CS523 Project 1

EPFL Laboratory for Data Security

February 10th, 2020

1 Introduction

This project aims at designing a N-party multiparty computations (MPC) engine in a semi-honest (passive) adversarial setting in Go programming language [1].

The project is divided into two parts. The first is to design a MPC framework that works for generic circuits assuming the existence of a trusted third-party. The second relaxes this strong condition by using homomorphic encryption to replace the trusted third-party by a pre-processing phase.

A skeleton of the implementation is given to you to ease your implementation:

`https://github.com/ldsec/CS523-Project1`

You need to fill the code sections corresponding to each of the step of this project. You are free to implement your own structures as long as it is consistent with the API we provide for the circuits and the network for our testing purposes.

2 Project objectives

1. **Part I:** Design a simple MPC framework supporting basic arithmetic circuits over additively secret-shared data with a trusted third-party providing the Beaver multiplication triplets.
2. **Part II:** Remove the need for a trusted third-party by implementing a pre-processing phase layer using BFV homomorphic encryption with Lattigo library to produce Beaver triplets.
3. Evaluate the trade-off between the first and second approaches.
4. *(Optional, not graded) Implement an approach based solely on fully homomorphic encryption that enables evaluating the circuits homomorphically, and make comparison with the previous approach.*

3 Deliverables

1. A 3-pages IEEE two columns handout on the provided skeleton.
2. An archive containing a working implementation in Go of the different parts of the project, and a README file.

4 Estimated Timeline

- Week 1: Understand the project, create the gate and wire structures.
- Week 2-3: Design the MPC framework (circuit evaluation with N parties and trusted third-party)
- Week 3-4: Design the pre-processing phase.
- Week 5: Handout and wrap-up the archive.

[1] `https://golang.org/`

5 Secure MPC with trusted third party for Beaver Generation

In this part of the project, you will implement the necessary components for multi-party computations (MPC) on generic circuits assuming the existence of a trusted third-party and additive secret sharing. Recall that additive secret sharing allows one to secretly share a *sub*-secret amongst several entities such that only the sum of all sub-secrets yields the original secret. We denote by $[x]$ the additive secret sharing of a value x among N parties. More precisely, $[x] = x_0, ..., x_{N-1}$ such that $x = \sum_{k=0}^{N-1} x_k$.

We consider generic circuits made of additions (ADD), additions by constant (ADDCST), subtractions (SUB), multiplication by constants (MULTCST), and multiplications (MULT). We denote the operation of reconstructing a secret from the additive secret sharing by REVEAL.

5.1 Initialisation

Retrieve the code from https://github.com/ldsec/CS523-Project1. It contains a dummy MPC protocol evaluating the sum of inputs held by three parties. Get familiar with the code and how it is structured. In particular, you should understand how the communication between the different parties is realized.

In order to build the project, run from terminal in the working directory:

```
go build .
```

To run the executable on inputs x_i for $i \in \{0, 1, 2\}$, type:

```
./mpc 0 x_0  ./mpc 1 x_1  ./mpc 2 x_2
```

Please note that the input x_i should be an integer.

To directly use the go-test, type:

```
go test -v .
```

As a rule of thumb, you should understand what the different components of the code are doing.

5.2 Create an additive secure MPC

Starting from the dummy code, we suggest you create a secure MPC protocol able to run addition gates.

An addition gates takes two inputs and return its sum. Recall, that for a secret sharing of $[x]$ and $[y]$, a party can just locally compute:

$$[x + y] = [x] + [y]$$

In order to tackle this task, create a similar file to dummy.go. It should handle the additive secret sharing of inputs and the execution of the MPC protocol. In this file, there should be an evaluation protocol for gates (keep in mind that different kind of gates will be used once the project is completed). To this end, it might be of relevance to create a file gates.go to implement the gate logic. Note that you also need to implement a reveal gate that does the reconstruction of the additive secret sharing at the end of the protocol.

Also create a file circuit.go that "reads" the circuits defined as in test_circuits.go[2].

Finally, modify main.go and dummy_test.go to accommodate the changes.

Note that your a free to implement the way you want. The only requirement is that your code should interface properly with the API we gave for inputting the different circuit.

That is, running :

```
go test -v -run=^TestEval$/^circuit1$,
```

should execute a test on circuit 1 defined in test_circuits.go.

[2]Description of the circuits available in Table 1

5.3 Adding more gates

Now that you have a working code running Secure MPC for addition gates, you can add additional gates:

- Subtractions: $[x - y] = [x] - [y]$
- Scalar multiplications: $[k \cdot x] = k \cdot [x]$

This requires you to add additional gates, evaluation protocols, and modify the circuit loader. You can test your implementation against two circuits: The subtraction :

`go test -v -run=^TestEval$/^circuit2$,`

The constant multiplication:

`go test -v -run=^TestEval$/^circuit3$,`

5.4 Adding addition by a constant

Now, you can add another gate that can be computed locally by each player: the addition by a constant.

Note however that this requires a bit more thinking that the classic addition gate since not all parties have the same role. For a constant k, the secret-sharing of the constant addition is :

$$[x + k] = \{x_0 + k, x_1, ..., x_{N-1}\}$$

Notice that only one of the players is adding the constant.
You can test your implementation against circuit 4:

`go test -v -run=^TestEval$/^circuit4$,`

5.5 Multiplication Gate

We can now add multiplication gates that take two inputs and return their product. This operation requires the use of "blinding" values a, b, and c sampled randomly such that $ab = c$. Those values are also additively shared and called Beaver triplets $[a]$, $[b]$, and $[c]$.

Given inputs $[x]$ and $[y]$, proceed to:

1. For each party, generate locally the share $[x - a]$ and broadcast it.
2. For each party, generate locally the share $[y - b]$ and broadcast it.
3. For each party reconstruct from the broadcast shares $(x - a)$ and $(y - b)$ using the REVEAL procedure.
4. Each player computes locally:

$$[z] = [c] + [x] \times (y - b) + [y] \times (x - a) - (x - a)(y - b)$$

NOTE: while the term $-(x - a)(y - b)$ is known by all the parties, it must be added locally by only one party or the result will be off by $(n-1) \cdot (-(x-a)(y-b))$. Therefor a multiplicative gate also contains an addition by constant gate.

One can verify that $[z]$ is indeed the additive secret sharing of $[xy]$. Since the values a, b, and c are sampled uniformly random, $(x - a)$ and $(y - b)$ are protecting the values of x and y providing information-theoretic privacy for the inputs.

In this section (Part 5), we assume that a trusted third-party is generating the Beaver triplets and providing to each party its share of $[a]$, $[b]$, and $[c]$ for all the multiplicative gates in the circuit[3].

Here are roughly the steps you need to follow:

- Generate an beaver.go file similar to the initial dummy.go. This file should handle the generation and exchange of the beaver triplets. Note that you need to import Lattigo ring to generate proper random values:

[3] Note that each multiplication gate requires a new triplet

Name	Parties	Function
circuit1.go	3	$f(a,b,c) = a + b + c$
circuit2.go	2	$f(a,b) = a - b$
circuit3.go	3	$f(a,b,c) = (a + b + c) * K$
circuit4.go	3	$f(a,b,c) = (a + b + c) + K$
circuit5.go	3	$f(a,b,c) = (a * K0 + b - c) + K1$
circuit6.go	4	$f(a,b,c,d) = a + b + c + d$
circuit7.go	3	$f(a,b,c) = (a * b) + (b * c) + (c * a)$
circuit8.go	5	$f(a,b,c,d,e) = ((a + K0) + b * K1 - c) * (d + e)$

Table 1: Description of the test circuits

```
import("github.com/ldsec/lattigo/ring")
```

- Modify the list of gates to be able to read them from the circuit description
- Modify you online.go file to add the beaver triplets binding, and the handling of multiplication in the evaluation.
- Finally modify main.go according to your implementation.
- Test your implementation on circuit 7 and 8:

```
go test -v -run=^TestEval$/^circuit7$,
```

5.6 Complex circuits

Now you should have a very versatile implementation on secure MPC handling a wide range of circuits. You can create different circuits in text_circuits.go. Please create a scenario and a circuit that you would describe in the handout. Your circuit should contain at least one of each gate and multiple parties.

6 Part II

In this part, you will design a new layer for the Beaver triplets generation using the BFV homomorphic encryption scheme. To help you in this task, we provide you with a Go library handling the BFV homomorphic encryption. You can download the library through `https://github.com/ldsec/lattigo`.

6.1 Introduction

The BFV (Brakerski-Fan-Vercauteren[4]) is the name of a somewhat fully homomorphic cryptographic scheme that enables modular arithmetic over vectors of integers in the encrypted domain in a SIMD[5] fashion:

$$\begin{aligned} \mathsf{Dec}(\mathsf{Enc}(a) + \mathsf{Enc}(b)) &= a + b \mod p, \\ \mathsf{Dec}(\mathsf{Enc}(a) \otimes \mathsf{Enc}(b)) &= a \odot b \mod p, \end{aligned}$$

where a and b are two vectors of n integers and p is a prime, $\odot$ represents a component-wise product and $\otimes$ represents a nega-cyclic convolution. It is "somewhat" homomorphic in the sense that only a limited number of operations can be carried out to enable correct decryption, but "fully" homomorphic in the sense that it supports both homomorphic additions and multiplications. The advantage of such a scheme is that the evaluation of an arithmetic circuit can be carried in the encrypted domain without requiring the knowledge of the secret-key (however special public-keys might be needed for some type of operations). The BFV scheme enables homomorphic operations between multiple ciphertexts, but also between ciphertexts and plaintexts. We will use the latter in this exercise and constraint the operations to additions and multiplications (other operations

[4]To read more about the BFV scheme : `https://eprint.iacr.org/2012/144`

[5]Rather than encrypting a single value, a ciphertext encrypts a vector of values and operations are carried in parallel on all the values, improving the efficiency of the scheme.

are possible, e.g., vector rotations).

In the following sections, you will be given the protocol to implement and the steps to follow to implement it. Technical details about the BFV scheme have been left aside on purpose as they are not a requirement of this project. You will, however, find in the appendix some mathematical background about the BFV scheme. We encourage you to look at them.

6.2 Notations

We work with elements of the ring $R = \mathbb{Z}_Q[X]/X^N + 1$, where N is a power of 2. R is the set of all polynomials with integer coefficients modulo Q taken modulo the polynomial $X^N + 1$. Elements of the ring R will be denoted with lowercase bold, e.g. $\mathbf{a} \in R$. The values to be encrypted are vectors of integers of N elements modulo t and will be denoted with lowercase italic, e.g. $a \in \mathbb{Z}_t^d$. We denote $\leftarrow \mathbb{Z}_p^n$ the sampling of a uniform vector of integers modulo p of n elements, $\leftarrow \chi_R$ the sampling of a uniformly distributed element of R, $\leftarrow \chi_\rho$ the sampling of an element of R with ternary coefficients [-1, 0, 1] with the distribution $[(1-\rho)/2, \rho, (1-\rho)/2]$ and $\leftarrow \chi_{err}$ the sampling an element of R with a truncated Gaussian distribution of variance σ and bound B. $\lfloor x \rfloor$, $\lceil x \rceil$ and $\lceil x \rfloor$ respectively denote the flooring, ceiling, and rounding of x to the nearest integer, while $\odot$, $\otimes$ and $\cdot$ respectively denote the component-wise multiplication, the polynomial nega-cyclic convolution in R and the multiplication by a constant.

We denote the operations of sending data to P and receiving data from P as $\Rightarrow P$ and $\Leftarrow P$, respectively. We denote the encoding and decoding operation between $\mathbb{Z}_t^n \leftrightarrow R$ as $\boldsymbol{\phi}$ and $\boldsymbol{\phi}^{-1}$, respectively.

6.3 The protocol

Algorithm 1: Naive beaver's triple generation

Input: P a set of ℓ parties who have agreed on a set of public parameters for the BFV scheme, notably on n the degree of R and t the plaintext modulus.

Output: The shared values c_i such that $c = \sum_{i=0}^{\ell-1} c_i = \sum_{i=0}^{\ell-1} a_i \cdot \sum_{i=0}^{\ell-1} b_i$.

foreach P_i **do**
  $a_i \leftarrow \mathbb{Z}_t^n$
  $b_i \leftarrow \mathbb{Z}_t^n$
  $c_i = a_i \odot b_i$
  $\mathbf{a}_i = \boldsymbol{\phi}(a_i)$
  $\mathbf{b}_i = \boldsymbol{\phi}(b_i)$
  $\mathbf{sk}_i \leftarrow \chi_{1/3}$
  $\mathbf{d}_i = \mathsf{Enc}_{\mathbf{sk}_i}(\mathbf{a}_i) \in R^2$
  foreach $P_j \neq P_i$ **do**
    $\mathbf{d}_i \Rightarrow P_j$
  end
end
foreach P_i **do**
  foreach $P_j \neq P_i$ **do**
    $\mathbf{d}_j \Leftarrow P_j$
    $r_{ij} \leftarrow \mathbb{Z}_t^n$
    $c_i = c_i - r_{ij}$
    $\mathbf{r}_{ij} = \boldsymbol{\phi}(r_{ij})$
    $(\mathbf{e}_{ij}^0, \mathbf{e}_{ij}^1) \leftarrow \chi_{err} \in R^2$
    $\mathbf{d}_{ij} = \mathsf{Add}(\mathsf{Mul}(\mathbf{d}_j, \mathbf{b}_i), \mathbf{r}_{ij}) + (\mathbf{e}_{ij}^0, \mathbf{e}_{ij}^1)$
    $\mathbf{d}_{ij} \Rightarrow P_j$
  end
end
foreach P_i **do**
  $\mathbf{c}' = (0, 0) \in R^2$
  foreach $P_j \neq P_i$ **do**
    $\mathbf{d}_{ji} \Leftarrow P_j$
    $\mathbf{c}' = \mathsf{Add}(\mathbf{c}', \mathbf{d}_{ij})$
  end
  $c_i = c_i + \boldsymbol{\phi}^{-1}(\mathsf{Dec}_{\mathbf{sk}_i}(\mathbf{c}'))$
end

6.4 Implementing the protocol

6.4.1 Scheme overview

The BFV scheme enables arithmetic modulo T over vectors of integers. It means that you will be able to encrypt a slice of `uint64` modulo some pre-defined value T and do operation in the encrypted domain on each slot in parallel. By parallel, we mean that given $a = (a_0, a_1, \ldots, a_{N-1})$ and $b = (b_0, b_1, \ldots, b_{N-1})$ two slice of `uint64` modulo T, we have

$$a + b = (a_0 + b_0 \mod \mathsf{T}, a_1 + b_1 \mod \mathsf{T}, \ldots, a_{N-1} + b_{N-1} \mod \mathsf{T}),$$
$$a \odot b = (a_0 \cdot b_0 \mod \mathsf{T}, a_1 \cdot b_1 \mod \mathsf{T}, \ldots, a_{N-1} \cdot b_{N-1} \mod \mathsf{T}).$$

The BFV scheme bases its security on the $\mathsf{R\text{-}LWE}$[6] (Ring Learning with Errors) problem, which implies that ciphertexts are inherently noisy. By noisy we mean that they carry some error with them, which grows after each homomorphic operation. Thus, only a finite number of homomorphic operations can be carried before this error becomes too large to allow correct decryption. To enable more homomorphic operations we must increase the size of the parameters and this has a direct impact on the efficiency of the scheme. But it also means that for any circuit that could be evaluated, there exists a set of parameters enabling this evaluation, regardless of how big and impractical it could be. The circuit that you will implement (the protocol) has low complexity, so you will be able to use small parameters, enabling fast, and efficient computations.

[6] `https://en.wikipedia.org/wiki/Ring_learning_with_errors`

6.4.2 Parameters

The BFV scheme needs to be configured with several parameters that affect both its security and efficiency. The parameters defining the security are the ring degree (N), the ciphertext's modulus (Q), the extra modulus for the keys (P), and the variance of the error (σ) which is a fixed value. Other parameters which are not relevant to the security are the extended modulus (QMul) used during the homomorphic multiplication, and the plaintext modulus (T). We summarize the impact of all these parameters on the security, efficiency, and homomorphic capacity in the **Table 2**.

Parameters impact			
	Security	Efficiency	H. Capacity
N	✓	✗	✗
Q	✗	✗	✓
P	✗	✓	-
σ	✓	-	✗
QMul	-	✗	-
T	-	-	✗

Table 2: Impact of increasing each individual parameter on the security, efficiency, and homomorphic capacity of the scheme.

The first step you need to do is to choose a set of parameters, which is not a trivial operation. Luckily, we did that for you, and we provide you with suitable parameters in your files. Those parameters are called PN13QP218. This string means that the degree of the underlying ring is 2^{13} and that the total bit-size of the modulus QP is 218. This set of parameters ensures a security of 128 bits against the most recent attacks and has a homomorphic capacity suited for our use case. You can access the default parameters using

```
params := bfv.DefaultParams[bfv.PN13QP218]
```

This will return a struct on which several methods can be called to access values and/or details about the parameters. You can find the complete set of methods and their description in the file lattigo/bfv/params.go. This set of parameters also defines T which is set to 65537 by default. You can access it with params.T. If you want to change T, you must make sure that T is prime and that $T \equiv 1 \mod 2N$. Be aware that larger values for T might not give enough homomorphic capacity for this project.

6.5 Steps

Since the BFV scheme encrypts vectors of several thousand values, a single run of the protocol will produce many Beaver triplets at once.

To be able to successfully implement this protocol you need to:

1. Create methods to manipulate the vector of integers. Namely, you should be able to sample random vectors in $\mathbb{Z}_t^n$ and add, subtract, and multiply (point-wise) vectors in $\mathbb{Z}_t^n$.
2. Understand how to instantiate and use the API of the BFV scheme of Lattigo.
3. Make use of your newly acquired knowledge from **Part I** about go channels and go routines to simulate a network and interactions between the participants of the protocol.

6.5.1 Vector operations

First you need to create methods that allow you to manipulate vectors of integers:

- newRandomVec(n,T uint64)([ ]uint64) : a method that generates a new slice of n uint64 variables uniformly distributed in $\mathbb{Z}_T$.
- addVec(a, b [ ]uint64, T uint64)([ ]uint64) : a method that takes two slices a and b of uint64 of same length and returns a new slice of uint64 which is the result of a + b mod T.
- subVec(a, b [ ]uint64, T uint64)([ ]uint64) : a method that takes two slices a and b of uint64 of same length and returns a new slice of uint64 which is the result of a - b mod T.

- mulVec(a, b []uint64, T uint64)([]uint64) : a method that takes two slices a and b of uint64 of same length and returns a new slice of uint64 which is the result of $\mathsf{a} \odot \mathsf{b} \bmod \mathsf{T}$.
- negVec(a []uint64, T uint64)([]uint64) : a method that takes a slice a of uint64 and returns a new slice of uint64 which is the result of -a mod T.

You can use the native % Go operator for the modular arithmetic, however be aware that overflow might occur if T is larger than 32 bits. **Optional**: to be able to use larger T you will need to implement your own modular arithmetic (or import the one use by the package Ring of Lattigo).

6.5.2 Protocol structs

Then you will need to the following Go structs and methods:

- struct BeaverProtocol : a Go struct able to call Lattigo and storing all the necessary data that can be re-used between two runs of the protocol. Similar to DummyProtocol in the dummy example.
- struct BeaverRemoteParty : an abstract representation of a distant party, storing its Go channel for the protocol. Similar to DummyRemote in the dummy example.
- struct BeaverMessage : the message type to be exchanged between the parties during the protocol. Similar to DummyMessage in the dummy example.
- struct BeaverInputs : a struct that stores the new inputs that have to be generated for each new call of the protocol
- func New : a method to create a new protocol struct. Similar to NewDummyProtocol in the dummy example.
- func GenInput : a method to generate new inputs for each new call of the protocol.
- func Run : a method to run the protocol. Similar to the one with the same name in the dummy example. When it terminates all participant of the protocol should end up with a list beaver triplets shares.
- func BindNetwork : a method to interface the Go channels used by the protocol and the network layer. Similar to the one used in the dummy example.

This list's purpose is to provide you with a guideline and ease your implementation. You are free to use different names and even different structs and/or methods as long as your code correctly implements the protocol described in Algorithm 1 and terminates with all the participants having their own shares of the beaver triplets.

A Appendix

A.1 BFV

A.1.1 The NTT transformation

Before we describe the scheme more in depth, we must introduce an important operation: the Number Theoretic Transformation[7] (NTT), which is the equivalent of the FFT but in a finite field (in our case $\mathbb{Z}_p$ with p prime). It is a linear transformation on polynomials that can be computed in $O(n \log(n))$ steps, where n is the polynomial degree. It provides the following properties:

$$\begin{aligned} \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \odot \mathsf{NTT}(\mathbf{b})) &= \mathbf{a} \otimes \mathbf{b} \mod p, \\ \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) + \mathsf{NTT}(\mathbf{b})) &= \mathbf{a} + \mathbf{b} \mod p, \\ \mathsf{NTT}(\lambda \cdot \mathbf{a}) &= \lambda \cdot \mathsf{NTT}(\mathbf{a}) \mod p. \end{aligned}$$

This transformation plays a central role in the efficiency of the scheme because it enables doing polynomial multiplications with a quasi-linear complexity and it is also used during the encoding process.

[7] To read more about the NTT and its implementation: `https://eprint.iacr.org/2016/504`

A.1.2 Plaintexts and ciphertexts as polynomials of R

Plaintexts, and ciphertexts will be interpreted as the coefficients of a polynomials in basis $R = \mathbb{Z}_Q[X]/X^N + 1$. A plaintext is always an element of R (of degree zero) and a fresh ciphertext an element of R^2 (of degree one). A ciphertext's degree can grow during the homomorphic operations. This interpretation as polynomials in basis R is important and will help describe the homomorphic operations and how they behave. Given a an element $\mathbf{a} = (\mathbf{a}^0, \dots, \mathbf{a}^{n-1}) \in R^n$, we will denote $\mathbf{a}^i$ its i-th.

A.1.3 Encoding

We can not directly encrypt a vector of integer as it not an element of R. The encoding step constructs a map between $\mathbb{Z}_t^N \longleftrightarrow R$. It maps $\omega \in \mathbb{Z}_t^N$ to $\mathbf{w} \in R$ while ensuring the following properties:

$$\begin{aligned} \mathsf{Decode}(\lceil (t/Q) \cdot (\mathsf{Encode}(\omega_0) \otimes \mathsf{Encode}(\omega_1) \mod Q^2) \rfloor \mod Q) &= \omega_0 \odot \omega_1 \mod t, \\ \mathsf{Decode}(\mathsf{Encode}(\omega_0) + \mathsf{Encode}(\omega_1) \mod Q) &= \omega_0 + \omega_1 \mod t, \end{aligned}$$

where $\odot$ is a component-wise product and $\otimes$ is a nega-cyclic convolution in the ring R. To enable such an encoding, the modulus t must be a prime congruent to 1 mod $2N$.
The encoding can be described as

$$\mathbf{w} = \mathsf{Encode}(\omega) = \left\lfloor \frac{Q}{t} \right\rfloor \cdot \mathsf{NTT}^{-1}(\omega),$$

where NTT^{-1} is the inverse nega-cyclic NTT in the ring $\mathbb{Z}_t[X]/X^N + 1$. The decoding function is

$$\omega = \mathsf{Decode}(\mathbf{w}) = \mathsf{NTT}\left(\left\lceil \frac{t}{Q} \cdot \mathbf{w} \right\rfloor \right),$$

where NTT is the nega-cyclic NTT in the ring $\mathbb{Z}_t[X]/X^N + 1$.

It is important to note that the integer arithmetic between two encoded vectors is done modulo Q and not modulo t. Indeed, the vector to encode is first treated as a polynomial of $\mathbb{Z}_t[X]/X^N + 1$, passed through an NTT in the field $\mathbb{Z}_t$ and then scaled up by $\lfloor Q/t \rfloor$ to map it to a polynomial of R and enable the arithmetic to be carried modulo Q. The decoding process is the inverse operation of the encoding. Every time two encoded vectors are multiplied together, a division by t/Q rounded to the nearest integer must be done on the resulting vector to cancel the additional t/Q factor that arises from this process.

A.1.4 Keys

A secret-key $\mathbf{sk}$ is an element of R sampled in $\chi_{1/3}$. A public-key $\mathbf{pk}$ is an element of R^2 of the form $\mathbf{pk} = (-\mathbf{a} \otimes \mathbf{sk} + \mathbf{e}, \mathbf{a})$ with $\mathbf{a} \leftarrow \chi_R$ and $\mathbf{e} \leftarrow \chi_{err}$. We observe that we can derive many public-keys from a single secret-key.

A.1.5 Encryption

To encrypt a plaintext $\mathbf{w}$ (an encoded vector of integers) using a public-key $\mathbf{pk} = (\mathbf{pk}^0, \mathbf{pk}^1)$ we compute

$$\mathsf{Enc}_{\mathbf{pk}}(\mathbf{w}) = (\mathbf{pk}^0 \otimes \mathbf{u} + \mathbf{w} + \mathbf{e}_0, \mathbf{pk}^1 \otimes \mathbf{u} + \mathbf{e}_1),$$

with $\mathbf{u} \leftarrow \chi_{0.5}$ and $\mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi_{err}$.
It is also possible to encrypt $\mathbf{w}$ using only the secret-key by computing

$$\mathsf{Enc}_{\mathbf{sk}}(\mathbf{w}) = (-\mathbf{sk} \otimes \mathbf{a} + \mathbf{w} + \mathbf{e}, \mathbf{a}),$$

with $\mathbf{a} \leftarrow \chi_R$ and $\mathbf{e} \leftarrow \chi_{err}$. Encrypting with the secret-key introduces less noise and can allow for more homomorphic operations.

A.1.6 Decryption

To decrypt a ciphertext $\mathbf{ct} \in R^n$, we evaluate the polyomial $\mathbf{ct}$ at $\mathbf{sk}$:

$$\mathsf{Dec}_{\mathbf{sk}}(\mathbf{ct}) = \sum_{i=0}^{n-1} \mathbf{ct}^i \otimes \mathbf{sk}^i = \mathbf{w} + \mathbf{e},$$

where $\mathbf{ct}^i$ is the i-th element of $\mathbf{ct}$, $\mathbf{sk}^i$ is the i-th power of $\mathbf{sk}$, $\mathbf{w}$ is the plaintext and $\mathbf{e}$ is the residual error, that will be removed by the decoding process as long as $\lceil \Delta^{-1} \cdot \mathbf{e} \rfloor = 0$. The decryption process can be efficiently computed by storing only $\mathbf{sk}$ and using Horner's method for polynomial evaluation.

A.1.7 Homomorphic addition

The homomorphic addition between two ciphertexts $\mathbf{ct}_0, \mathbf{ct}_1 \in R^2$ is done as:

$$\mathsf{Add}(\mathbf{ct}_0, \mathbf{ct}_1) = (\mathbf{ct}_0^0 + \mathbf{ct}_1^0, \mathbf{ct}_0^1 + \mathbf{ct}_1^1).$$

More generally, it is possible to add two ciphertexts $\mathbf{ct}_0 \in R^n$ and $\mathbf{ct}_1 \in R^m$ (for this example $1 < n < m$):

$$\mathsf{Add}(\mathbf{ct}_0, \mathbf{ct}_1) = (\mathbf{ct}_0^0 + \mathbf{ct}_1^0, \ldots, \mathbf{ct}_0^{n-1} + \mathbf{ct}_1^{n-1}, \ldots, \mathbf{ct}_1^{m-1}).$$

It is also possible to add a plaintext $\mathbf{pt} \in R$ to a ciphertext $\mathbf{ct}_0 \in R^n$:

$$\mathsf{Add}(\mathbf{ct}_0, \mathbf{pt}) = (\mathbf{ct}_0^0 + \mathbf{pt}, \mathbf{ct}_0^1, \ldots, \mathbf{ct}_0^{n-1})$$

A.1.8 Homomorphic multiplication

The homomorphic multiplication between two ciphertexts $\mathbf{ct}_0, \mathbf{ct}_1 \in R^2$ is done as:

1. Compute <u>without modular reduction</u>

$$\begin{aligned} c_0 &= \mathbf{ct}_0^0 \otimes \mathbf{ct}_1^0 \\ c_1 &= \mathbf{ct}_0^1 \otimes \mathbf{ct}_1^0 + \mathbf{ct}_0^0 \otimes \mathbf{ct}_1^1 \\ c_2 &= \mathbf{ct}_0^1 \otimes \mathbf{ct}_1^1 \end{aligned}$$

2. Compute

$$\begin{aligned} c_0' &= \left\lceil \frac{t}{Q} \cdot c_0 \right\rfloor \mod Q \\ c_1' &= \left\lceil \frac{t}{Q} \cdot c_1 \right\rfloor \mod Q \\ c_2' &= \left\lceil \frac{t}{Q} \cdot c_2 \right\rfloor \mod Q \end{aligned}$$

3. Return $\mathsf{Mul}(\mathbf{ct}_0, \mathbf{ct}_1) = (c_0', c_1', c_2')$

We observe that the result is consistent with a polynomial convolution in basis R between $\mathbf{ct}_0 \in R^2$ and $\mathbf{ct}_1 \in R^2$ and that the result has grown by one element. By generalizing this operation, it is possible to multiply two ciphertexts of arbitrary degree together. The resulting ciphertext's degree will be $\deg(\mathbf{ct}_2) = \deg(\mathbf{ct}_0) + \deg(\mathbf{ct}_1)$. The degree grows very fast, along with the complexity of the operation and the size of the ciphertexts, but it is possible to reduce it back to the original degree of one. This operation, called "relinearization", is doing a partial decryption of the ciphertext using an encryption of the secret-key under itself.

Similarly to the homomorphic addition, we can multiply a ciphertext $\mathbf{ct}_0 \in R^2$ with a plaintext $\mathbf{pt} \in R$ as:

1. Compute <u>without modular reduction</u>

$$\begin{aligned} c_0 &= \mathbf{ct}_0^0 \otimes \mathbf{pt} \\ c_1 &= \mathbf{ct}_0^1 \otimes \mathbf{pt} \end{aligned}$$

2. Compute

$$c_0' = \left\lceil \frac{t}{Q} \cdot c_0 \right\rfloor \mod Q$$
$$c_1' = \left\lceil \frac{t}{Q} \cdot c_1 \right\rfloor \mod Q$$

3. Return $\mathsf{Mul}(\mathbf{ct}_0, \mathbf{pt}) = (c_0', c_1')$