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

placeholder

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

Contents

Chapter 1

Introduction

placeholder

Chapter 2

Review of literature

2.1 Encoding messages as polynomials

Encoding messages as polynomials follows simply by the fact that a letter can be represented by a number. These numbers can then be combined into a vector which can be encoded as a polynomial. As an example take the message: "CRYPTO", using the ASCII table, this message can be represented by the vector $[67, 82, 89, 80, 84, 79]$. This vector can be represented as a polynomial by $67 + 82x + 89x^2 + 80x' + 84x^4 + 79x^5$. Trivially we can also decode any polynomial into a message, by reverting these steps one by one.

2.2 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 [] 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 $R_t = \mathbb{Z}_t[x]/\langle f(x)\rangle$.

2.2.1 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 $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$.

Key-gen

Let our secret key be a randomly sampled element from the error distribution $s \leftarrow^{\$} \chi$ 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

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

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 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

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

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

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

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

2.2.2 Public key version

2.3 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 https://www.cs.cmu.edu/ odonnell/hits09/gentry-homomorphic-encryption.pdf.

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.

[1]

Chapter 3

Implementation

►Dette skal nok opdateres nÃ¥r vi er fÃ¦rdige med implementationen. Skal inkludere det om MPC.◄ We implemented a somewhat-homomorphic public-key encryption scheme in the Rust programming language, as well as functions for adding and multiplying ciphertexts for that encryption scheme. **►Hvor meget skal forklares her? At det er Brakersky-Vaikuntanathan vi implementerede? At det er baseret pÃ¥ Ring-LWE? Eller bliver det alt sammen beskrevet et andet sted?◄**

3.1 Polynomials (`poly.rs`)

Since our public-key encryption scheme uses polynomials to represent all of the values we work on (messages, ciphertexts, secret keys, and public keys), we made an implementa

3.2 Quotient ring (`quotient_ring.rs`)

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

3.2.1 `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`.

3.2.2 `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.

3.2.3 `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.

3.3 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

3.3.1 `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).

3.3.2 `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` (a contiguous growable array type) $c = [b + m, a]$ and return it.

3.3.3 `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.

►Skal nok lige have nogle andre til at tilpasse denne - jeg har svÃ¦rt ved at finde ud af hvordan man forklarer det.◄ Currently, the message has coefficients in $\mathbb{Z}_q$, but we need them to be in the range $-\frac{q}{2}$ to $\frac{q}{2}$. **►Er det inklusive eller eksklusive i intervallet?◄** 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.

Chapter 4

Conclusion

...

Acknowledgments

We would like to thank no one.

Bibliography

[1] Bertoni Guido, Daemen Joan, P Michaël, and VA Gilles. Cryptographic sponge functions, 2011.

Appendix A

Placeholder