CSCI 1515: Applied Cryptography

P. Miao

Spring 2024

These are lecture notes for CSCI 1515: Applied Cryptography taught at Brown University by Peihan Miao in the Spring of 2024.

These notes were originally taken by Jiahua Chen with gracious help and input from classmates and fellow TAs. Please direct any mistakes/errata to a thread on Ed, or feel free to pull request or submit an issue to the notes repository.

Notes last updated March 5, 2024.

Contents

§1 March 4, 2024

§1.1 Anonymous Online Voting

Recall anonymous online voting.

Say we have n voters with votes $v_1, \ldots, v_n \in \{0, 1\}$. Each voter encrypts their vote $\text{Enc}(v_1), \ldots, \text{Enc}(v_n)$. Our goal is to compute the sum of these votes without having to decrypt each vote individually. Somehow, we must find $\text{Enc}(\sum v_i)$ then decrypt to find $\sum v_i$.

There are three questions.

1. How do we compute $\text{Enc}(\sum v_i)$?
2. How do we ensure each vote is 0 or 1?
3. Who decrypts $\text{Enc}(\sum v_i)$?

§1.2 Additively Homomorphic Encryption

1. Additively homomorphic encryption is taking two encryptions $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$ and combine them to get $\text{Enc}(m_1 + m_2)$.
2. Multiplicatively homomorphic encryption is taking two encryptions $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$ and getting the product $\text{Enc}(m_1 \cdot m_2)$.

Example 1.1 (Elgalmal Encryption)

Elgamal Encryption: Cyclic group G with generator g, and the public key is given by $\text{pk} = g^{\text{sk}}$. The encryption of message m_1 is given by $\text{Enc}_{\text{pk}}(m_1) = (g^{r_1}, \text{pk}^{r_2} \cdot m_1)$ and the encryption of message m_2 is given by $\text{Enc}_{\text{pk}}(m_2) = (g^{r_2}, \text{pk}^{r_2} \cdot m_2)$. If we multiply the first components together and then the second components together, we get $(g^{r_1+r_2}, \text{pk}^{r_1+r_2} \cdot (m_1 \cdot m_2))$ which is exactly $\text{Enc}(m_1 \cdot m_2)$. Thus, we have multiplicatively homomorphic encryption.

Exponential Elgamal: The encryption of message m_1 is given by $\text{Enc}_{\text{pk}}(m_1) = (g^{r_1}, \text{pk}^{r_1} \cdot g^{m_1})$ and the encryption of message m_2 is given by $\text{Enc}_{\text{pk}}(m_2) = (g^{r_2}, \text{pk}^{r_2} \cdot g^{m_2})$. If we multiply them together element-wise like before, we get $(g^{r_1+r_2}, \text{pk}^{r_1+r_2} \cdot g^{m_1+m_2})$ which is exactly $\text{Enc}(m_1 + m_2)$. thus, we have additively homomorphic encryption.

How do we do decryption? Normally, we take $c_1 = g^{r_1+r_2}$ and $c_2 = \text{pk}^{r_1+r_2} \cdot g^{m_1+m_2}$ and compute c_2/c_1^{sk}. Usually this equals the plaintext, but in this scenario it equals $g^{m_1+m_2}$.

In our anonymous online voting scenario, each vote will be 0 or 1. Thus, the summation of all the votes is at most n, where n is the number of voters. This is a polynomial number of

possibilities, so we can compute g^m for each $m \in \{0, 1, \ldots, n\}$ and see which one matches $g^{\sum m_i}$ to recover the summation of the votes.

§1.3 Threshold Encryption

Definition 1.2 (t-out-of-n threshold)

In t-out-of-n threshold encryption, we must have t parties out of n parties come together in order to decrypt.

Let t parties be denoted by $p_1, \ldots, p_t$. Each party p_i works independently and runs a partial gem algorithm $\text{PartialGem}(1^\lambda)$, which generates a public and secret key pair $(\text{pk}_i, \text{sk}_i)$. After everyone is done, we combine all of the public keys pk_i to get one collective public key pk. A message is encrypted using it $\text{ct} \leftarrow \text{Enc}_{\text{pk}}(m)$. Note that a single party cannot decrypt by themselves.

In order to decrypt, each party p_i runs a partial decryption algorithm $\text{PartialDec}(\text{sk}_i, \text{ct})$ that gives a partial decryption d_i. Then, we combine all of the partial decryptions d_i to get the plaintext m.

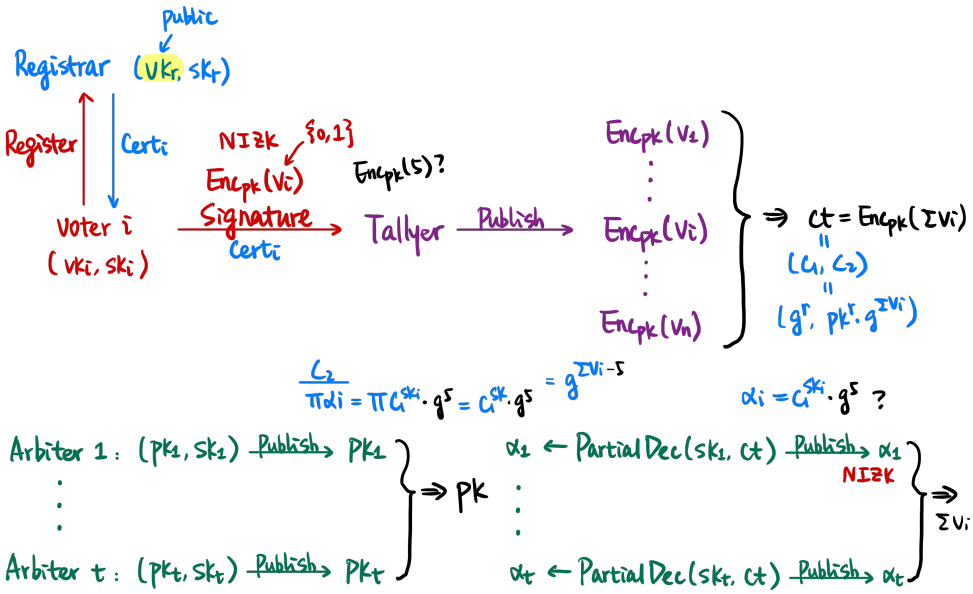
§1.3.1 Threshold Encryption: Elgamal

Now we give an explicit construction of threshold encryption using Elgamal.

Each party p_i generates a random secret key $\text{sk}_i \leftarrow \mathbb{Z}_q$ and public key $\text{pk}_i = g^{\text{sk}_i}$. Let pk be the product of all pk_i, which gives us $g^{\sum \text{sk}_i}$. Next we encrypt $\text{ct} = (c_1, c_2) = (g^r, \text{pk}^r \cdot g^m)$. In order to decrypt this, we need to compute c_2 / c_1^{sk} where $\text{sk} = \sum \text{sk}_i$. However, any single party does not know sk by themselves.

To decrypt, each party p_i does a partial decryption by computing $d_i = c_1^{\text{sk}_i}$. When all parties come together, they can multiply all the partial decryptions d_i which gives us $c_1^{\sum \text{sk}_i} = c_1^{\text{sk}}$. We can use this to compute c_2 / c_1^{sk} and decrypt.

§1.4 Voting Framework



We have some servers:

Registrar. For a voter to be able to vote, they register with the Registrar to obtain a certificate to vote. They get a certificate for their verification key.

Arbiters. The arbiters will generate the threshold encryption keys. There will be t arbiters and each will have their (pk_i, sk_i). They all reveal pk_i to the public, so that everyone can compute the full public key pk.

Voter. The voter, using the public key, will encrypt $v_i \in \{0, 1\}$. The voter will sign this vote using their signing key. They will send this vote to the Tallyer.

Tallyer. The tallyer will check that the signature is valid[1]. Then, they will strip the signature and output $\mathsf{Enc}_{pk}(v_1), \ldots, \mathsf{Enc}_{pk}(v_i), \ldots, \mathsf{Enc}_{pk}(v_n)$.

§1.4.1 Correctness of Partial Decryption

Given a cyclic group G of order q with generator g, we have three pieces of public information.

1. The partial public keys of each party $\mathrm{pk}_i \in G$.

2. The ciphertext $c = (c_1, c_2)$.

[1]This way, the registrar cannot figure out who has voted and who hasn't.

3. The partial decryption of each party d_i.

The witness is the partial secret key sk_i which is private to each party. The language for ZKP for partial decryption is

$$R_L = \{((c_1, pk_i, d_i), \underbrace{sk_i}_{\text{witness}}) : pk_i = g^{sk_i} \wedge d_i = c_1^{sk_i}\}$$

This is still the Diffie-Hellman tuple! $pk_i = g^{sk_i}, c_1 = g^r, d_i = g^{r \cdot sk_i}$. We can use the NIZK for Diffie-Hellman tuple as discussed in previous lectures.

§1.4.2 Correctness of Encryption

We want voters to prove that their encryption is either of 0 or 1. We're in group $\mathbb{G}$ with order q and generator g. We have public key $pk \in \mathbb{G}$, and ciphertext $c = (c_1, c_2)$. We're trying to prove the statement "c is an encryption of 0 OR c is an encryption of 1."

Our languages are then encryptions of 0 and encryptions of 1:

$$R_{L_0} = \{(\underbrace{(pk, c_1, c_2)}_{x}, \underbrace{r}_{\text{witness}}) : c_1 = g^r \wedge c_2 = pk^r\}$$
$$R_{L_1} = \{(\underbrace{(pk, c_1, c_2)}_{x}, \underbrace{r}_{\text{witness}}) : c_1 = g^r \wedge c_2 = pk^r \cdot g\}$$

where r is our private key. Using this, we can prove that c is an encryption of 0 ($c_2 = pk^r$) or c is an encryption of 1 ($c_2 = pk^r \cdot g$).

§1.5 Proving AND/OR Statements

For AND, our statements are x_1, x_2 and our witnesses are w_1, w_2. The language is given by

$$R_{AND} = \{((x_1, x_2), (w_1, w_2)) : (x_1, w_1) \in R_{L_1} \text{ AND } (x_2, w_2) \in R_{L_2}\}$$

To prove this language, we can use a ZKP for R_{L_1} and a ZKP for R_{L_2}.

For OR, our statements are x_1, x_2 and our witness is w. The language is

$$R_{OR} = \{((x_1, x_2), w) : (x_1, w) \in R_{L_1} \text{ OR } (x_2, w) \in R_{L_2}\}$$

To prove this language, we cannot use a ZKP for R_{L_1} and a ZKP for R_{L_2}. If we do, then we reveal whether $(x_1, w) \in R_{L_1}$ and whether $(x_2, w) \in R_{L_2}$, which is revealing more information than allowed.

To prove R_{OR}, we know that both languages R_{L_1} and R_{L_2} works with a sigma protocol. The prover is going to send (A_1, B_1) for the first language and (A_2, B_2) for the second language, pretending that both are correct. The verifier sends a challenge $\sigma \leftarrow \mathbb{Z}_q$. The prover separates σ into σ_1 and σ_2, and computes responds S_1, S_2 for σ_1, σ_2 respectively. Then the verifier will verify that $\sigma = \sigma_1 + \sigma_2$, as well as the responses $((A_1, B_1), \sigma_1, S_1)$ and $((A_2, B_2), \sigma_2, S_2)$.

How does the Prover compute a response for both statements? Since we are working with OR, we might not have inclusion in one of the languages. For that language, we will simulate it.