CKKS Lecture Notes

Part 4: Multiparty

FEI

October 2026

1 Introduction

Hypothetical Situation

The governments of Aliceland and Bobania each operate some satellites. They want to ensure that their satellites do not crash without revealing the locations of their satellites.

Let's look at some simple-sounding solutions and see why they fail.

Attempt 1

The government of Aliceland encrypts the locations of their satellites under CKKS and sends this to the government of Bobania. The government of Bobania constructs a circuit that returns 1 if a satellite is in danger of crashing into a Bobanian satellite, and 0 otherwise. They run this circuit on the government of Aliceland's ciphertext, and send this back to them.

Fully homomorphic encryption guarantees that this doesn't reveal anything to Bobania about Aliceland's data. However, there is no guarantee that this doesn't reveal anything to Aliceland about Bobania's circuit.

Attempt 2

The governments of Aliceland and Bobania each send the locations of their satellites to the government of Carolistan. The government of Carolistan determines which satellites are in danger of crashing and sends the relevant information to each government.

This reveals the locations of the satellites to the government of Carolistan. The governments of Aliceland and Bobania might not want to reveal this information

to anybody.

Attempt 3

The governments of Aliceland and Bobania agree on a CKKS secret key. They each encrypt the locations of their satellites under that key and send the result to the government of Carolistan. The government of Carolistan runs a circuit that determines which satellites are in danger of crashing and sends the results back.

Carolistan has Aliceland's encrypted data, and Bobania has the secret key that it was encrypted under. If Bobania and Carolistan work together, they can determine the locations of the Alicelandian satellites.

2 Multiparty

The solution to the situation in the previous section is multiparty computation. More generally, it achieves the following goal:

Goal

Participants $P_1, \ldots, P_n$ have data $x_1, \ldots, x_n$ and an agreed-upon function f. Each participant learns $f(x_1, \ldots, x_n)$, but no participant learns anything else about another participant's data.

Multiparty CKKS works similarly to Attempt 3. Each participant encrypts their data under the same key and the server homomorphically computes over it. However, no participant learns the secret key. Instead, each participant has a "share" of the group secret, which they use to jointly generate a public keyset and jointly decrypt the result of the computation.

Procedure

1. Participants $P_1, \ldots, P_n$ generate secret key shares $s_1, \ldots, s_n$. The secret key is $s = s_1 + \cdots + s_n$.
2. Participants $P_1, \ldots, P_n$ jointly compute:
   - A public key corresponding to s
   - A relinearization key corresponding to s
   - Automorphism keys corresponding to s

   Notably, they learn this information without learning s.
3. Each participant encrypts their data under the public key and sends it to the server.
4. The server homomorphically computes some function of the data and reveals the encrypted result $\mathbf{ct}_{\text{final}}$.
5. Each participant uses their key share to generate a *decryption share* of $\mathbf{ct}_{\text{final}}$. Using these shares, they jointly compute the result.

This achieves the goal and has some additional advantages:

- Most of the work happens on the server.
- The participants only need to send and receive data a fixed number of times at the beginning and end of the computation.
- If the same group wants to do another computation in the future, they can reuse the keyset.

Because the server does most of the work, people don't need a strong computer to participate in multiparty CKKS computations.

3 Review: Key generation

In single-party CKKS, the secret key holder has a secret key $s \in \mathbb{Z}[X]/(X^N+1)$ with small coefficients.

They can reveal a *public key* $(a_{\mathsf{pk}}, b_{\mathsf{pk}})$ such that $a_{\mathsf{pk}}s + b_{\mathsf{pk}} \approx 0 \pmod{Q}$. This public key allows others to encrypt ciphertexts under s without knowing s.

In addition, for other secrets s', the secret key holder can reveal a *keyswitching key* $\mathsf{ksk}_{s'}$. This relies on a public *decomposition basis* $w_1, \ldots, w_k$ of large integers. The keyswitching key is a set of (a_i, b_i) such that

$$a_i s + b_i \approx w_i s' \pmod{PQ}.$$

There are two main types of keyswitching keys:

- **Automorphism keys.** For odd r, let σ_r be the automorphism of $\mathbb{Z}_{PQ}[X]/(X^N+1)$ that fixes $\mathbb{Z}_{PQ}$ and sends X to X^r. The automorphism keys are of the form $\mathsf{ksk}_{\sigma_r(s)}$. This allows others to rotate and conjugate ciphertexts without knowing s.
- **Relinearization key.** The relinearization key is ksk_{s^2}. This allows others to multiply ciphertexts without knowing s.

4 Distributed key generation

In distributed key generation, each participant P_i generates their own secret share s_i. The participants jointly compute the public key and keyswitching keys corresponding to the group secret $s = \sum_i s_i$.

4.1 Public key

The participants agree on a randomly generated $a \in \mathbb{Z}_Q[X]/(X^N+1)$.

Each participant P_i randomly generates error e_i and sends $b_i = -as_i + e_i$. The server computes $b = \sum_i b_i$. The public key is (a, b). Indeed, we have

$$as + b = as + \sum_i (-as_i + e_i) \approx 0.$$

4.2 Generating automorphism keys

To generate the automorphism keyswitching keys, it suffices to do the following:

Generating a digit of an automorphism key

Given $w \in \{w_1, \ldots, w_k\}$ and $r \in \mathbb{Z}$, generate (a, b) such that

$$as + b \approx wP\sigma_r(s).$$

Now, we will look at how to do this for a given r and w. The participants agree on a randomly generated $a \in \mathbb{Z}_{PQ}[X]/(X^N+1)$. Each participant P_i generates error e_i and reveals $b_i = wP\sigma_r(s_i) - as_i + e_i$. Then, the server computes

$$b = \sum_i b_i \approx wP\sigma_r(s) - as.$$

Then,

$$as + b \approx wP\sigma_r(s)$$

as desired.

4.3 Relinearization key

Generating the relinearization key requires two rounds.

In the first round, the participants agree on a randomly generated $a' \in \mathbb{Z}_{PQ}[X]/(X^N+1)$. Each P_i randomly generates an auxiliary secret u_i with small coefficients. Let $u = \sum_i u_i$.

Each P_i generates small error $e_{0,i}, e_{1,i}$ and reveals

$$h_{0,i} = -u_i a' + s_i wP + e_{0,i}$$
$$h_{1,i} = s_i a' + e_{1,i}.$$

Then, the server computes

$$h_0 = \sum_i h_{0,i} \approx -ua' + wPs$$
$$h_1 = \sum_i h_{1,i} \approx a's$$

and sends this to the participants. Then, P_i generates small error $e_{2,i}$ and reveals

$$b_i = s_i h_0 + (u_i - s_i)h_1 + e_{2,i} \approx -ua's_i + wPss_i + a'su_i - a'ss_i$$

The server computes

$$a = h_1 \approx a's$$
$$b = \sum_i b_i \approx -ua's + wPs^2 + a'su - a's^2 = wPs^2 - a's^2.$$

Then,

$$as + b \approx wPs^2$$

as desired.

4.4 Decryption

Once the server computes the final result (a, b), the participants jointly decrypt it. Each participant P_i reveals their decryption share

$$d_i = as_i + e_i,$$

where e_i is a randomly generated error. Then, the decrypted result is

$$\left(\sum_i d_i\right) + b \approx as + b.$$

Note

Anyone who sees all of the d_i's can determine the decrypted result of the computation. Often, this is fine, because the participants want to keep their inputs to the computation private, but don't care about the output being revealed.

Noise flooding

Consider the earlier example. Assume there are 32768 possible locations for a satellite. These can be mapped onto the 32768 slots of a ciphertext. Aliceland and Bobania can submit ciphertexts that have a 1 in each slot corresponding to a location where they have a satellite, and 0's elsewhere. Carolistan multiplies the ciphertext and returns it to reveal the locations where Aliceland and Bobania's satellites might crash.

The slots in Aliceland's ciphertext that are supposed to be 0 will contain some small noise. When this is multiplied with Bobania's ciphertext, this noise is multiplied by 1 or 0.

By looking at how large the noise in that slot of the decrypted result, Aliceland might be able to guess whether Bobania's ciphertext contained a 0 or a 1 there. So, the noise of the output can leak information about the input.

The solution to this is noise flooding, where the e_i that each participant adds to their decryption share is large enough to hide the noise accumulated in the computation.

It is also possible to do decryption in such a way that only one participant finds out the result. Every participant computes their decryption share, and participants $P_2, \ldots, P_n$ reveal theirs. Then, P_1 can compute $(\sum_i d_i) + b$. This way, only P_1 knows the result of the computation. Neither the other participants nor anyone intercepting communications between the participants knows the result.

5 Adversarial server

Assume participants $P_1, \ldots, P_n$ agree on some circuit C to run on their input ciphertexts $\mathfrak{c}_1, \ldots, \mathfrak{c}_n$.

If the server does what it is supposed to do, then it will output a ciphertext $\mathfrak{c}_{\text{result}}$ that encodes $C(\mathfrak{c}_1, \ldots, \mathfrak{c}_n)$. The participants jointly decrypt this. They discover the output of the circuit, but get no further information about each others' inputs.

An adversarial server can leak information. For example, P_1 and the server

can collude to reveal P_2's secret input.

The server can add a multiple of the public key and a small amount of noise to $\mathfrak{c}_2$. This produces a new ciphertext that encrypts the same message as $\mathfrak{c}_2$. The server outputs this ciphertext, pretending that it is the output of the computation it was supposed to run. The participants cannot distinguish this from the correct output. When they jointly decrypt it, it will reveal P_2's input. Therefore, the participants should only agree to decrypt an output ciphertext if they trust the server that computed it.

Side note

The server can publish a SNARK (succinct non-interactive argument of knowledge), which is a short proof that it ran the computation correctly. The participants can check this proof in a much shorter amount of time than the time it took to run the computation. If the server did not run the computation correctly, participants can refuse to reveal their decryption shares. This way, participants can avoid leaking information even if they do not trust the server.

6 Adversarial participants

An adversarial participant can also try to leak information by lying in their input ciphertext.

Consider the earlier example, where Aliceland and Bobania's inputs are are a binary vector of their satellite locations. If the government of Aliceland is malicious, they can submit a ciphertext of all 1's. Then, the output will be equal to Bobania's entire input.

To alleviate this issue, Carolistan can ensure that each participant has a valid input. Assume Carolistan knows that each country has 100 satellites. Then, a valid input consists of 100 ones and 32668 zeros.

The government of Carolistan can run a circuit that outputs 1 if both participants submitted a valid input, and 0 otherwise. Then, it multiplies this by the product of the inputs.

This way, if either participant tried to cheat, the output will be all zeroes, which reveals nothing. Aliceland could still cheat by lying about which 100 locations they have satellites in, but this will still only reveal 100 slots of Bobania's ciphertext, rather than revealing all 32768.

7 Example

In this section, we'll review the procedure using the example from the first section. The governments of Aliceland and Bobania are the participants, and the government of Carolistan is the server.

- Aliceland and Bobania generate secret shares s_a and s_b, respectively. Their group secret is $s = s_a + s_b$.
- They run distributed key generation to generate a public key and keyswitching keys for s.
- The two governments each encrypt their satellite data under the public key and send the ciphertexts to Carolistan.
- Using the keyswitching keys generated during distributed key generation, Carolistan computes and reveals a ciphertext encoding the locations where both countries have a satellite. Carolistan can also compute a circuit that zeros out this output if either participant submitted an invalid input.
- Aliceland and Bobania both reveal noise-flooded decryption shares of the result. Carolistan uses this to compute the decryption of the result, which it sends back to the two governments.

Consider a different case, where each government wants to know which of its own satellites is in danger of crashing into one of the other government's satellites.

- Aliceland and Bobania generate secret shares s_a and s_b, respectively. Their group secret is $s = s_a + s_b$.
- They run distributed key generation to generate a public key and keyswitching keys for s.
- The two governments each encrypt their satellite data under the public key and send the ciphertexts to Carolistan.
- Using the keyswitching keys generated during distributed key generation, Carolistan computes and reveals:
  - A ciphertext output_a encoding the set of Aliceland's satellites that are in danger of colliding into Bobania's satellites.
  - A ciphertext output_b encoding the set of Bobania's satellites that are in danger of colliding into Aliceland's satellites.
- Aliceland computes its decryption share of output_b, noise-floods it, and sends it to Bobania. Similarly, Bobania sends a noise-flooded decryption share of output_a.
- Aliceland uses Bobania's decryption share to decrypt output_a. Similarly, Bobania computes output_b.