LELEC2770 – Practical Session 3: Verifiable Voting

In this session, we use a voting system based on ElGamal homomorphic encryption in a public group $\mathbb{G}$ (with generator g and size q), as discussed in the lecture.

Notations: private key x, public key h, ciphertext (c_1, c_2), integer α.

We recall some high-level operations that you can do with the ElGamal cryptosystem:

- Standard public key operations: $(x, h) \leftarrow \mathsf{GenKey}(\mathbb{G})$, $(c_1, c_2) \leftarrow \mathsf{Enc}_h(m, r)$ (where r should be a fresh random element), $m \leftarrow \mathsf{Dec}_x((c_1, c_2))$,
- Homomorphic operations: $(c_1, c_2) \leftarrow \mathsf{Add}((c_{1,1}, c_{2,1}), (c_{1,2}, c_{2,2}))$, $(c'_1, c'_2) \leftarrow \mathsf{Mul}(\alpha, (c_1, c_2))$,
- Zero-knowledge proof of decryption for Elgamal ciphertext: proving that $P \equiv (c_1, c_2) = Enc_h(m)$, knowing the randomness r $\mathsf{ElGPDec}(r, h, (c_1, c_2), m)$. (This is a sigma protocol.)
- Fiat-Shamir heuristic: a method to transform a sigma-protocol (which is an interactive zero-knowledge proof protocol) into a non-interactive zero-knowledge proof.
- Zero-knowledge proof that an ElGamal ciphertext encrypts either a 0 or a 1. (This is considered as a black-box.)

Exercise 1. How do trustees prove knowledge of their secret key? Detail the proof generation and proof verification operations. Show what problem may happen if the system does not require the trustees to make this proof.

Exercise 2. For each of the following election settings, design the ballot and a sigma protocol that makes the ballot verifiable. Describe how to generate the ballot and how to decrypt (while verifying) it. Write your answer using the high-level operation listed above.

1. A single yes/no question.
2. Five yes/no questions.
3. Five yes/no questions, and there must be exactly 3 "yes" answers in each ballot.

Exercise 3. For each of the high-level operations you used in the previous question, explain how to implement them (e.g. using pseudo-code) in terms of group operations. (You may skip the operations that you already implemented during previous exercise sessions, and the "0 or 1" ciphertext proof ;)).

Exercise 4. We now move to the implementation of an election where the ballot is a single yes/no.

The bulletin board is encoded in JSON, with the following structure.[1]

```
<group> ::= {"p": <p>, "g": <g>}
<pk> ::= <y>
<ballot> ::= {
    "ct": {
        "c1": <c1>,
        "c2": <c2>,
    },
    "dproof": <dproof>,
}
<bulletin_board> ::= {
    "group": <group>,
    "pk": <pk>,
    "ballots": [<ballot_0>, <ballot_1>, ... ],
}
```

All elements which are not defined (such as `<g>`, `<c1>`,...) are integers, except for `<dproof>` that you can treat as a black-box[2].

The syntax for the group and the ElGamal public key (`pk`) should be self-explaining. The pair `(<c1>, <c2>)` is an ElGamal ciphertext that encrypts the vote.

Your goal is:

1. To implement the tally process to be able to compute the result of the election `bb.json`. You have access to a decryption oracle (at `https://lelec2770.pythonanywhere.com/elections1`) but you cannot query it on the ciphertexts of `bb.json`. In order to test you implementation, the bulletin board `bb_test.json` contains 3 valid votes and one invalid vote (the last one) with the encryption secret key $x = 567904$.

2. To unveil the choice of the first voter of `bb.json`.

3. To implement the ballot generation process. (Test your implementation against you tallying code.)

Exercise 5.

1. To securely manage the election, we need multiple trustees.

    - Implement the combination of multiple trustee public keys in a single election key.

[1] To encode and decode to/from this format, use the `json` module.

[2] Use the function `vote_dproof.generate_ballot` to generate a valid ballot and the function `vote_dproof.verify_ballot` to verify that the disjunctive proof is valid.

- Implement the decryption factor generation by the trustees and their usage to decrypt the tally.
- Integrate your code with the one of exercise 4 to simulate a full election.

2. For the trustees operations to be publicly verifiable, they need to prove that they know the secret key corresponding to their public key and that their decryption factor is correct. We first provide you a bulletin board (`bb_proof.json`, with the trustees public keys, the ballots, the decryption factors, and the relevant zero-knowledge proofs. Adapt your code to reveal the election result while checking that all the proofs are valid.

   See the description of the bulletin board and the proofs below.

3. Implement the generation of the proofs, and simulate a full election.

The new bulletin board is encoded in JSON, with the following structure:

```
<tallier_key> ::= {
    "pk": <y>,
    "commit": <d>,
    "response": <f>,
}
<dec_factor> ::= {
    "pk": <y>,
    "c1": <c1>,
    "decryption_factor": <s>,
    "commit": [<d0>, <d1>],
    "response": <f>,
}
<full_bulletin_board> ::= {
    "group": <group>,
    "tallier_keys": [ <tallier_key_0>, <tallier_key_1>, ... ],
    "decryption_factors": [ <decryption_factor_0>, <decryption_factor_1>, ... ],
    "ballots": [<ballot_0>, <ballot_1>, ... ],
}
```

where:

- `"group"` and `"ballots"` are the same as in the previous bulletin board.
- `"tallier_keys"` contains one element per tallier, the public key is `<y>` and the proof is a non-interactive Schnorr proof such that $\texttt{<g>}^{\texttt{<f>}} = \texttt{<d>} \cdot \texttt{<y>}^e$, where $e = \mathcal{H}_q(\{\texttt{"pk": <y>, "commit": <d>}\})$.
- `"decryption_factors"` contains one element per tallier (being the decryption factor for the corresponding tallier key. For and ElGamal ciphertext (c_1, c_2) encrypted under the secret key x, the decryption factor `<s>` is c_1^x. The proof is a Chaum-Pedersen ZKP such that $\texttt{<g>}^{\texttt{<f>}} = \texttt{<d0>}\cdot\texttt{<y>}^e$ and $\texttt{<c1>}^{\texttt{<f>}} = \texttt{<d1>}\cdot\texttt{<s>}^e$, where

  $e = \mathcal{H}_q(\{\texttt{"pk":<y>,"c1":<c1>,"decryption\_factor":<s>,"commit":[<d0>,<d1>]}\})$.

The $\mathcal{H}_q$ hashes a JSON object into an element of $\{0, \ldots, q-1\}$. For q, use the size of the group (`group.q` if `group` is an `ElgamalGroup` object). This function is implemented in the file `vote_dproof.py`.