Multiparty Computation based on Ring-LWE

Mikkel Gaba, Marcus Sellebjerg, Kasper Ilsøe

Project Report (10 ECTS) in Computer Science
Advisor: Ivan Damgård
Department of Computer Science, Aarhus University
May 22th, 2022

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

...

Mikkel Gaba, Marcus Sellebjergen, Kasper Ilsøe
Aarhus, May 22th, 2022.

Contents

Chapter 1

Introduction

...

Chapter 2

Review of literature

2.1 Ring LWE - A somewhat homomorphic encryption scheme

In the paper "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages" written by Zvika Brakerski and Vinod Vaikuntanathan [1] they describe a method for converting the Ring Learnig with errors (RingLWE) problem into an encryption scheme which reduces to the worst-case hardness of problems on ideal lattices. We'll shortly describe the encryption scheme here but will omit proofs and detailed discussions. Both system will be over the message space of $R_t = \mathbb{Z}_t[x]/\langle f(x)\rangle$.

2.1.1 Polynomial learning with errors

The polynomial learning with errors problem is made as a decision problem and is defined by in the Hermite normal form as the following.

The PLWE Assumption - Hermite Normal Form. for all $\kappa \in \mathbb{N}$, let $f(x) = f_\kappa(x) \in \mathbb{Z}[x]$ be a polynomial of degree n = n(κ), let $q = q(\kappa) \in \mathbb{Z}$ be a prime integer, let the ring $R = \mathbb{Z}[x]/\langle f(x)\rangle$ and $R_q = R/qR$ with χ denoting a distribution over the ring R. Then the Polynomial learning with error ($PLWE_{f,q,\chi}$) assumption can be defined as

$$\{(a_i, a_i \cdot s + e_i)\}_{i\in[l]} \approx^c \{(a_i, u_i)\}_{i\in[l]}\}$$

where s is sampled from χ, a_i is uniform in R_q, the error polynomials e_i are sampled from χ and u_i are random ring elements from R_q.

2.1.2 Symmetric version

Let κ be the security parameter and let further q and t be prime numbers where $t \in \mathbb{Z}_n^*$. We also need a polynomial of degree n $f(x) \in \mathbb{Z}[x]$ and an error distribution χ over the ring $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$, then we can define the following operations.

Key-gen

Let our secret key be a randomly sampled element from the error distribution $s \xleftarrow{\$} \chi$ Then given the security parameter κ sample a ring element s uniformly at random from κ and define the secret key vector by $(s^0, s^1, s^2, \ldots, s^D) \in R_q^{D+1}$.

Encryption

Rememer that all messages are encodeable in our message space R_t, thus we will encode our message m as a n degree polynomium with coefficient mod t. To encrypt we sample $(a, b = a \cdot s + t \cdot e)$ where $a \xleftarrow{\$} R_q$ and $e \xleftarrow{\$} \chi$, then compute

$$c_0 := b + m \qquad\qquad c_1 := -a$$

and from this output the ciphertext $\mathbf{c} := (c_0, c_1) \in R_q^2$.

Decryption

Note that a ciphertext is on the form $(c_0, c_1, \ldots, C_D) \in R_q^{D+1}$. Define the inner product over R_q as

$$\langle c, s \rangle = \sum_{i=0}^{D} c_i \cdot s^i$$

Then to decrypt, simply set m as the inner product of c and s and take modulo t.

$$m = \langle c, s \rangle \bmod t$$

m will then be the decrypted message.

Eval

To obtain the homomorphic abilities of the encryption scheme, Zvika Brakerski and Vinod Vaikuntanathan show how to obtain homomorphic addition and multiplication of ciphertexts.

Addition: Assume we have 2 ciphertexts $c \in R_q^{D+1}$ and $c' \in R_q^{D+1}$, then an encryption of the sum of the 2 underlying messages will be

$$c_{Add} = c + c' = (c_0 + c'_0, c_1 + c'_1, \ldots, c_d + c'_d) \qquad c_{Add} \in R_q^{D+1}$$

The decryption of c_{Add} will then be the sum of the unencrypted messages from c and c'.

Multiplication: Assume we have 2 ciphertexts $c \in R_q^{D+1}$ and $c' \in R_q^{D'+1}$ and let v be a symbolig value then calculate the updated ciphertext
$(\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_d + d') \in R_q^{D+D'+1}$ by

$$c_{mul} = (\sum_{i=0}^{D} c_i \cdot v^i) \cdot (\sum_{j=0}^{D'} c'_i \cdot v^i) = \sum_{i=0}^{D+D'} \hat{c}_i \cdot v^i \qquad c_{mul} \in R_q^{D+D'+1}$$

The output of the multiplication operation will then be $c_{mul} = (\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{D+D'})$

2.1.3 Public key version

To achieve a public scheme instead, we can make the following changes

- In the key generation we generate in addition to the secret key $sk = s \xleftarrow{\$} \chi$, we output a public key $pk = (a_0, b_0 = a_0 \cdot s + t \cdot e_0)$, where $a_0 \xleftarrow{\$} R_q, e_0 \xleftarrow{\$}$

- In the encryption algorithm, instead we use $(a_0 \cdot v + t \cdot e', b_0 \cdot v + t \cdot e'')$ where $v, e' \xleftarrow{\$} \chi$ and $e'' \xleftarrow{\$} \chi'$.

where we have then obtained a public key $pk = (a_0, a_0 \cdot s + t \cdot e_0)$ corresponding to the secret key $sk =$.

2.2 Circuit privacy

Each time we add or multiply ciphertexts the error term grows. As a result of this the error term will be larger for a ciphertext output by a call to **eval**, than for a ciphertext output by **encrypt**. This poses a problem, as an adversary might be able to derive information about the function computed by looking at the ciphertexts produced. To deal with this problem we would like the output distributions of the ciphertexts output by **eval** and **encrypt** to be identical, which is known as *circuit privacy*. This property along with how to achieve it has been described in [4].

To achieve *circuit privacy* we can make an encryption of 0 with a very large error term, and then add this ciphertext to the original ciphertext. By doing this we esentially drown out information about the error vector of the original ciphertext. This will not modify the encrypted data, as the new error term will be removed by the (mod t) computation done in **decrypt** anyways.

2.3 Protocol for Multiparty Computation based on SHE

The Multiparty Computation (MPC) problem is the problem where n players each with some private input x_i, want to compute some function f on the input, without revealing anything but the result.

In [3] the authors describe a protocol for MPC based on a SHE scheme. The protocol is able compute arithmetic formulas consisting of up to a single multiplication, along with a relatively large number of additions, while being statistically UC-secure against an active adversary and $n-1$ corruptions.

The protocol proceeds in two phases. In the first phase, preprocessing, a global key $[\![\alpha]\!]$, random values in two representations $[\![r]\!], \langle r \rangle$, and a number of multiplicative triples $\langle a \rangle, \langle b \rangle, \langle c \rangle$ satisfying $c = ab$ are generated.

In the second phase, the online phase, the players use the global key and secret-shared data generated in the preprocessing phase to do the actual computations.

The online phase therefore only makes indirect use of the SHE scheme, as it is only used in the preprocessing phase to generate input for the online phase.

These two phases are described in further detail in section 2.3.2 & 2.3.3.

Representations of shared values The protocol makes use of two different representations of shared values $[\![\cdot]\!], \langle \cdot \rangle$. For a shared value $a \in F_{p^k}$ the $\langle \cdot \rangle$ representation is defined as follows:

$$\langle a \rangle := (\delta, (a_1, ..., a_n), (\gamma(a)_1, ..., \gamma(a)_n))$$

where $a = \sum_i a_i$ and $\alpha(\delta + a) = \sum_i \gamma(a)_i$. The $\gamma(a)_i$ values are thus MAC values used to authenticate a. Such a value is shared s.t. each party P_i has access to the global value δ

along with shares $(a_i, \gamma(a)_i)$. Multiplication by a constant and regular addition and are then defined entry-wise on the representation, while addition by a constant is defined as

$$c + \langle a \rangle := (\delta - c, (a_1 + c, ..., a_n), (\gamma(a)_1, ..., \gamma(a)_n))$$

When a value $\langle a \rangle$ is partially opened, it means that the value a is revealed without revealing a's MAC values.

The second representation used for the protocol, $[\![\cdot]\!]$, is defined in the following way:

$$[\![a]\!] = ((a_1, ..., a_n), (\beta_i, \gamma(a)_1^i, ..., \gamma(a)_n^i)_{i=1,...,n})$$

where $a = \sum_i a_i$ and $a\beta_i = \sum_j \gamma(a)_i^j$. Thus the $\gamma(a)_i^j$ values are used to authenticate a under P_i's personal key β_i. Each player P_i then has the shares $(a_i, \beta_i, \gamma(a)_1^i, ..., \gamma(a)_n^i)$. To open a $[\![\cdot]\!]$ value each player P_j sends $a_j, \gamma(a)_i^j$ to P_i, who checks that $a\beta_i = \sum_j \gamma(a)_i^j$. Afterwards P_i can compute $a = \sum_i a_i$.

2.3.1 Abstract SHE scheme and instantiation

The cryptosystem used as the SHE scheme in the protocol has to have certain properties to be admissible. The authors of the MPC protocol present a concrete instantiation of such an abstract SHE scheme, using the Ring-LWE based public key encryption scheme by Zvika Brakerski and Vinod Vaikuntanathan [1] described in section x.x. To be able to use this scheme we have to define the functions $encode : (\mathbb{F}_{p^k})^s \mapsto \mathbb{Z}^N$ and $decode : \mathbb{Z}^N \mapsto (\mathbb{F}_{p^k})^s$ where $M = (\mathbb{F}_{p^k})^s$ denotes the message space.

In addition to the aforementioned requirements, we also want the cryptosystem to implement a functionality $\mathscr{F}_{KeyGenDec}$. This functionality will on receiving "start" from all honest players generate a keypair (pk, sk), and then distribute pk to the players and store sk. The players can then use the functionality to cooperate in decrypting a ciphertext encrypted under pk.

2.3.2 Preprocessing phase

The preprocessing phase is implemented by the Prep protocol, which consists of the steps **initialize**, **pair**, and **triple**. These steps use the additional protocols Reshare, PAngle, and PBracket as subroutines.

Protocol Reshare: The Reshare protocol takes a ciphertext e_m as input and a parameter enc, which can be set to either $NewCiphertext$ or $NoNewCiphertext$. The protocol then outputs a share m_i of m to each player along with a new fresh ciphertext e'_m if $enc = NewCiphertext$, where e'_m contains $\sum_i m_i$. To do this the players first each sample $f_i \in \mathbb{F}_{p^k}$, and then broadcast

$$e_{f_i} \leftarrow Enc_{pk}(f_i)$$

Each P_i then runs the ZKPoPK protocol while acting as a prover on the previously generated ciphertext e_{f_i}, and if any of these proofs fail, then parties abort. Now, each player homomorphically adds each encrypted share

$$e_f \leftarrow e_{f_1} \boxplus ... \boxplus e_{f_n}$$

and then homomorphically adds e_m and e_f to get e_{m+f}. The players now use $\mathscr{F}_{KeyGenDec}$ to decrypt e_{m+f} so that they get $m+f$. Now P_1 sets $m_1 \leftarrow m+f-f_1$, while the rest of the players P_i set $m_i \leftarrow -f_i$. If $enc = NewCiphertext$, then the players each compute

$$e'_m \leftarrow Enc_{pk}(m+f) \boxminus e_{f_1} \boxminus ... \boxminus e_{f_n}$$

where default randomness is used for the encryption.

Protocol PAngle: PAngle takes as input a ciphertext e_v along with privately held shares $v_1,...,v_n$. These are then used to generate a value in the angle representation $\langle v \rangle$. To achieve this all players first compute

$$e_{v \cdot a} \leftarrow e_v \boxplus e_\alpha$$

Reshare is then used with $e_{v \cdot \alpha}$ as input, such that each player P_i recieves a share γ_i of $v \cdot \alpha$. Finally, $\langle v \rangle = (0,(v_1,...,v_n),(\gamma_1,...,\gamma_n)$ is output.

Protocol PBracket: PAngle takes as input a ciphertext e_v along with privately held shares $v_1,...,v_n$. These are then used to generate a value in the angle representation $[\![v]\!]$. For $i = 1,...,n$ all players compute

$$e_{\gamma_i} \leftarrow e_{\beta_i} \boxtimes e_v$$

and then generate $(\gamma_i^1,...,\gamma_i^n)$ by calling Reshare with e_{γ_i} and $NoNewCiphertext$ as input. The representation $[\![v]\!] = ((v_1,...,v_n),(\beta_i,\gamma(v)_1^i,...,\gamma(v)_n^i)_{i=1,...,n})$ is then output.

Initialize: The **initialize** step generates the global and personal keys. This is acheived by the players first calling "start" on $\mathscr{F}_{KeyGenDec}$, so that every player obtains the public key pk. Then each player samples $\alpha_i, \beta_i \in \mathbb{F}_{p^k}$, and broadcasts

$$e_{\alpha_i} \leftarrow Enc_{pk}(Diag(\alpha_i)), \;\; e_{\beta_i} \leftarrow Enc_{pk}(Diag(\beta_i))$$

where $Diag(a) = (a,a,...,a) \in (\mathbb{F}_{p^k})^s$. Now each P_i runs the ZKPoPK protocol twice with $diag = true$, while acting as a prover, where the inputs are the ciphertexts e_{α_i} and e_{β_i} repeated sec times. Finally, the players homomorphically add the encrypted shares e_{α_i} to get e_α, which they use along with their share $Diag(\alpha_i)$ to generate $[\![Diag(\alpha)]\!]$ using a call to PBracket. Then $[\![Diag(\alpha)]\!]$ is the global key, while β_i is P_i's personal key.

Pair: In **pair** the players generate random values in the two representations $[\![r]\!], \langle r \rangle$. This is done by each player first sampling a share $r_i \in (\mathbb{F}_{p^k})^s$, then broadcasting Each player then encrypts their share to get

$$e_{r_i} \leftarrow Enc_{pk}(r_i)$$

which they then broadcast. Once again P_i will now runs the ZKPoPK protocol acting as a prover with input e_{r_i}, and if the ZK proof fails, then the protocol is aborted. The players then homomorphically add the encrytped shares to get e_r, which they use along with their share r_i as input to PBracket and PAngle to generate $[\![r]\!], \langle r \rangle$.

Triple: The **triple** step generates triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ satisfying $c = ab$. To do this the players start off by sampling shares $a_i, b_i \in (\mathbb{F}_{p^k})^s$. The players then encrypt their shares and broadcast the result

$$e_{a_i} \leftarrow Enc_{pk}(a_i), \;\; e_{b_i} \leftarrow Enc_{pk}(b_i)$$

Now each P_i acts as a prover running the ZKPoPK protocol first with e_{a_i} and then with e_{b_i} as input, and if any proof fails, then the protocol is aborted. The players then homomorphically add the encrypted shares to get e_a and e_b, and use these along with their shares a_i, b_i to generate $\langle a\rangle$, $\langle b\rangle$ using calls to PAngle. Following this each player homomorphically multiplies e_a and e_b to get e_c, which the players use as input to Reshare to get shares of c along with a new ciphertext $(c_1, ..., c_n, e_{c'})$. Then the players then use their shares of c along with $e_{c'}$ to generate $\langle c\rangle$ by calling PAngle. Finally, the triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ is output.

2.3.3 Online phase

The Online protocol implements the online phase, and consists of the steps **initialize**, **input**, **add**, **multiply**, and **output**. These steps are executed as needed to evaluate the arithemtic circuit that we wish to evaluate.

Initialize: The **initialize** step simply consists using the Prep protocol to generate a global key, along with enough multiplicative triples and random values in the two representations showed earlier, for the circuit that we want to evaluate.

Input: The *input* step lets a player P_i share their private input x_i. The input is shared by taking a pair $[\![r]\!], \langle r\rangle$, and then opening $[\![r]\!]$ to P_i so that P_i gets r. Following this P_i computes and broadcasts $\varepsilon \leftarrow x_i - r$. All players finally set $\langle x_i\rangle \leftarrow \langle r\rangle + \varepsilon$.

Add: To add two values $\langle x\rangle, \langle y\rangle$, we simply perform the component-wise addition $\langle z\rangle = \langle x\rangle + \langle y\rangle$, meaning that each player adds their shares locally $z_i = x_i + y_i, \gamma(z)_i = \gamma(x)_i + \gamma(y)_i$.

Multiply: To multiply two values $\langle x\rangle, \langle y\rangle$, we use two multiplicative triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$. We use the second triple to check that $ab = c$, but this could also be done in preprocessing instead. To do this check we first open $[\![t]\!]$ to get t, then partially open $t * \langle a\rangle - \langle f\rangle$ and $\langle b\rangle - \langle g\rangle$ to get ρ and σ respectively. Finally, we compute and partially open

$$t * \langle c\rangle - \langle h\rangle - \sigma\langle f\rangle - \rho\langle g\rangle - \sigma\rho$$

if the result is 0, then we conclude that $ab = c$ and move on, otherwise the players abort.

Now, partially open $\langle x\rangle - \langle a\rangle$ and $\langle y\rangle - \langle b\rangle$ to get ε and δ respectively. Finally, we ouptut

$$\langle z\rangle \leftarrow \langle c\rangle + \varepsilon\langle b\rangle + \delta\langle a\rangle + \varepsilon\delta$$

Output: To output a value y given $\langle y \rangle$, the players do the following. First, a random value $[\![e]\!]$ is opened so each player gets e, which is used to compute

$$a = \sum_j e^j \cdot a_j$$

where a_j are all of the opened values of the form $\langle a_j \rangle$. Each player P_i then uses the commitment functionality $\mathscr{F}_{Com}$ to commit to $\gamma_i \leftarrow \sum_j e^j \gamma(a_j)_i$ along with y_i and $\gamma(y)_i$. Then, the global key $[\![\alpha]\!]$ is opened. The players then use $\mathscr{F}_{Com}$ to open γ_i and then check that indeed

$$\alpha(a + \sum_j e^j \delta_j) = \sum_j \gamma_i$$

If this is not the case, then the players abort. Finally, to make sure that all players end up with y the commitments to y_i and $\gamma(y)_i$ are opened. Now the players check that $\alpha(y+\delta) = \sum_i \gamma(y)_i$, and if this is the case, then $y = \sum_i y_i$ is output.

2.4 Reuse of unrevealed secret-shared data

In [2] a technique that allows for reuse of unrevealed secret-shared data is used. This technique revolves around not having to open $[\![\alpha]\!]$, and in fact we do not even need the $[\![\cdot]\!]$ representation when using this technique.

The technique works as follows. First, when we generate the global key we now just need each player P_i to have a share α_i of α. In the output step the player will then instead of the current MAC check instead invoke the new MACCheck protocol on all of the values in the $\langle \cdot \rangle$ representation that have been opened, and if this succeeds the player invokes MACCheck on the output value $\langle y \rangle$. The MACCheck protocol works as follows:

Protocol MACCheck First each P_i samples a seed s_i and use $\mathscr{F}_{Commit}$ to broadcast $\tau_i^s \leftarrow Commit(s_i)$. Following this each player opens all commitments using $\mathscr{F}_{Commit}$ to get all n seeds s_j. Now, all players set

$$s \leftarrow s_1 \oplus \ldots \oplus s_n$$

Players then use s as seed to sample a random vector of length t with entries in the interval $[0, p)$. All players then first compute

$$a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$$

where the a_j's are the opened values. Now P_i computes

$$\gamma_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i \text{ and } \sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$$

Player i then uses $\mathscr{F}_{Commit}$ to broadcast $\tau_i^\sigma \leftarrow Commit(s_i)$. All players invoke $\mathscr{F}_{Commit}$ to open the commitments received to get the σ_j's. Finally, the players check that $\sigma_1 + \ldots + \sigma_n = 0$, and if this is not the case, then they abort.

Chapter 3

Implementation

For the implementation part of the project we first implemented the Ring-LWE cryptosystem described in section 2.1. We then used this for implementing the MPC protocol described in section 2.3, while making slight deviations to allow for reuse of secret-shared values, as described in section 2.4. The code for the protocol was written in the programming language Rust, and can be found in the repository at `https://github.com/Gabaa/homomorphic-encryption-project`. In this chapter we describe the implementation details of these systems.

3.1 Ring-LWE cryptosystem

3.1.1 Polynomials (`poly.rs`)

Since our public-key encryption scheme uses polynomials to represent most of the values (messages, ciphertexts, secret keys, and public keys), we implemented a simple `Polynomial` data structure, along with the most common operations we will perform on it.

Internally, a `Polynomial` is simply a `Vec` (a contiguous growable array type) of `Integer` values, each representing a coefficient in the polynomial.

We implemented operations for adding, subtracting, negating, and multiplying (with both constants and other polynomials), and functions for trimming the polynomial (removing trailing zero-coefficients), right-shifting the coefficients (from lower to higher degrees), retrieving the ℓ_∞, calculating the modulo, and normalizing the coefficients to be in the range $[-q/2, q/2)$ instead of $[0, q)$, which is necessary during decryption.

3.1.2 Quotient ring (`quotient_ring.rs`)

Encryption, decryption, and key generation involves adding, subtracting, multiplying, and negating elements in the quotient ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. To be able to do these computations we made a quotient ring implementation, which can be found in `quotient_ring.rs`.

`Rq`

The quotient ring module contains a struct definition `Rq`, which represents an instantiation of a quotient ring $Z_q[x]/\langle f(x)\rangle$. It therefore has fields q and *modulo*, where q is a `BigInt`, and *modulo* is a `Polynomial` representing $f(x)$. The **`new`** function takes q and *modulo* as input, and is used to make a new instantiation of `Rq`.

`reduce`

The `reduce` method found in the quotient ring module is called on an `Rq` struct, takes a polynomial `pol` as input, and returns the normal form of the element `pol` with respect to `modulo`.

To achieve this the method first does polynomial long division with `pol` as the dividend and the `modulo` from the `Rq` struct as the divisor. The remainder computed in this way is then stored in the variable r.

Lastly, we reduce the coefficients of the resulting polynomial r modulo q, by using the remainder operation(%) defined in the `poly.rs` module, and then return the result.

`add`, `times`, `neg`, `mul`

The methods `add`, `times`, `neg`, `mul` are called on an `Rq` struct. These methods first use the addition, scalar multiplication, negation, and polynomial multiplication methods defined in the the `poly.rs` module on the input. Then a reduce call is done with the result as input to get a new R_q element.

3.1.3 Public-key encryption scheme (`encryption.rs`)

The encryption scheme, as usual, has three major components:

- the `generate_key_pair` function
- the `encrypt` function
- and the `decrypt` function

`generate_key_pair`

The `generate_key_pair` function takes as input an instance of the `Parameters` struct. This struct essentially just defines the parameters that nearly all functions in the encryption scheme use in some form or another. This includes r, n, q, t, and the quotient ring R_q, which are all relevant for the key generation function.

The function starts by sampling polynomials sk and e_0 from a Gaussian distribution with standard deviation r, and the polynomial a_0 uniformly from $\mathbb{Z}_q$.

It then calculates the public-key as $pk = (a_0, a_0 \cdot sk + e_0 \cdot t)$, and finally returns the key pair (pk, sk).

encrypt

The `encrypt` function takes a `Parameters` instance, as described above, and additionally takes a polynomial m and a public key pk.

We extract the two polynomials of the public key, a_0 and b_0.

First, we make sure that the message polynomial we are trying to encrypt is in R_t. Then, we sample polynomials v and e' from a Gaussian distribution with standard deviation r, and the polynomial e'' from a Gaussian distribution with standard deviation r' (which is also defined in the `Parameters` struct).

We then calculate $a = a_0 \cdot v + e' \cdot t$ and $b = b_0 \cdot v + e'' \cdot t$. Finally, we create the ciphertext as a `Vec` $c = [b+m, a]$ and return it.

decrypt

The `decrypt` function takes a `Parameters` instance, as well as a ciphertext $c = [c_0, c_1, \ldots]$ and a secret key sk.

We start by constructing the secret key vector $\mathbf{s} = [1, s, s^2, \ldots]$ from the secret key. We only create the first $|c|$ entries of the secret key vector, as those are the only ones we'll need.

We then initialize a polynomial $msg = 0$, which will become the decrypted message. Then, iterating over each element c_i in the ciphertext, we add $c_i \cdot \mathbf{s}_i$ to msg, where $\mathbf{s}_i$ is the i'th entry (zero-indexed) in the secret key vector.

Currently, the message has coefficients in $\mathbb{Z}_q$, but we need them to be in the range $-\frac{q}{2}$ to $\frac{q}{2}$. Therefore, we iterate through all coefficients x and map them to the new range such that if $x > \frac{q}{2}$, then we let $x' = x - q$, and otherwise $x' = x$.

Finally, we reduce the message modulo t to remove the $e \cdot t$ part of the encryption, and return the result.

Skal nok lige have nogle andre til at tilpasse denne - jeg har svært ved at finde ud af hvordan man forklarer det.

Er det inklusive eller eksklusive i intervallet?

3.2 MPC protocol

We now describe our implementation of an actively secure MPC protocol.

For this protocol we did not implement the cyclotomic polynomial optimisation described earlier, so $s = 1$, meaning that we are working in $\mathbb{F}_{p^k}$. This also means that we cannot do entry-wise multiplications like done in [3]. Therefore, to encode an element $a \in \mathbb{F}_{p^k}$ to an element in R_q, we simply return `Polynomial::new(vec![a])`, which is the `Polynomial` corresponding to $f(x) = a$. To decode a polynomial $f(x) = a \in R_q$, we simply return the `Integer` a.

3.2.1 ZKPoPK (zk.rs)

In zk.rs the zero knowledge proof of plaintext is implemented for each party to run. The zero-knowledge proof has been implemented as in figure 10 of "multiparty computation from somewhat homomorphic encryption" [3]. The file is divided into 2 different functions. To generate a zero knowledge proof, we must call the function make_zkopk() which takes 6 arguments

- **params** The parameters of the PLWE instantiation.

- **x**: The plaintext for which we intend to generate the proof.
- **r**: The randomness to be used.
- **c**: The ciphertexts generated in the preprocessing protocol.
- **diag**: A boolean to indicate whether the diagonal element should be checked.
- **Pk**: The public key to be used.

make_zkopk() will generate 3 values (a, z, T), which will be used in the verification of the proof. To verify the proof we call the function verify_zkopk, which takes as input 6 arguments as well

- **a**: value generated by the make_zkopk function.
- **z**: value generated by the make_zkopk function.
- **t**: value generated by the make_zkopk function.
- **c**: The ciphertexts generated in the preprocessing protocol.
- **params** The parameters of the PLWE instantiation.
- **Pk**: The public key to be used.

verify_zkopk will return a boolean indicating whether or not the proof was valid.

3.2.2 Preprocessing phase (prep.rs)

The file prep.rs contains all of the functions that are called in the preprocessing phase to generate values for the online phase.

The functions `reshare` and `p_angle` correspond to the Reshare and PAngle protocols explained previously, while `initialize`, `pair`, and `triple` correspond to three steps of the Prep protocol.

The file also contains a type definition `AngleShare`, which for some value $\langle v \rangle$ is a pair of the form $(v_i, \gamma(v)_i)$ for some player i. Additionally, the file also contains a definition `MulTriple`, which contains three `AngleShare` values, and represent a players shares of the $\langle \cdot \rangle$ values in a multiplicative triple.

reshare

The first function, `reshare`, takes as input a `Parameters` struct `params`, a Ciphertext `e_m`, a `PlayerState state`, and an `Enc` enum `enc`, which can take on values `NewCiphertext` or `NoNewCiphertext`.

The first thing done in this function is to sample a value `f_i` uniformly from Z_t using `sample_single` from `prob.rs`. Then, `f_i` is encrypted to get `e_f_i`. The facilitator is then used to broadcast `e_f_i`, and subsequently receive the encrypted shares from the other parties, which are then homomorphically added to get `e_f`. Then, we compute `e_m_plus_f = add(params, e_m, &e_f)`.

The `ddec` function from mod.rs is then called with `e_m_plus_f` as input to get the plaintext `m_plus_f`.

Then we set `m_i` to be $e_{m+f} - f_i \bmod t$ if the player calling the function is the player with index 0, and $-f_i \bmod t$ otherwise. If `enc = NewCiphertext`, we now encrypt `m_plus_f` using `encrypt_det` where we use a triple of 1-polynomials instead of the randomness to make encryption deterministic. Now, we use `add` from encrypt.rs to homomorphically subtract the encrypted shares e_{f_i} from the encryption of `m_plus_f` to get `e_m_prime`. Afterwards, we return `(Some(e_m_prime), m_i)`

If `enc = NoNewCiphertext`, we instead just return `(None, m_i)`.

p_angle

The `p_angle` function takes a `Parameters` struct `params`, an `Integer` `v_i`, a `Ciphertext` `e_v`, and a `PlayerState` `state` as input.

The first thing done in `p_angle` is to homomorphically multiply `e_v` and `state.e_alpha` to get `e_v_mul_alpha`. Then, `reshare` is called to get `gamma_i`, which is a share of an `Integer`, namely the plaintext in `e_v_mul_alpha`. Lastly, the function returns `(v_i, gamma_i)`, which is an `AngleShare` value corresponding to a share of $\langle v \rangle$.

As can be seen from the implementation we omit the public δ value as done in [2], such that the MAC's now instead satisfy $\alpha v = \sum_i \gamma(v)_i$.

initialize

The initialize function takes a `Parameters` struct `params`, and a mutable `PlayerState` `state` as input.

First, we sample a uniformly random `Integer` from $[0, t)$ using the `sample_single` function with `params.t` as input, and set the `alpha_i` variable of the player state to this value. This represents the given players share of the global key. We then encrypt `state.alpha_i` to get an encrypted share `e_alpha_i`. Now, `e_alpha_i` is broadcast using `state.facilitor`, and each player then uses their facilitator to receive e_{α_i} from the other players. The e_{α_i}'s are then homomorphically added to get e_α, and the result is then assigned to the `e_alpha` variable of `state`. Finally, we run `zpopk` from `zk.rs` in a loop *sec* times.

Notice how we do not compute the personal keys β_i as done in [3]. As mentioned earlier these are not needed when we use the trick described in section 2.4.

pair

The `pair` function takes a Parameters struct params, and a PlayerState state as input.

The function first samples a uniformly random `Integer` `r_i` from Z_t. Now, `r_i` is encrypted to get `e_r_i`, which is the broadcast using the facilitator. All players then again use the facilitator to receive the encrypted shares from the other parties, which are then homomorphically added to compute `e_r`. Then, we call `zkpopk` with `e_r_i` as input.

Following this we compute the given players share of $\langle r \rangle$ with a call to `p_angle` with `r_i` and `e_r` as arguments, which returns `r_angle`.

Again, we use the trick explained in section 2.4, so we don't need the values in the bracket representation, and therefore we just return `(r_i, r_angle)`, which corresponds to shares of the pair $(r, \langle r \rangle)$.

triple

The triple function takes a `Parameters` struct `params`, along with a `PlayerState` `state` as arguments.

First, we sample `a_i`, `b_i` uniformly at random from Z_t, which are both then encrypted, and the encrypted shares are then broadcast.

Then, when the player receives the encrypted shares of a and b from the other players, then these are homomorphically added to get `e_a` and `e_b`.

Following this, we generate shares `a_angle` and `b_angle` with calls to `p_angle` using `a_i`, `e_a` and `b_i`, `e_b` as input respectively.

Now, the player calling the function has shares of $\langle a \rangle$ and $\langle b \rangle$, and we need to compute a share of $\langle c \rangle$.

To do this we compute `e_c` by homomorphically multiplying `e_a` and `e_b`, and then we call `reshare` with `e_c` and `NewCiphertext` as input to get a new ciphertext `e_c_prime` and `c_i`, which is a share of `c`.

This allows us to call `p_angle` using `c_i` and `e_c_prime` to get `c_angle`.

Finally, we return a `MulTriple` containing `a_angle`, `b_angle`, and `c_angle`.

3.2.3 Commitments (commit.rs)

For the online phase we also need a commitment functionality as described in [2]. Our implementation of this functionality can be found in commit.rs. This file contains the methods `commit` and `open`, which are explained in greater detail below.

commit

The commit function simply takes two `Vec<u8>` values called `v` and `r`, along with a `PlayerState` `state` as input. Then `v` is the value that we wish to commit to, while `r` is the randomness that we use when committing.

In this method we utilize the implementation of sha256 found in the `sha2` package to hash the concatenation of `v` and `r`, which yields the commitment `c`.

`c` is then broadcast to all players using the facilitator.

open

This function takes a commitment `c` and a value `o` as input, and these are both `Vec<u8>`. The `o` value is then supposed to satisfy that $o = v||r$.

When calling this function it hashes `o` using sha256, and then checks whether `h(o) = c`. If this is indeed the case, then we return `Ok(o)`, and otherwise we return an error.

3.2.4 Online phase (prep.rs)

The code related to the online phase can be found in prep.rs. The implementation of this phase is based on the online protocol in [2], but in that protocol the triple check is done in preprocessing, while we do it in the online phase as in [3].

The two functions `give_input` and `receive_input` together correspond to the Input step of the protocol. The `add`, `multiply`, and `output` functions correspond to the steps of the same names. The function `maccheck` corresponds to the MACCheck protocol, and `triple_check` is used as a subroutine in `multiply` to check for validity of a multiplicative triple.

give_input

This function takes a `Parameters` struct `params`, an `Integer` `x_i`, a `(Integer, AngleShare)` pair called `r_pair`, and a `PlayerState` `state`.

First, the function broadcasts a message `BeginInput` to indicate that the player calling the function wants to give some input.

Then, the player receives all shares of r from the other players using the facilitator, and opens r by computing $r = r_1 + \ldots + r_n \mod p$.

The value `eps = x_i - r` is then computed, and subsequently broadcast to all players.

The `AngleShare` corresponding to $\langle r \rangle + \varepsilon$ is then computed and returned.

receive_input

The `receive_input` function takes a `Parameters` struct `params`, an `Integer` `x_i`, a `(Integer, AngleShare)` pair called `r_pair`, and a `PlayerState` `state`.

The first thing that the function does is to receive a message using the facilitator, and if this is not `BeginInput`, then we panic.

Following this we take r_i from `r_pair`, and send it to the player `p_i`, which is the player that is providing input, and thus also the player that sent the initial `BeginInput` message.

Now, we receive ε from `p_i`, and then use this to compute and return the `AngleShare` corresponding to $\langle x_i \rangle = \langle r \rangle + \varepsilon$ for the given player.

add

Add simply takes two `AngleShare` values `x` and `y` as input.

These are then used to compute `(x.0 + y.0, x.1 + y.1)`, and the resulting `AngleShare` is then returned.

partial_opening

To partially open some value v where player i holds share v_i, each player calls `partial_opening` with a `Parameters` struct `params`, an `Integer` `to_share`, and a `PlayerState` `state` as arguments. When partially opening we send all shares to some designated player, and in this case we simply let all players send their share to player P_1, who has index 0.

To do this we first send the calling players share of v, `to_share`, to P_1 using the facilitator. Player P_1 then computes $v = v_1 + ... + v_n \mod p$, which is subsequently broadcast to all players using the facilitator.

Finally, the function returns the value v received from P_1.

triple_check

The `triple_check` function takes a `Parameters` struct `params`, two `MulTriple` values `abc_triple` and `fgh_triple`, an `Integer` `t_share`, and a mutable `PlayerState` `state` as arguments.

The first thing done is to use the facilitator to broadcast the players share of t, `t_share`. The player then uses the facilitator to receive shares of t from the other players, and then compute $t = t_1 + ... + t_n \mod p$.

Now, we compute $t \cdot \langle a \rangle - \langle f \rangle$ and save the resulting `AngleShare` in variable `rho_share`. Following this we call `partial_opening` with `rho_share.0` as input to get ρ. Lastly, we push `(rho, rho_share.1)` to `state.opened`, to ensure that we check the MAC of ρ in `maccheck`.

Now, we use the same approach as for ρ to compute $\langle b \rangle - \langle g \rangle$ and store the resulting `AngleShare` in variable `sigma_share`. Then we call `partial_opening` with `sigma_share.0` as input to get σ. Then we push `(sigma, sigma_share.1)` to `state.opened`.

We then compute $t \cdot \langle c \rangle - \langle h \rangle - \sigma \cdot \langle f \rangle - \rho \cdot \langle g \rangle - \sigma \cdot \rho$, call `partial_opening` with the resulting `AngleShare` as input to get the variable `zero`.

The last thing done is to check whether `zero` is 0. If this is the case, then we return `Ok`, otherwise we return `Err`.

multiply

The `multiply` function takes a `Parameters` struct `params`, two `AngleShare` values `x` and `y`, two `MulTriple` values `abc_triple` and `fgh_triple`, an `Integer` `t_share`, and a `PlayerState` `state` as arguments.

First, we call `triple_check` with `params`, `abc_triple`, `fgh_triple`, `t_share`, and `state` as input. If the call to `triple_check` returns `Err`, then we panic and abort the protocol, otherwise we proceed.

Following this we compute the AngleShare corresponding to $\langle x \rangle - \langle a \rangle$, so that we get `eps_share`. We then call `partial_opening` with `eps_share.0` as input to get `eps`. Then we push `(eps, eps_share.1)` to `state.opened`.

Now we compute the AngleShare corresponding to $\langle y \rangle - \langle b \rangle$, so we get `delta_share`, then call `partial_opening` with `delta_share.0` as input to get `delta`. And now we push `(delta, delta_share.1)` to `state.opened`.

Finally, we compute the `AngleShare` corresponding to $\langle z \rangle = \langle c \rangle + \varepsilon \langle b \rangle \delta \langle a \rangle + \varepsilon \delta$ and return it.

maccheck

The `maccheck` function takes a `Parameters` struct `params`, a `Vec<(Integer, Integer)>` of $(a_j, \gamma(a_j)_i)$ pairs named `to_check`, and a `PlayerState` `state` as arguments.

First, we use the `rand` package to sample random 32-byte values `s_i` and `r`.

Then we use the `commit` function from `commit.rs`, to commit to `s_i` using `r` as randomness.

Following this we use the facilitator to receive the commitments from all n parties. Then once all of the commitments have been received the parties broadcast the value $o = s_i||r$, such that all parties then open the commitments that they have received and get the seed s_i for all players i.

Now, we XOR the received seeds to get $s = s_1 \oplus ... \oplus s_n$. This is followed by using the `rand` package and the seed s to sample a vector `rng_seed` with n entries and with values in the range $[0, p)$.

Then the function uses the values $a_1, ..., a_t$ stored in `state.opened` to compute $a = \sum_{j=1}^{t} r_j a_j$.

After generating a we use it to compute $\gamma_i = \sum_{j=1}^{t} r_j \gamma(a_j)_i$ and $\sigma_i = \gamma_i - \alpha_i a$. Then we convert σ_i to bytes and store the result in `sigma_i_bytes`.

Afterwards we need to commit to `sigma_i_bytes`, and to do this we sample 32 bytes of randomness `r` as done earlier. This is followed by committing to `sigma_i_bytes` using `r` as randomness.

Now we use the facilitator to receive commitments from all n players, and once these have been received we broadcast $o = sigma\_i\_bytes||r$.

Then we use the o values received to open the σ commitments received earlier, such that we get bytes corresponding to σ_i from all players i. Then, we convert the bytes into values of the `Integer` type.

Finally, we add the n σ_i values received and check that indeed the sum is equal to `Integer::ZERO`. If this is the case, then we return true, otherwise we return false.

output

The `output` function is called by a player when that player is ready to output some value $\langle y \rangle$. The function takes a `Parameters` struct `params`, `AngleShare y_angle`, and a `PlayerState state` as arguments.

To output a value we first call the `maccheck` function on `state.opened`, to check that the MAC values are correct for all of the values v in the $\langle r \rangle$ representation that have been opened. If this returns false, then the check was unsuccessful and we `panic` to abort the protocol.

Then we broadcast `y_angle.0`, which is the players share y_i of the output value y. This is followed by receiving shares from all players, and then computing $y = y_1 + ... + y_n \mod p$.

Now, we once again call the `maccheck` function but this time we use the `(y, y_angle.1)` as input, which is the opened value y and the players corresponding MAC value. If this returns false, then we once again `panic` to abort the protocol.

If we did not abort at any point during the abort, then we return the final result `y`.

Chapter 4

Conclusion

...

Acknowledgments

...

Bibliography

[1] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[2] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[3] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[4] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

Appendix A

First appendix