Implementing an E-Voting Scheme Using Cryptography

CS 517 - Advanced Cryptography & Data Security
Term Project for Spring 2019

E. Savaş
Computer Science & Engineering
Sabancı University
İstanbul

Abstract

You are required to develop an electronic voting scheme using cryptographic primitives where there are l voters $(V_1, V_2, \ldots, V_l)$, k candidates $(G_1, G_2, \ldots, G_k)$, one *honest verfier (HV)*, and n tallying authorities $(A_1, A_2, \ldots, A_n)$. The honest verifier (HV) is an authorized party who is in charge during the actual voting process. It does not see the contents of the votes, but verifies that what is cast is an actual and legitimate vote. The tallying authorities are the parties in charge of counting votes after the election is completed. Since we do not trust a single entity with counting the votes, we employ n tallying authorities who need to collaborate for the task of vote counting. This way we can distribute the trust and no single entity can cheat since a certain number of tallying authorities must participate in the process.

1 Introduction

The electronic voting scheme will be implemented in four phases as briefly explained in the following:

1. **Setup Phase:** Domain parameters and system public and private keys are generated.
2. **Vote Generation:** Voter generates his own vote.
3. **Vote Casting:** Voter casts his vote with the help of the honest verifier (HV).
4. **Vote Counting (Tallying):** At least t out of n tallying authorities come together to count the votes.

The project will be completed in four phases corresponding to the described phases of the election. The following sections provide the detailed explanation of the project phases.

2 Setup Phase

We will use an ElGamal-like encryption scheme that allows homomorphic computation over the encrypted votes. You are required to develop codes to generate the following parameters for the encryption scheme:

- Two primes: p and q such that $q|p-1$. p and q must be 3072-bit and 256-bit, respectively. Use Algorithm 1.

- g: a generator in $\mathcal{G}_q$ (a subgroup of Z_p^*). In other words, the powers of g will generate q distinct elements of Z_p^*, which will form a group with q elements under modular multiplication where modulus is p. Use Algorithm 2.
- s: system private key (a random integer with $1 \leq s \leq q-1$).
- h: system public key $h = g^s \bmod p$.
- Finally, generate k independent and randomly chosen generators $(G_1, G_2, \ldots, G_k)$ for k candidates using Algorithm 3.

Algorithm 1 Generating two large primes p and q

Require:
Ensure: p, q

```
1: Generate a random prime number $q$ of at least 256 bit.
2: Generate a random number $k$ of about $3072 - 256 = 2816$ bit.
3: $p \leftarrow kq + 1$
4: if $p$ is not prime then
5:     go to Step2.
6: end if
7: return $\{p, q\}$
```

Algorithm 2 Finding a generator element

Require: p, q
Ensure: g

```
1: Select a random element $\alpha \in Z_p^*$
2: Compute $g \leftarrow \alpha^{(p-1)/q} \bmod p$
3: if $g = 1$ then
4:     go to Step 3.
5: end if
6: return $\{p, q, g\}$
```

Algorithm 3 Verifiably random generation of k generators of G_q

Require: p, q
Ensure: $(G_0, \rho_0), \ldots (G_{k-1}, \rho_{k-1})$

```
1: for $i = 0$ to $k - 1$ do
2:     $\rho_i \leftarrow Z$.
3:     $\phi_i \leftarrow \texttt{SHA256}(\rho_i) \pmod q$.
4:     Compute $G_i \leftarrow \phi_i^{(p-1)/q} \bmod p$
5:     if $G_i = 1$ then go to Step 2.
6:     end if
7:     return $\{(G_1, \rho_1), \ldots (G_{k-1}, \rho_{k-1})\}$
8: end for
```

Your implementation will be tested using the python code “setup_test.py”. You are required to use the same function names and syntax and your functions must be callable from code “setup_test.py”.

3 Vote Generation Phase

In the vote generation phase, there are three types of participants in the system, namely l voters, one tallying authority, and one system manager. Voters cast their votes, and vote casting results in encrypted votes using the public key of the system, h. The tallying authority counts the votes using the encrypted individual votes. The system manager is responsible for system setup and knows the private key of the system, s.

You will implement the following steps:

1. **Vote Casting (executed by voters):** The voter V_i casts his vote for the candidate G_j by calculating $(x_i, y_i) = (g^{r_i}, h^{r_i} G_j)$, where r_i is uniformly randomly chosen integer in Z_q^*.
2. **Tallying (executed by a single tallying authority):** Combine the votes of l users, namely (x_i, y_i) for $i = 0, 1, \ldots, l-1$, using the formula $(X, Y) = (\prod_{i=0}^{l-1} x_i, \prod_{i=0}^{l-1} y_i)$.
3. **Decryption of the Election Result - Step 1 (executed by the system manager):** Decrypt (X, Y) using the formula $YX^{-s} = \prod_{j=0}^{k-1} G_j^{b_j}$, where b_j is the vote count of the candidate G_j and $\sum_{j=0}^{k-1} b_j = l$.
4. **Calculation of the Election Result - Step 2 (executed by the system manager):** Find the number of votes that each candidate receives by exhaustive search.

After completing these steps, you need to demonstrate your implementation using $k = 4, l = 40$ as an example scenario in which votes are randomly selected. Your implementation will be tested using the python code “vote_gen_test.py”. You are required to use the same function names and syntax and your functions must be callable from code “vote_gen_test.py”.

4 Tallying Phase

Your implementation will be tested using the python code “tally_test.py”. You are required to use the same function names and syntax and your functions must be callable from code “tally_test.py”.

4.1 Counting the Votes

In this phase of the project, n tallying authorities count the votes. The previous steps are as you implemented them in the previous phase of the project; but this time the vote counting will be done collaboratively with n tallying authorities, each of which knows a share of the private key s. For secret sharing, you will implement (t, n)-threshold scheme using Shamir’s technique. The following are the steps:

1. Private key s is shared among n tallying authorities A_a $(0 < a < n-1)$ using Shamir’s threshold scheme so that any t tallying authorities can count the votes collaboratively. As a result, each authority has a share (a, s_a), where $s_a \in Z_q^*$.

2. The votes are multiplied as follows: $(X, Y) = (\prod_{i=0}^{l-1} x_i, \prod_{i=0}^{l-1} y_i)$.
3. At least t tallying authorities form a quorum Λ to start the tallying process.
4. Tallying authorities in Λ multicast $\Omega_a = X^{s_a}$ to other tallying authorities.
5. The tallying authorities compute $\prod_{j=0}^{k-1} {G_j}^{b_j} = \dfrac{Y}{\prod_{a \in \Lambda} \Omega_a^{\lambda_{a,\Lambda}}}$, where $\lambda_{a,\Lambda} = \prod_{m \in \Lambda, m \neq a} \dfrac{m}{m-a}$

   (Recall that $s = \sum_{a \in \Lambda} s_a \cdot \lambda_{a,\Lambda}$).
6. (Decryption of the election result) Find the number of votes each candidate receives by exhaustive search on $\prod_{j=0}^{k-1} {G_j}^{b_j}$.

For the demonstration, consider the following cases:

1. Use $k = 4, l = 100$ for this project phase and assign votes of each voter randomly to candidates.
2. Use $n = 5, t = 3$ for tallying operation.

4.2 Proof of Knowledge of Common Exponent

Each tallying authority in Λ publishes its share of the public key, namely $h_a = g^{s_a}$. Each tallying authority is required to prove that $\Omega_a = X^{s_a}$ and $h_a = g^{s_a}$ have the same exponent s_a without revealing it. Implement the protocol in the course slides (Slide 12 in 10_E-voting.ppt) and run it with a randomly chosen tallying authority for demonstration.

Here, you have to perform two checks:

1. Check if the quorum in Section 4.1 is a good quroum. For this, you need to combine the partial public keys of the tallying authorities, h_a in the quorum and show the result is the public key h. Use `CheckQuorum()` function in tally_test.py.
2. Run the zero-knowledge proof-of-knowledge of common exponent protocol to show that a randomly chosen authority a can prove that he knows the secret share and Ω_a is indeed equal to X^{s_a}. Use `ZK_commonexp()` function in tally_test.py.

5 Vote Casting Phase

This stage involves the voter and the honest verifier (HV). The requirements of this phase are given in three stages in the following.

5.1 Stage I

The voter has to prove to HV that his/her vote is valid without revealing the vote (i.e., zero-knowledge proof). In other words, in his/her vote $(x, y) = (g^{r_i}, h^{r_i} G_j)$, the second term contains one of the generators (G_1 or G_2 or … or G_k), each of which corresponds to one candidate in the system. In Appendix I, a zero-knowledge proof system is given for two candidates.

Firstly, you are required to extend the protocol in Appendix I to k candidates and describe the protocol steps. Then, you are required to demonstrate your implementation of the protocol. In your demo, you need to show that the protocol is working for at least a three-candidate election, namely $k = 3$. During the demo, you need to show the following cases:

- The user votes for G_1, the honest verifier accepts the vote.
- The user votes for G_2, the honest verifier accepts the vote.
- The user votes for G_3, the honest verifier accepts the vote.
- The user prepares a fake vote using another generator G, the honest verifier detects it and rejects the vote.
- The user prepares a fake vote G_i^{100} for $i \in \{1, 2, 3\}$, the honest verifier detects it and rejects the vote.

5.2 Stage II

In this stage of the project, you are required to implement the protocol in Appendix III to generate the final vote of a user. This protocol is necessary for receipt-freeness. For more information about receipt-freeness see also Appendix II. You need to generalize the protocol in Appendix III to k users.

5.3 Stage III

Note that the final ballot of voter i is $(x_i, y_i) = (u_i x, v_i y)$, where $(u_i, v_i) = (g^{t_i}, h^{t_i})$ for $t_i \leftarrow Z_q^*$ chosen by HV (and unknown to the voter). To sell his vote, the voter can show, to any party who are interested in buying his vote (*buyer*), that his original ballot (x, y) can be reconstructed by revealing u_i and v_i; then he tries to convince the buyer that the original ballot is a vote for a particular candidate G_j. For this, he reveals to the buyer r_i, where $(x, y) = (g^{r_i}, h^{r_i} G_j)$.

But the buyer would not be convinced with this as the voter can transform the final ballot to a vote for any other candidate by $(\tilde{x}, \tilde{y}) = (g^{r_i}, h^{r_i} G_k)$ for $k \neq j$. Demonstrate this by providing $(\tilde{u}_i, \tilde{v}_i)$, where $(x_i, y_i) = (\tilde{u}_i \tilde{x}, \tilde{v}_i \tilde{y})$.

6 Appendix I: Timeline & Deliverables & Weight etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement	NA	03/05/2019	NA
Setup Phase	setup.py	10/05/2019	10%
Vote Generation Phase	vote_gen.py	10/05/2019	10%
Tallying Phase	tally.py	10/05/2019	40%
Vote Casting Phase	vote_cast_I.py vote_cast_II.py vote_cast_III.py	24/05/2019	40%

Notes:

1. You may be asked to demonstrate every phase to the instructor.
2. Students are required to work alone.

7 Appendix I: Zero-Knowledge OR-Proof for Two-Way Election

In this section, we show how the voter proves to the honest verifier that he voted for one of the two candidates without revealing which one. Assume that the candidates are represented by two generator elements, G_1 and G_2 and that the voter i has voted for G_j, where $j \in \{1, 2\}$. In other words, the vote is in the form $(x, y) = (g^{r_i}, h^{r_i} G_j)$ for a randomly chosen r_i. Note that $w \leftarrow Z_q$ means that w is randomly chosen in Z_q.

1. **Voter (Prover):** Performs the following for his vote $(x, y) = (g^{r_i}, h^{r_i} G_j)$:
   (a) Computes $(a_j, b_j) = (g^w, h^w)$ for $w \leftarrow Z_q^*$.
   (b) Computes $(a_{2-j+1}, b_{2-j+1}) = (g^{z_{2-j+1}} x^{c_{2-j+1}}, h^{z_{2-j+1}} (y/G_{2-j+1})^{c_{2-j+1}})$
   for $c_{2-j+1}, z_{2-j+1} \leftarrow Z_q^*$
   (Note that if $j = 1$ then $2 - j + 1 = 2$; otherwise if $j = 2$ then $2 - j + 1 = 1$).
   (c) Sends (x, y), (a_j, b_j) and (a_{2-j+1}, b_{2-j+1}) to the honest verifier.
2. **Honest Verifier:** Sends c to Voter, where $c \leftarrow Z_q^*$.
3. **Voter (Prover):** Computes $(c_j, z_j) = (c - c_{2-j+1}, w - r_i c_j)$ and sends (c_1, z_1) and (c_2, z_2) to the honest verifier.
4. **Honest Verifier:** Accepts the ballot (x, y) if the following equalites are satisfied (use the correct modulus):
   (a) $c = c_j + c_{2-j+1}$,
   (b) $a_1 = g^{z_1} x^{c_1}$,
   (c) $a_2 = g^{z_2} x^{c_2}$,
   (d) $b_1 = h^{z_1} (y/G_1)^{c_1}$,
   (e) $b_2 = h^{z_2} (y/G_2)^{c_2}$.

8 Appendix II: Zero-Knowledge Proof for Common Exponent

The following protocol is used to prove that the HV knows a common exponent t_i, satisfying $u_i = g^{t_i}$ and $v_i = h^{t_i}$ without revealing the exponent to the voter. In this protocol, roles are switched; namely, the voter is the verifier and the HV is the prover. This protocol is used to blind (re-randomize) the initial vote generated by the honest verifier for implementing *receipt-freeness* feature. In the end the vote will have the form

$$(x, y) = (g^{r_i+t_i}, h^{r_i+t_i}G_j).$$

Since the voter does not know t_i he will not be able to prove a potential buyer his vote.

The protocol can be described as follows:

1. **Honest Verifier:** Performs the following operations:
   (a) Computes $(u_i, v_i) = (g^{t_i}, h^{t_i})$ for $t_i \in_R Z_q^*$.
   (b) Computes $(d, e) = (g^{w_2}, h^{w_2})$ for $w_2 \in_R Z_q^*$.
   (c) Sends (u_i, v_i) and (d, e) to the voter.
2. **Voter (Prover):** Sends f to the honest verifier, where $f \in_R Z_q^*$.
3. **Honest Verifier:** Computes and sends σ to the voter, where $\sigma = w_2 + ft_i$.
4. **Voter (Prover):** Accepts (u_i, v_i) if the following equalites are satisfied:
   (a) $g^\sigma = du_i^f$,
   (b) $h^\sigma = ev_i^f$.

9 Appendix III: Final Ballot Generation Protocol for Two-Way Election

This protocol is in fact the combination of the protocols in Appendix I and Appendix II.

1. **Voter (Prover):** Performs the following for his initial vote $(x, y) = (g^{r_i}, h^{r_i} G_j)$, where $r_i \leftarrow Z_q^*$:
   (a) Computes $(a_j, b_j) = (g^{w_1}, h^{w_1})$ for $w_1 \leftarrow Z_q^*$.
   (b) Computes $(a_{2-j+1}, b_{2-j+1}) = (g^{z_{2-j+1}} x^{c_{2-j+1}}, h^{z_{2-j+1}} (y/G_{2-j+1})^{c_{2-j+1}})$ for $c_{2-j+1}, z_{2-j+1} \leftarrow Z_q^*$.
   (c) Sends (x, y), (a_j, b_j) and (a_{2-j+1}, b_{2-j+1}) to the honest verifier.

2. **Honest Verifier:** Perfoms the following computations:
   (a) Computes $(u_i, v_i) = (g^{t_i}, h^{t_i})$, for $t_i \leftarrow Z_q^*$.
   (b) Computes $(d, e) = (g^{w_2}, h^{w_2})$, for $w_2 \leftarrow Z_q^*$.
   (c) Sends (u_i, v_i), (d, e), and c to the voter, where $c \leftarrow Z_q^*$.

3. **Voter (Prover):** Perfoms the following computations:
   (a) Computes $(c_j, z_j) = (c - c_{2-j+1}, w_1 - r_i c_j)$
   (b) Sends (c_1, z_1), (c_2, z_2), and f to the honest verifier, where $f \leftarrow Z_q^*$.

4. **Honest Verifier:** Accepts the ballot (x, y) if the following equalites are satisfied (use the correct modulus) and sends $\sigma = w_2 + f t_i$ to the voter:
   (a) $c = c_j + c_{2-j+1}$,
   (b) $a_1 = g^{z_1} x^{c_1}$,
   (c) $a_2 = g^{z_2} x^{c_2}$,
   (d) $b_1 = h^{z_1} (y/G_1)^{c_1}$,
   (e) $b_2 = h^{z_2} (y/G_2)^{c_2}$.

5. **Voter (Prover):** Generates the final ballot $(x_i, y_i) = (u_i x, v_i y)$ if the following equalites are satisfied:
   (a) $g^{\sigma} = d u_i^f$,
   (b) $h^{\sigma} = e v_i^f$.