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

Secure multiparty computation is an important problem, where multiple parties can provide private inputs to securely evaluate a function and receive the output, without revealing any other information in the process. One way of implementing such a system is with homomorphic encryption, which allows operations to be performed on encrypted data.

For this work, we implement a multiparty computation system based on somewhat homomorphism in the Rust programming language. Our somewhat homomorphic encryption scheme is based on the polynomial learning with errors problem, which is commonly used for post-quantum secure encryption schemes. The system is able to compute arithmetic formulas consisting of up to a single multiplication, along with a relatively large number of additions. Supporting new arithmetic circuits is fairly simple by extending the code.

We found that in order for the system to be secure, the parameters must be prohibitively large relative to the performance of the system, making our implementation either insecure or impractical. Specifically, running the system with secure parameters for a simple arithmetic circuit would require around 6.5 days to compute, in large part due to an inefficient preprocessing phase.

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

Contents

Chapter 1

Introduction

By the work of Peter W. Shor, we know of an algorithm that is expected to break the prime factorization and discrete logarithm problems in polynomial time, assuming that it can be run on a large enough quantum computer [9]. While quantum computers today are not large enough, the field is increasing rapidly and as such we should strive for new solutions.

One solution is to change the underlying mathemathical problem to no longer use prime factorization or discrete log, and instead use encryption schemes based on polynomial learning with errors as proposed by [7]. This problem can be reduced to worst case problems on ideal lattices, making it a good candidate for future encryption schemes, as can also be seen by the submissions to the post-quantum NIST competition, since many of the submissions are based on lattices.

Another highly desired property in cryptography is that of secure multiparty computation (MPC). In this problem, n players wish to compute some function such that only the output of the function is revealed to every player, even in the case of an adversary controlling some of the corrupted players. It has been shown, for a limited number of operations, by Brakerski and Vaikuntanathan [3] that it is possible to get a homomorphic encryption scheme based on the polynomial learning with errors problem.

While encryption based on lattices seems very promising, parameters can still be chosen such that the underlying problem does not impose the desired level of security. To prevent this, the LWE estimator tool like [1], maintained by Martin Albrecht, can be used to argue about the security of different parameter sets.

During this project we have implemented a system, mostly inspired by the work of Brakerski and Vaikuntanathan (BV) [3] and Ivan et al. [5]. Our system is made in the Rust programming language, which provides absolute memory safety, making it an ideal choice for doing low-level efficient implementations, while still having many of the nice features of a modern language. While performance has not been our main concern, we have tried to optimize the code written when possible, using and evaluating many of the features in the Rust language to have the optimal code for the features implemented.

Chapter 2

Review of literature

A homomorphic encryption scheme is an encryption scheme that allows computing functions on encrypted data. Such a scheme was first presented by Gentry in [6]. This scheme is built on what is known as a "somewhat" homomorphic encryption scheme (SHE), which allow the evaluation of a limited set of functions on encrypted data [3].

In this literature review we first present a concrete SHE scheme. We then present a Multiparty Computation (MPC) protocol based on this SHE scheme.

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 Brakerski and Vaikuntanathan [3], the authors describe a method for using the ring learning with errors (Ring-LWE) problem to construct an encryption scheme, which reduces to the worst-case hardness of problems on ideal lattices [3]. We will shortly describe the encryption scheme here, but will omit proofs and detailed discussions.

2.1.1 Polynomial learning with errors

Polynomial learning with errors (PLWE) is a variant of the Ring-LWE problem. The decisional PLWE problem is parameterized by a polynomial $f(x) \in \mathbb{Z}[x]$ where $deg(f) = N$, a prime $q \in \mathbb{Z}$, a distribution χ (specifically an error distribution over elements in R_q), and an integer ℓ (a limit on the number of samples given in the problem). f defines the ring $R = \mathbb{Z}[x]/\langle f(x)\rangle$, and R and q together define $R_q = R/qR = \mathbb{Z}_q[x]/\langle f(x)\rangle$. An instance of the problem is then written as $PLWE^{(\ell)}_{f,q,\chi}$.

Definition (The $PLWE^{(\ell)}_{f,q,\chi}$ assumption)**.** Let s be a uniformly random element from R_q. Then it holds that

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

where all a_i and u_i are uniformly random elements from R_q, and all e_i are sampled from χ.

Likewise, the decisional $PLWE^{(\ell)}_{f,q,\chi}$ problem is then to distinguish how a set of ℓ samples was generated: as elements of the form $(a_i, a_i \cdot s + e_i)$, or of the form (a_i, u_i).

Beside the decision variant, the PLWE problem could also be stated in terms of a search problem, one in which an adversary has to find the secret vector $s \in R_q$.

2.1.2 Symmetric encryption scheme

Now we summarise a symmetric encryption scheme based on PLWE as defined in [3].

Let κ be the security parameter and let further p and q be prime numbers where $p \in \mathbb{Z}_N^*$. We also need a polynomial of degree N: $f(x) \in \mathbb{Z}[x]$ and an error distribution χ over the ring $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$. The message space is then $R_p = \mathbb{Z}_p[x]/\langle f(x) \rangle$. Then we can define the following operations for a somewhat-homomorphic symmetric encryption scheme.

Key generation

Let our secret key be a randomly sampled element from the error distribution $s \xleftarrow{\$} \chi$. Now, for the purpose of decryption we define the secret key vector by $(s^0, s^1, s^2, \ldots, s^D) \in R_q^{D+1}$, where D is related to the maximal degree of homomorphism allowed.

Encryption

All messages are encodeable in our message space R_p, thus we will encode our message m as a degree N polynomial with coefficients modulo p. To encrypt, we sample $(a, b = a \cdot s + p \cdot e)$ where $a \xleftarrow{\$} R_q$ and $e \xleftarrow{\$} \chi$, then compute

$$c_0 \coloneqq b + m \qquad\qquad c_1 \coloneqq -a$$

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

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 p.

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

m will then be the decrypted message.

Note that for decryption to work correctly, we require that the ℓ_∞ norm (defined as the largest absolute coefficient in the polynomial) for the polynomial $\langle c, s \rangle$ is less than $q/2$.

Eval

To obtain the homomorphic abilities of the encryption scheme, Brakerski and 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 symbolic value. Then calculate the updated ciphertext $(\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{d+d'}) \in R_q^{D+D'+1}$ as

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

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 encryption scheme

To achieve a public-key encryption scheme instead, we introduce the error distribution χ'. Let $\chi = D_{\mathbb{Z}^N, r}$ and $\chi' = D_{\mathbb{Z}^N, r'}$ be the discrete gaussian distributions with standard deviation r and r' respectively. Then, we can make the following changes:

- In the key generation we generate in addition to the secret key $sk = s \xleftarrow{\$} \chi$, a public key $pk = (a_0, b_0 = a_0 \cdot s + p \cdot e_0)$, where $a_0 \xleftarrow{\$} R_q, e_0 \xleftarrow{\$} \chi$.
- In the encryption algorithm, we use a public key to calculate the ciphertext as $(a_0 \cdot v + t \cdot e', b_0 \cdot v + p \cdot e'')$ where $v, e' \xleftarrow{\$} \chi$ and $e'' \xleftarrow{\$} \chi'$.

2.1.4 On the security of PLWE

To later generate a secure set of parameters for our encryption scheme, we will have to touch a bit on the security. Several considerations come to play when thinking about the security and optimizations, most noticeably the choice of a proper function $f(x)$. The best well-known attack should also come into consideration. A lot of math, including calculations on lattices and especially ideal lattices, are needed for a proper discussion of the choices to be made in choosing secure parameters, but we will only briefly outline those processes, as it is not the main focus of this project.

On the choice of functions $f(x)$

The choice of function $f(x)$ is normally set to be a cyclotomic polynomial. Doing so, we get some useful algebraic properties, which opens for optimizations and adds to the security, as described in section 2.3.1. One way to choose a cyclotomic polynomial would be to set $f(x) = x^N + 1$ where $N = 2^k$ for $k \in \mathbb{N}$.

Reducing RLWE to LWE samples

Another important property to see is that any RLWE sample of the form $(a, s \cdot a + e)$, where $a \in R_q$ and $s, e \xleftarrow{\$} \chi$ can be written into N LWE samples by using the following method. Let A_a be the matrix of multiplication by $a \in R_q$, then we get N LWE samples by

$$(A_a, \mathbf{b} = \mathbf{s}^T \cdot A_a + \mathbf{e}^T)$$

where $\mathbf{b} \in (\mathbb{R}/q\mathbb{Z})^N$ and $e \in \mathbb{R}^N$ [8].

Parameter testing tool

Martin Albrecht, professor at the university of Royal Holloway in London, is the current maintainer of an open-source project [1] geared towards choosing secure parameters for the RLWE encryption scheme, which can be found at `https://github.com/malb/lattice-estimator`. By using this tool, we will be able to calculate the amount of security by using the lattice reduction algorithm BKZ [2]. We will be using this tool to evaluate the equivalent symmetric security level against an adversary with access to a large-scale quantum computer, while still holding the degree of the function $f(x)$ to a minimum for performance reasons. The tool will use lattice based mathemathics to calculate the amount of security, which will be given to us in the form of a number β, that will then be used to test the security level. More specifically, to utilise the tool we have used the following Sage script:

```
from estimator import *
from estimator.lwe_parameters import *
from estimator.nd import *

N = 2048
q = 80708763
Xe = NoiseDistribution(3.2)
Xs = NoiseDistribution(3.2)
m = 2*N

params = LWEParameters(N, q, Xs, Xe, m, tag="params")
result = LWE.estimate.rough(params)
print(result)
```

In this, we can vary the parameters of our modulus q, the degree of the cyclotomic-polynomial N, our noise distributions on r and r' (Xs, Xe) and the value of m typically provided by the authors.

Best known attack

The best known attack on the LWE scheme is the BKZ lattice reduction algorithm, which by the earlier mentioned method to divide RLWE into N LWE samples, also gives an attack on RLWE. We will not be going into how BKZ works, but will mention

that the security level from the BKZ algorithm has been shown in the "New Hope" paper [2] written by Alkim and Poppelman to be

$$2^{0.292 \cdot \beta}$$

where we can get the value of β from the testing tool.

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 the homomorphic operations, 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 [6].

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 p) computation done in **decrypt** anyways, as long as the requirements on the ℓ_∞ norm are still satisfied.

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 [5] the authors describe a protocol for MPC based on a SHE scheme. The protocol is able to 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 $[\![\cdot]\!], \langle \cdot \rangle$.

The first 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 $\beta_i \in \mathbb{F}_{p^k}$. 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 $a = \sum_i a_i$.

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 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.

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 Brakerski and Vaikuntanathan [3] described in section 2.1.

encode and decode To be able to use this scheme we have to define the function **encode**, which maps elements in the plaintext space of the MPC protocol $M = (\mathbb{F}_{p^k})^s$ to elements in a ring R, which is equivalent to $\mathbb{Z}^N$. In addition to this we also need to define a function **decode**, which maps elements in $\mathbb{Z}^N$ to M, s.t. $\mathbf{decode}(\mathbf{encode}(m)) = m$ for $m \in M$.

To do this we first have to pick a polynomial for the quotient ring $R = \mathbb{Z}[X]/\langle f(x)\rangle$ used by the cryptosystem. Picking $f(x)$ in a specific way allows an optimization, which makes it possible to do component-wise multiplication, such that we can perform N multiplications in parallel [5]. To do this we pick $f(x)$ to be the m'th cyclotomic polynomial $f(x)$ of degree $N = \phi(m)$, s.t. modulo p the polynomial $f(x)$ factors into l' irreducible factors of degree k', where $l' \geq s$ and k divides k'.

We can then define the function $\phi : M \to R_p$ which embeds M into R_p. We also define $\iota : R_p \to \mathbb{Z}^N$, which maps the coefficients from the polynomial given as input to a vector of length N with coefficients in the range $(-p/2, ..., p/2]$.

Finally, we define $\mathbf{encode}(\mathbf{m}) = \iota(\phi(\mathbf{m}))$ and $\mathbf{decode}(\mathbf{x}) = \phi^{-1}(\mathbf{x} \pmod p)$.

key distribution and distributed decryption 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. We have omitted the description of a protocol that implements the functionality, but it can be found in [5].

2.3.2 Preprocessing phase

The preprocessing phase is implemented by the following protocol, which consists of the steps **initialize**, **pair**, and **triple**. These steps use the additional protocols Reshare, PAngle, and PBracket as subroutines [5].

Protocol Reshare: The Reshare protocol distributes shares of a plaintext, given the ciphertext, to all parties, without revealing the plaintext to any of the players. Specifically, it 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 the plaintext m to each player along with a new fresh ciphertext e'_m if $enc = NewCiphertext$, where e'_m contains $\sum_i m_i$. The protocol proceeds as follows:

1. Each player P_i samples $f_i \in \mathbb{F}_{p^k}$, and then broadcasts $e_{f_i} \leftarrow Enc_{pk}(f_i)$.
2. Each player P_i runs the ZKPoPK protocol (as defined in 2.5) as a prover on e_{f_i}, aborting if any proof fails.
3. Each player P_i homomorphically adds $e_f \leftarrow e_{f_1} \boxplus ... \boxplus e_{f_n}$ and $e_{m+f} \leftarrow e_f \boxplus e_m$.
4. The players collectively use $\mathscr{F}_{KeyGenDec}$ to decrypt e_{m+f} so that they get $m+f$.
5. Player P_1 sets $m_1 \leftarrow m+f-f_1$, and each other players P_i sets $m_i \leftarrow -f_i$.
6. If $enc = NewCiphertext$, then each player each computes $e'_m \leftarrow Enc_{pk}(m+f) \boxminus e_{f_1} \boxminus ... \boxminus e_{f_n}$ using default randomness, and gets output (m_i, e'_m).
7. If $enc = NoNewCiphertext$, then each player P_i gets output m_i.

Protocol PBracket: The PBracket protocol produces a value in the $[\![v]\!]$ representation, given a ciphertext e_v along with privately held shares $v_1, ..., v_n$.

1. For $i = 1, \ldots, n$, every player P_j computes $e_{\gamma_i} \leftarrow e_{\beta_i} \boxtimes e_v$, and gets the share γ_i^j by calling Reshare with $e_m = e_{\gamma_i}$ and $enc = NoNewCiphertext$.
2. The players output $[\![v]\!] = ((v_1, ..., v_n), (\beta_i, \gamma(v)_1^i, ..., \gamma(v)_n^i)_{i=1,...,n})$.

Protocol PAngle: The PAngle protocol produces a value in the $\langle v \rangle$ representation, given a ciphertext e_v along with privately held shares $v_1, ..., v_n$.

1. Each player P_i computes $e_{v \cdot a} \leftarrow e_v \boxplus e_\alpha$.
2. The players collectively run Reshare with $e_m = e_{v \cdot \alpha}$ and $enc = NoNewCiphertext$, such that each player P_i receives a share γ_i of $v \cdot \alpha$.
3. The players output $\langle v \rangle = (0, (v_1, ..., v_n), (\gamma_1, ..., \gamma_n))$.

Initialize: The **initialize** step generates the global and personal keys.

1. Each player sends "Start" to $\mathscr{F}_{KeyGenDec}$ and obtains the public key pk.
2. Each player P_i 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 $Diag(a) = (a, a, ..., a) \in (\mathbb{F}_{p^k})^s$.
3. Each player P_i runs the ZKPoPK protocol twice as a prover with $diag = True$, on e_{α_i} and e_{β_i}, each repeated sec times, aborting if any proof fails.
4. Each player P_i homomorphically adds the encrypted shares e_{α_i} to get e_α.
5. The players collectively run PBracket with their shares $\alpha_1, \ldots, \alpha_n$ to obtain $[\![Diag(\alpha)]\!]$.
6. Then the players output $[\![Diag(\alpha)]\!]$ as the global key and each P_i gets β_i as their personal key.

Pair: In **pair** the players generate random values in the two representations $[\![r]\!], \langle r \rangle$.

1. Each player P_i samples a share $r_i \in (\mathbb{F}_{p^k})^s$ and broadcasts $e_{r_i} \leftarrow Enc_{pk}(r_i)$.
2. Each player P_i runs the ZKPoPK protocol as a prover on e_{r_i}, aborting if any proof fails.
3. Each player P_i homomorphically adds the encrypted shares to get $e_r \leftarrow e_{r_1} \boxplus \cdots \boxplus e_{e_n}$.
4. The players collectively run PBracket to get $[\![r]\!]$ and PAngle to get $\langle r \rangle$.

Triple: The **triple** step generates triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ satisfying $c = ab$.

1. Each player P_i samples $a_i, b_i \in (\mathbb{F}_{p^k})^s$ and broadcasts $e_{a_i} \leftarrow Enc_{pk}(a_i)$, $e_{b_i} \leftarrow Enc_{pk}(b_i)$.
2. Each P_i runs the ZKPoPK protocol as a prover first on e_{a_i} and then on e_{b_i}, aborting if any proof fails.
3. Each player P_i homomorphically adds the encrypted shares to get e_a and e_b.
4. The players collectively run PAngle to get $\langle a \rangle$ and $\langle b \rangle$.
5. Each player P_i homomorphically multiplies e_a and e_b to get e_c.
6. The players collectively run Reshare with $e_m = e_c$ and $enc = NewCiphertext$, such that each player P_i receives a share c_i of c and a new ciphertext e'_c.
7. The players collectively run PAngle to get $\langle c \rangle$.
8. Each player P_i gets the output $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$.

2.3.3 Online phase

The Online protocol implements the online phase, and consists of the steps **initialize**, **input**, **add**, **multiply**, and **output** [5]. These steps are executed as needed to evaluate the arithmetic circuit that we wish to evaluate.

Initialize: The **initialize** step simply consists using the preprocessing protocol to generate a global key $[\![\alpha]\!]$, along with enough multiplicative triples and random values in the $\langle\cdot\rangle$ and $[\![\cdot]\!]$ 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 using one pair $([\![r]\!], \langle r\rangle)$ from the preprocessing phase.

1. Each player P_j sends their share $[\![r]\!]$ to P_i, allowing P_i to open r.
2. P_i computes $\varepsilon \leftarrow x_i - r$ and broadcasts it to all players.
3. Each player P_j sets $\langle x_i\rangle \leftarrow \langle r\rangle + \varepsilon$.

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 $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$, and a shared random value $[\![t]\!]$.

We use the second triple to check that $ab = c$, but this could also be done in preprocessing instead.

1. Each player P_i broadcasts their share $[\![t]\!]$, and uses the broadcasted values to compute t.
2. Each player P_i computes $t \cdot \langle a\rangle - \langle f\rangle$, and the players partially open the result to get ρ.
3. Each player P_i computes $\langle b\rangle - \langle g\rangle$, and the players partially open the result to get σ.
4. Each player P_i computes the value $t \cdot \langle c\rangle - \langle h\rangle - \sigma \cdot \langle f\rangle - \rho \cdot \langle g\rangle - \sigma \cdot \rho$, and the players partially open the result. If the result is non-zero, the protocol is aborted. Otherwise, $ab = c$.
5. The players partially open $\langle x\rangle - \langle a\rangle$ to get ε and $\langle y\rangle - \langle b\rangle$ to get δ.
6. Each player P_i computes and outputs $\langle z\rangle \leftarrow \langle c\rangle + \varepsilon \cdot \langle b\rangle + \delta \cdot \langle a\rangle + \varepsilon \cdot \delta$.

Output: To output a value y given $\langle y \rangle$, we use a random value $[\![e]\!]$ and a commitment functionality $\mathscr{F}_{Commit}$, as described in [5].

1. Each player P_i broadcasts their share $[\![e]\!]$, and uses the broadcasted values to compute e.
2. Each player P_i computes
$$a = \sum_j e^j \cdot a_j$$
for each opened value a_j of the form $\langle a_j \rangle$.
3. Each player P_i uses $\mathscr{F}_{Commit}$ to commit to the values $\gamma_i \leftarrow \sum_j e^j \cdot \gamma(a_j)_i$, y_i, and $\gamma(y)_i$.
4. Each player P_i broadcasts their share $[\![\alpha]\!]$, and uses the broadcasted values to compute α.
5. Each player P_i uses $\mathscr{F}_{Commit}$ to open γ_i, checks that
$$\alpha(a + \sum_j e^j \delta_j) = \sum_j \gamma_i$$
and aborts the protocol if not.
6. Each player P_i uses $\mathscr{F}_{Commit}$ to open y_i and $\gamma(y)_i$, checks that
$$\alpha(y + \delta) = \sum_i \gamma(y)_i$$
and aborts the protocol if not.
7. Each player outputs $y \leftarrow \sum_i y_i$.

2.4 Reuse of unrevealed secret-shared data

In [4] a technique that allows for reuse of unrevealed secret-shared data is used. This technique revolves around not having to reveal the global key $[\![\alpha]\!]$, and in fact we do not even need the $[\![\cdot]\!]$ representation when using this technique.

The technique works as follows:

1. When generating the global key α, each player P_i gets a share α_i of the key α.
2. In the **output** step, each player invokes the **MACCheck** protocol (instead of using the previous method, where α is opened) on all values in the $\langle \cdot \rangle$ representation that have been opened, aborting if the check fails.
3. Finally, each player invokes the **MACCheck** protocol on $\langle y \rangle$, outputting y if it succeeds.

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

$$s \leftarrow s_1 \oplus ... \oplus s_n$$

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

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

where the a_j's are the values in the $\langle \cdot \rangle$ representation that have been opened. 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 $\sigma_1 + ... + \sigma_n = 0$, and if this is not the case, then they abort.

2.5 Zero-knowledge proof

In [5] a ZK protocol called Zero Knowledge Proof of Plaintext Knowledge (ZPoPK) is presented. The protocol is run with *SEC* ciphertexts $c_1, ..., c_{SEC}$, which have been generated by one of the players, as input.

The purpose of the this protocol is that if the prover behaves honestly while running the protocol, then the ciphertexts are validly generated and the prover knows the plaintext. More specifically, $c_i = \text{Enc}(x_i, r_i)$, where x_i has been obtained from **encode**, and the randomness r_i used for encryption has been sampled from the error distribution χ. The protocol is thus a ZK proof of knowledge for the following relation

$$\begin{aligned} R_{PoPK} = \{(x,w) | x = (pk, c), w = ((x_1, r_1), ..., (x_{SEC}, r_{SEC})) : \\ c = (c_1, ..., c_{SEC}), c_i = Enc_{pk}(x_i, r_i), \\ ||x_i||_\infty \leq B_{plain}, \text{decode}(x_i) \in (\mathbb{F}_{p^k})^s, ||r_i||_\infty \leq B_{rand}\} \end{aligned}$$

where B_{plain} and B_{rand} are bounds on the coefficients in the plaintext and randomness respectively. A detailed description of the protocol has been omitted, but can be found in [5].

2.6 Parameter setting for the MPC system

When choosing our parameter sets for the MPC system it is important to think not only of the security, but also of the practicality of the systems. We are already limited in the parameters we can pick from the restrictions outlined in section x.x, but in the article [5] from Ivan et al. we are provided with additional limitations that we have to adhere to.

We will here outline these limitations, but will omit the proofs and justification as those are outside the scope of our project. Let $N = deg(f(x))$, let r be the standard deviation of our error distribution χ, SEC be 40, the amount of players $n = 3$ with c_{SEC}, Y and Z defined by

$$\begin{aligned} c_{sec} &= 9 \cdot N^2 \cdot SEC^4 \cdot 2^{SEC+8} \\ Y &= \frac{p}{2} + p \cdot (4 \cdot C_m \cdot r^2 \cdot N^2 \cdot 2 \cdot \sqrt{N} \cdot r + 4 \cdot C_m \cdot r^2 \cdot N^2) \\ Z &= C_m \cdot N^2 \cdot n^2 \cdot c_{sec}^2 \cdot Y^2 + n \cdot c_{sec} \cdot Y \end{aligned}$$

where C_m is a constant, as defined in [5]. Then the article specifies that the following inequalities for q and r need to hold.

$$\begin{aligned} q &> 2 \cdot Z \cdot (1 + 2^{sec}) \\ r &> max\{3.2, 1.5 \cdot \gamma^{-t'} \cdot q^{1-\frac{N}{t'}}\} \end{aligned}$$

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. For this protocol we had to be able to represent arbitrarily large integers, and to do this we use the `Integer` type from the "rug" rust library.

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_∞ value, 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 $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$. 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`. This struct represents an instantiation of a quotient ring $\mathbb{Z}_q[x]\langle f(x)\rangle$. It therefore has fields q and *modulo*, where q is an `Integer`, 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 performs synthetic division with `pol` as the dividend and `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 `modulo` method defined in the `poly.rs` module, and then return the result.

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

The methods `add`, `sub`, `times`, `neg`, `mul` are called on an `Rq` struct. These methods first use the addition, scalar multiplication, negation, and polynomial multiplication methods (or some combination thereof), as defined in the the `poly.rs` module on the input. Then, `reduce` is called, and the result is returned.

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

In addition to this we also have two functions responsible for the homomorphic operations, namely `add` and `mul`.

`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, p, 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 R_q.

It then calculates the public-key as $pk = (a_0, a_0 \cdot sk + e_0 \cdot p)$, 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_p. 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 $b = b_0 \cdot v + e'' \cdot p$. 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 $c = [c_0, c_1, \dots]$ and a secret key sk.

We start by constructing the secret key vector $\mathbf{s} = [1, s, s^2, \dots]$ 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.

Since the message has coefficients in $\mathbb{Z}_q$, but we need them to be in the interval from $-\frac{q}{2}$ to $\frac{q}{2}$, we call the `normalized_coefficients` method on the msg polynomial at this point.

At this point, we want to ensure that the ℓ_∞ for the message is at most $\frac{q}{2}$. If not, decryption cannot work.

If the check succeeds, we reduce the message modulo p to remove the $e \cdot p$ part of the encryption, and then we return the result.

add

The `add` function homomorphically adds the two ciphertexts `c1` and `c2` given as input along with a `Parameters` struct.

To do this computation the function simply creates a new `Vec` of length `max(c1.len(), c2.len())`. Following this the function iterates over the two ciphertexts and for each entry i, it adds `c1[i]` and `c2[i]` to entry i in the newly created `Vec` using `add` from `quotient_ring.rs`. After iterating over all entries the resulting `Vec` is returned.

mul

The `mul` function homomorphically multiplies the two ciphertexts `c1` and `c2`, which it takes as input along with a `Parameters` struct.

First, a new `Vec` called `res` is initialized. For each entry i in `c1` and each entry j in `c2` the function adds $c1[i] \cdot c2[j]$ to the entry `res[i + j]`, where the addition and multiplication is done using `add` and `mul` from `quotient_ring.rs`. Afterwards `res` is returned.

3.2 MPC protocol

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

For this protocol we did not implement the component-wise multiplication optimisation described earlier, so $s = 1$, meaning that we are working in $\mathbb{F}_{p^k} = \mathbb{Z}_p$. This also means that we cannot do component-wise multiplications like done in [5]. Therefore, to encode an element $a \in \mathbb{F}_{p^k}$ to an element in R_p, we simply return `Polynomial::new(vec![a])`, which is the `Polynomial` corresponding to $f(x) = a$. To decode a polynomial $f(x) = a \in R_p$, we simply return the `Integer` a.

For testing purposes, we implemented three protocols: one where all inputs are added together, one where all inputs are multiplied, and one specifically for three players, where $x_1 \cdot x_2 + x_3$ is computed. The set of protocols is easily extendable using our framework.

3.2.1 Dealer and Player

There are two types of actors in this system, the *dealer* and the *player*.

The *dealer* acts as a trusted party for establishing connections between all parties and, once all parties have connected, provides the parties with key material (i.e. a public key and shares to the corresponding secret key), enabling distributed decryption. For practical purposes, all players only need to know the address of the dealer, who will then forward the receiving addresses of all players to each other, enabling point-to-point communication.

All other parties in the protocol are *players*. A player contributes some private input to the function being computed, in the end receiving only the result and no other information.

To prepare for computing the agreed function, all players connect to the dealer, sending a "Start" message with their address for receiving TCP connections, and getting back a list of players who have already connected to the dealer (along with themselves). Each player's position in this list indicates their "player number", which is relevant for some parts of the protocol. When all players have connected, and all of the key material has been distributed, the dealer shuts down, as it is no longer needed. This is also described in figure 3.1.

Facilitator

The `Facilitator` is a small, but central component that enables communication between the parties while running the protocol. While it is not a theoretically important part of the system, it ended up being used in almost all other components, and will therefore be summarized here.

After the dealer has shut down, each player uses a `Facilitator` to communicate with the other players, providing functions for broadcasting or sending point-to-point messages (`broadcast` and `send`), as well as for receiving messages from a designated player or from all players at once (`receive` and `receive_from_all`).

To ensure that all messages are received in the desired order, sending a message is handled entirely synchronously for the player (i.e. the `send` function is handled in the

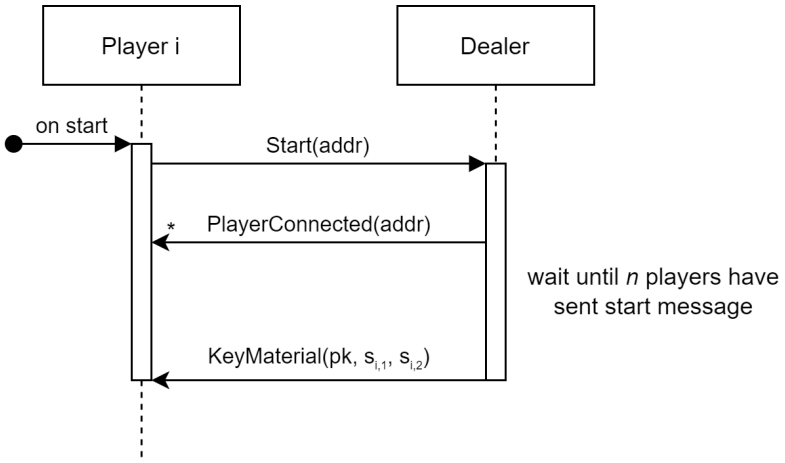

Figure 3.1: A sequence diagram describing the protocol for player initialization using the trusted dealer.

main thread, and therefore will not return until the message has been accepted by the other party).

Oppositely, receiving messages is handled by a dedicated thread that holds a `TCPListener` (the one for which the address was published). Once a messages is received, it is added to a channel corresponding to the player that sent the message, ready to be read whenever the receiving player is ready. If no message is ready, the `receive` function blocks until a message is received.

The system currently does not have any mechanism for verifying the origin of messages, as a new connection is opened on a new port each time a player wants to send a message to another player. Therefore, each player sends along their own identity when sending a message, which would obviously be insecure in a production-ready system. This issue could be resolved in many ways, for example using public-key cryptography and authenticated channels.

PlayerState

The `PlayerState` struct is also frequently used in our implementation. Similarly to the `Parameters` struct, it is a container for most of the values that need to be known by the player for most of the MPC protocol.

It includes the key material received from the dealer $(pk, s_{i,1}, s_{i,2})$, our share of the global MAC key α_i, the encrypted global key e_α, a list of pairs $(a, \gamma(a)_i)$ (where a is a value that has been opened, and $\gamma(a)_i$ is our share of αa), and a `Facilitator` instance.

3.2.2 Distributed decryption

The `ddec` function implements the distributed decryption part of the $\mathcal{F}_{KeyGenDec}$ functionality described earlier. The function takes a `Parameters` struct `params`, a `PlayerState` `state`, and a mutable `Ciphertext` `c` as arguments.

First we check if `c` contains 3 elements, and if this is not the case then we pad the ciphertext with a 0, since we need there to be three entries for the following

computations.

If it is P_1 calling the method, then we use our quotient ring implementation to compute

```
v_i = c[0] + state.sk_i1 * c[1] + state.sk_i2 * c[2]
```

otherwise we compute

```
v_i = state.sk_i1 * c[1] + state.sk_i2 * c[2]
```

Now we use `sample_from_uniform` to sample a polynomial with ℓ_∞ norm bounded by $2^{sec} \cdot B/(n \cdot p)$, meaning that the coefficients are in the interval $[0, 2^{sec} \cdot B/(n \cdot p)]$, where B is a bound on the ℓ_∞ norm of t. The reason why we let the coefficients depend on B, is that we do not want to add too much noise, since this would result in decryption returning the incorrect result.

We then add p times the polynomial that we just sampled to v_i, so that we get t_i. Adding this randomness corresponds to drowning the noise of the ciphertext as described in the section on circuit security.

Following this we broadcast t_i, and then receive the other players shares, which allows us to compute $t' = t_1 + ... + t_n \mod p$.

We then normalize the coefficients of t' to get `msg_minus_q` like done in `decrypt`, so that they are in the interval $[-q/2, q/2)$.

Finally, we compute `msg_mod_q.modulo(&params.p)`, then call decode on the result to get an Integer which we return.

3.2.3 ZKPoPK (mpc/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" [5]. Due to time constraints we had to simplify the implementation, by always letting the input to the protocol be the same ciphertext c repeated *SEC* times. This is a requirement when running the protocol for the initialize step of preprocessing anyways, but in pair this results in a significant slowdown, as we cannot generate *SEC* pairs in parallel as we would otherwise have been able to. The file is divided into 2 different functions.

make_zkpopk To generate a zero knowledge proof, we call the function make_zkpopk, which takes 6 arguments: A `Parameters` struct `params`, a plaintext `x`, randomness `r`, a `Ciphertext` `c`, a `bool` `diagonal`, and a public key `pk`. make_zkpopk will generate 3 values (a, z, T), which will be used in the verification of the proof.

verify_zkpopk To verify the proof we call the function verify_zkpopk, which takes as input 6 arguments as well: A `Parameters` struct `params`, Integer matrices `a, z, t`, a `Ciphertext` `c`, and a public key `pk`. verify_zkpopk will return a boolean indicating whether or not the proof was valid.

3.2.4 Preprocessing phase (mpc/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 the 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 $\mathbb{Z}_p$ 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 $e_{m+f} - f_i \bmod p$ if the player calling the function is the player with index 0, and $-f_i \bmod p$ 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 [4], 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 $\mathbb{Z}_p$ using the `sample_single` function with `params.p` 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. Then, we run the ZKPoPK protocol by calling `make_zkpopk` with `e_alpha_i` as input, and then calling `verify_zkpopk` on the values received from the other players.

Notice how we do not compute the personal keys β_i as done in [5]. 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 $\mathbb{Z}_p$. Now, `r_i` is encrypted to get `e_r_i`, which is then 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 run the ZKPoPK protocol by calling `make_zkpopk` with `e_r_i` as input, and then calling `verify_zkpopk` on the values received from the other players.

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 $\mathbb{Z}_p$, which are both then encrypted, and the encrypted shares are then broadcast.

Afterwards, we run the ZKPoPK protocol by calling `make_zkpopk` first with `e_a_i` as input, then `e_b_i`, and then calling `verify_zkpopk` on the values received from the other players.

Then, when the player receives the encrypted shares of a and b from the other players, 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`.

3.2.5 Commitments (mpc/commit.rs)

For the online phase we also need a commitment functionality as described in [4]. 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.

3.2.6 Online phase (mpc/online.rs)

The code related to the online phase can be found in `mpc/online.rs`. The implementation of this phase is based on the online protocol in [4], but in that protocol the triple check is done in preprocessing, while we do it in the online phase as in [5].

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 $r = r_1 + ... + r_n \mod p$.

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 t $(a_j, \gamma(a_j)_i)$ 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 `r` with t 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`.

3.3 Testing of implementation

For testing our implementation we used both automated unit tests and "semi-automated" integration tests.

3.3.1 Automated tests

The automated unit tests cover the polynomial implementation (`poly.rs`), the quotient ring implementation (`quotient_ring.rs`), the Ring-LWE encryption scheme (`encryption.rs`), and the MPC implementation (`mpc/prep.rs` and `mpc/online.rs`).

The automated tests for the polynomial and quotient ring implementations ensure that the basic operations: addition, subtraction, multiplication, scalar multiplication, negation, and reduce work as expected.

The tests that cover the Ring-LWE encryption scheme ensure consistency, i.e. $\text{decrypt}(\text{encrypt}(m)) = m$, and additionally checks that the homomorphic operations add and mul work correctly.

For the automated tests for the MPC implementation we only run 1 player, and we generate the key material for the player directly, instead of running the dealer. We decided to conduct our automated tests of the MPC system in this way, since the added complexity from the communication with the other players made it hard to test.

This is of course not sufficient by itself, so to test the system in a more realistic setting we turn to manual, or "semi-automated" integration tests.

3.3.2 Integration tests

The "semi-automated" integration tests consists of a bash script `run.sh`, that simply starts a dealer and three players, which then execute the protocol. The function that is computed then depends on the `protocol` enum which can be set in `player.rs`. The script then saves logs which contain the inputs and outputs for each player, and then we can check manually that the correct result was computed.

Chapter 4

Evaluation

In the following chapter we will first outline our process of picking parameters for our MPC system. Following this we use these considerations to evaluate the performance of the system, and discuss why we see the results that we do.

4.1 Choosing parameters for our MPC system

We have in sections 2.1.4 and 2.6 previously outlined some limitations on the parameter sets that we can choose. We use the previously mentioned lattice reduction tool to evaluate the equivalent amount of bits of symmetric security level that we get. To ensure that our chosen values for q and r satisfy the requirements outlined in section 2.6, we wrote a small program, which can be seen in appendix B.

We choose $f(x) = x^N + 1$ for $N = 2^k$ as our cyclotonmic polynomial as done in [3], and for the standard deviations of the error distributions, we let $r = r'$ as done in [5]. With the script we found that a 512 degree polynomial with $p = 127$ would result in q being of 315 bits in size, and our r parameter (used for encryption in the mpc protocol) should be above $3.85 \cdot 10^{73}$. While these values provide enough security, the value of the noise standard deviation r is way too high to be practical in any way, since we would likely end up with too much noise for us to decrypt, and as a consequence we need to decrease r to avoid such a situation.

By using the tool from Martin Albrecht [1] and using the values, which we got from the program above, we can see that setting q to some prime of size > 313 bits and setting the degree of our cyclotomic polynomial $f(x)$ to be $N = 12900$, we will have around 627 bits of security against the BKZ lattice reduction. These tests also align with the how the parameters in [5] are chosen, as described in section 2.6.

While this provides enough security, it is by no means effective for us to run, and we do not have the component-wise multiplication optimisation, meaning that the efficiency of our implementation degrades further. Therefore, we have to settle for a smaller degree, but still leave q high enough make space for the noise. From an empirical analysis we see that our implementation becomes very slow when $deg(f(x)) > 1024$, and we still have to provide a q that is large enough. Running these parameters through the lwe-estimator, we can see that our security level decreases down to around 13 bits, which is of course not sufficient.

N	Time (s)
8	19.900202
16	38.093643
32	79.117386
64	165.27484
128	397.68887
256	923.9578
512	2300.92
1024	6230.6777

Table 4.1: A table relating the size of the N parameter to the wallclock run-time of the entire MPC system, for $n = 3$ with the function $x_1 \cdot x_2 + x_3$. The timer was started as soon as all players had connected to the system and received the key material, and was stopped when the output had been calculated.

4.2 Measuring system performance

To get an estimate on the performance of the system, we ran the program a number of times with different parameters. Specifically, these parameters were chosen to be secure, as described in section 2.6, but with the degree of the polynomials N set to different values to test the impact that would have on the performance of the system.

The results can be seen in table 4.1. Performing quadratic regression on the results, we get an expression of the form $f(x) = ax^2 + bx + c$ with

$$a = 0.00312337$$
$$b = 2.90545$$
$$c = -17.2764$$

where $R^2 \approx 1$. This shows that the polynomial system is clearly quadratic in its runtime. Though the quadratic coefficient term is negligible for the smaller values of N, it ends up contributing more than half of the output value for $N = 1024$.

Running the system with $N = 12900$ as the degree of the polynomial, this regression suggests that the computation would take 557223.03 seconds (or 6.45 days), making it completely infeasible in practice. It is worth noting, however, that a large majority of the time is spent in the preprocessing phase, and that the online phase is still relatively efficient.

As we had previously implemented the polynomial reduction method using polynomial long division instead of synthetic division, we were able to run these tests for both cases and compare the results. See table A.1.

The performance for our implementation is clearly sub-par, which may be due to multiple factors:

- The zero-knowledge proof is in many cases run too many times, causing a major slowdown in the preprocessing phase (see 3.2.3).

- The system does not implement multiplication using parallel SIMD processing, as in [5].
- Our implementations of polynomial operations and ring quotient operations were not made with performance as a main priority.
- In many cases, values are copied unnecessarily instead of edited in place, which may cause slowdowns. Note that compiler optimizations may eliminate this in some cases.

Chapter 5

Conclusion

We have implemented the system as described by Ivan et al. [5], with some modifications to better suit the Rust programming language and the specifics of our system, as well as to improve the performance and allow reuse of secret-shared material.

The preprocessing phase is clearly a bottleneck in the system, which may be due to multiple factors, most notably not having implemented component-wise multiplication.

Without the implementation of component-wise multiplication, the system seems to run into one of two scenarios. In one scenario, the cryptosystem is insecure with respect to the results from the LWE estimator tool, due to choosing too small parameters. In the other scenario, the MPC system is too slow to use in practice.

For future work, it could be valuable to improve the performance of the system, so that it can be run practically while still being secure.

Acknowledgments

...

Bibliography

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. `https://ia.cr/2015/046`.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key {Exchange—A} new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, 2016.

[3] 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.

[4] 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.

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

[6] 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.

[7] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Chris Peikert. How (not) to instantiate ring-lwe. In *International Conference on Security and Cryptography for Networks*, pages 411–430. Springer, 2016.

[9] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

Appendix A

MPC system performance with polynomial long division

N	Time (s)
8	19.036512
16	39.95138
32	85.003136
64	200.78773
128	568.9214
256	1631.5626
512	5160.727

Table A.1: A table relating the size of the N parameter to the wallclock run-time of the entire MPC system, for $n = 3$ with the function $x_1 \cdot x_2 + x_3$ when using polynomial long division.

Appendix B

Parameter search Python script

```
from math import sqrt, floor, log2

N = 512
p = 127
r = 3.2
SEC = 40
C_m = 8.6
n = 3

# The values for the formulas
csec = 9 * N**2 * SEC**4 * 2**(SEC + 8)
Y = p/2 + p * (4 * C_m * r**2 * N**2 + 2 * sqrt(N) * r + 4 *
    C_m * r**2 * N**2)
Z = C_m * N**2 * n**2 * csec**2 * Y**2 + n * csec * Y

q_size = 2 * Z * (1 + 2**SEC)
print("The bit size of q should be above: ",
      floor(log2(q_size)))

chosen_q = int("""6440092097492369874468694478456476902429
935263779065830479393474203066496323859298183983608879""".
    replace('\n', ''))
gamma = 1.005
t_prime = sqrt(N * (log2(chosen_q)/log2(gamma)))
r_bound = max(3.2, 1.5 * gamma**(-t_prime) * chosen_q**(1 - (N
     / t_prime)))
print("The value of the randomness r should be above: ",
    r_bound)
```