes are encrypted, the Mixer takes them and performs two key operations:

1. **Re-encryption** – to freshen the ciphertext without changing the vote.
2. **Shuffling** – to randomize the order of the votes.
3. **Proof Generation** – to prove that the Mixer only re-encrypted and shuffled, without altering the vote content.

This ensures that the final tally is correct, while making it impossible to trace a vote back to a specific voter.

## Re-encryption

When a message m is encrypted using ElGamal, the output is a pair of values, $(C_1, C_2) = (g^r \ mod \ p, m \cdot pk^r \ mod \ p)$.

To re-encrypt without decrypting, we introduce a new random nonce r'. The Mixer computes:

- $C_1{}' = C_1 * \ g^{r'} = \ g^r \cdot \ g^{r'} = g^{(r+r')} \ mod \ p$
- $C_2{}' = C_2 \cdot \ pk^{r'} = \ m \cdot pk^r \cdot \ pk^{r'} = m \cdot pk^{(r+r')} \ mod \ p$

So, the new ciphertext becomes,
$(C_1{}', C_2{}') = (g^{(r+r')} \ mod \ p, \ m \cdot pk^{(r+r')} \ mod \ p)$.

This is mathematically equivalent to encrypting m with a fresh nonce $r + r'$, but without ever touching the plaintext.

## Creating the Proof

After the Mixer re-encrypts and optionally shuffles the encrypted votes, to ensure trust, the Mixer must prove that:

- The votes were only re-encrypted and shuffled.
- No vote was changed or tampered with.

This is done using a **Zero-Knowledge Proof**, specifically a **Fiat-Shamir heuristic** (FDH) based proof.

Proof Construction:

- **Compute:** compute the difference between the original and re-encrypted ciphertexts: $(b_1, \ b_2) = (g^{r'}, \ pk^{r'})$.
- **Random:** Choose random value s, and compute $(A_1, A_2) = (g^s, \ pk^s)$.
- **Challenge:** Use a hash function to generate a **challenge** $c$ from public data.
- **Response:** Compute the response $r = s + c \cdot v \ mod \ Group.order$.

The proof will be $(A_1, \ A_2, \ c, \ r)$.

# Verifying

Once the Mixer has re-encrypted and possibly shuffled the votes, we need to **verify** that the transformation was honest—without knowing the actual vote content.

## Checking Equality Between Two Encrypted Votes

This confirms whether the encrypted votes and the re-encrypted votes contain the same votes without knowing the content of the votes.

Given:

- Original ciphertext: $(C_1,\ C_2) = (g^r\ , m \cdot pk^r\ )$.
- Re-encrypted ciphertext: $(C_1{}',\ C_2{}') = (g^{(r+r')}\ ,\ m \cdot pk^{(r+r')}\ )$.

We compute, $(b_1,\ b_2) = \left(C_1{}' \cdot\ C_1{}^{-1},\ C_2{}' \cdot\ C_2{}^{-1}\right) = (g^{r'},\ pk^{r'})$

Given the proofs, **Commitments**: $(A_1,\ A_2) = (g^s,\ pk^s)$, **Challenge**: $c$, **Response**: $r = s + c \cdot v\ mod\ Group.order$. The verifier checks whether,

$$(g^r,\ pk^r) = (A_1 \cdot\ b_1{}^c,\ A_2 \cdot\ b_2{}^c)$$

If the equality hold, the re-encryption is valid, and the vote content is unchanged.

If fails, the Mixer may have tampered with the vote or submitted a false proof.

## Verifying Proof

To confirm that the Mixer correctly re-encrypted and possibly swapped two encrypted votes, we perform **three critical equality checks**. Each check uses the zero-knowledge proof logic:

- $(C_1 \approx C_1{}')\ or\ (C_1 \approx C_2{}')$
- $(C_2 \approx C_1{}')\ or\ (C_2 \approx C_2{}')$
- $(C_1 \approx C_1{}')\ or\ (C_2 \approx C_2{}')$

All three conditions must hold to accept the proof.

## Handling a Cheater

If a **verifier detects cheating**—meaning the Mixer failed to correctly re-encrypt and shuffle the votes—the verifier immediately **notifies the Bulletin Board**. At this point, the Bulletin Board has two options:

*Overrule that Mixer*

- The Bulletin Board **rejects the mix** and removes all encrypted votes that were saved **after** the faulty Mixer.
- This effectively **rolls back** the system to the state before the mix.
- The original encrypted votes are **restored**, and the Mixer's attempt to cheat is neutralized.
- Mixing can then resume with a new or trusted Mixer.

*Not Overrule the Mixer*

- The Bulletin Board may choose to **ignore the verifier's claim**—either because it doesn't trust the verifier or wants to **re-run the verification**
- In this case, the mix is **not immediately rejected.**
- The system continues, and the mixing process **resumes** while further checks or decisions are made.

This flexible design ensures both **security** and **resilience**:

- Cheating is detectable and reversible.
- The system can handle disputes or false accusations without halting the election.