Project 1: Homomorphic Encryption from RLWE

Computer Algebra for Cryptography (B-KUL-H0E74A)

Deadline: 4pm (Leuven time) on 01/04/2025

Homomorphic encryption is one of the hottest topics in cryptography since it allows to manipulate encrypted data without decrypting it first. The prototypical use case for this technology is where a user encrypts its data and stores the encrypted data in the cloud, whilst still allowing the server to perform computations on this encrypted data. To illustrate this with a simple example: assume I encrypt the marks for each student, and store these on a server, I would still be able to ask the server to compute **an encryption** of the average mark, and send it back to me. The server has no knowledge on the individual marks, it only knows it has to compute the average of all the encrypted marks.

The goal of this project is to implement a basic version of the BGV cryptosystem [1] and mount several attacks on it.

1 Submission

The result of this project consists of two parts: one .m file containing all the Magma code you wrote during the project and one .pdf file containing your report. Note that the report should be **max 5 pages (not counting references)**. Reports written in LaTeX are preferred, but this is not mandatory. Please follow the numbering of the tasks in this document, e.g. Task 1.a and so on.

Make sure that **both** files clearly contain your **name and student number**.

The two files should be submitted via Toledo under Project 1. The deadline is 01/04/2025 at 4pm (Leuven time).

2 The BGV cryptosystem

General warning: the description below uses indexing starting from 0 which is mathematically convenient. Magma indexing starts from 1.

The BGV cryptosystem is based on the Ring-Learning-With-Errors problem

(RLWE) which consists of the following ingredients:

- $R = \mathbb{Z}[x]/(f(x))$ the quotient ring of polynomials with integral coefficients modulo $f(x)$. In practice (and also in our project), $f(x) = x^n + 1$ with $n = 2^k$, which equals the $2n$-th cyclotomic polynomial. Given an element $\mathbf{a} \in R$, we will denote $||\mathbf{a}||_2$ and $||\mathbf{a}||_\infty$ the 2-norm and ∞-norm of its coefficient vector, i.e. $\sqrt{\sum a_i^2}$ and $\max_i\{|a_i|\}$ where the a_i are the n coefficients of the polynomial $\mathbf{a}$.
- A ciphertext modulus q (which does not need to be prime) and the quotient ring $R_q = R/(qR)$, i.e. elements in R whose coefficients are reduced modulo q. In several places we will need centered reduction, which means that we will take the representative in the interval $]-q/2, q/2]$ instead of the usual $[0, q[$. For an integer $z \in \mathbb{Z}$, we denote this as $[z]_q \equiv z \bmod q$ and $[z]_q \in]-q/2, q/2]$. We extend this notation coefficient-wise to polynomials and elements of the ring R_q. For an element $\mathbf{a} \in R_q$ we will denote $||\mathbf{a}||_2$ and $||\mathbf{a}||_\infty$ the corresponding norms of the centered reduction $[\mathbf{a}]_q$.
- The private key is given by a polynomial $\mathbf{s} \in R$ with small coefficients; in practice it is chosen ternary, meaning its coefficients are in the set $\{-1, 0, 1\}$.
- An error distribution χ defined on R resulting in polynomials with small coefficients. In practice, each coefficient is sampled uniformly from the interval $[-B, B]$ or sampled from a binomial distribution. We will simply use the uniform distribution defined by $B \ll q$.

Given the above setup, the **search-RLWE problem** is then, for a fixed secret key $\mathbf{s}$, we are given many samples of the form

$$(\mathbf{a}, \mathbf{b}) \quad \text{with } \mathbf{a} \leftarrow R_q, \mathbf{e} \leftarrow \chi, \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in R_q$$

where $\mathbf{a}$ is a random element in R_q, $\mathbf{e}$ distributed according to χ and $\mathbf{b}$ computed as above, to recover the secret key $\mathbf{s}$. Note that each sample requires a new $\mathbf{a}$ and $\mathbf{e}$, but that $\mathbf{s}$ is fixed. When the ring dimension n is large enough (and $B > 0$), then this problem is hard to solve. Furthermore, the elements $\mathbf{b}$ computed according to the RLWE-distribution look completely random. Distinguishing between the RLWE-distribution and the uniform distribution on R_q^2 is called the **decision-RLWE problem**.

The plaintext space of the BGV encryption scheme consists of the ring R_p for some smallish prime p. In this project we will mainly use the Fermat prime $p = 2^{16} + 1$. The main primitives of BGV are then as follows: for given system parameters R, q, χ and p defined above

- **Key Generation**
  - The private key is a polynomial $\mathbf{s} \in R$ with ternary coefficients, i.e. in $\{-1, 0, 1\}$. The private key is kept secret by the user.

– The public key is a pair of polynomials in R_q corresponding to an RLWE-instance (note the ordering is swapped and the minus sign in front of $\mathbf{a}$ and that the error terms are all multiples of p):

$$\mathtt{pk} = [\mathbf{b}, -\mathbf{a}] \quad \text{with } \mathbf{a} \leftarrow R_q, \quad \mathbf{e} \leftarrow \chi \quad \text{and} \quad \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + p\mathbf{e} \in R_q$$

Note that by definition $[\mathtt{pk}[0] + \mathtt{pk}[1] \cdot \mathbf{s}]_q = p\mathbf{e}$ a polynomial with small coefficients all divisible by p.

- **Encryption** To encrypt a message $\mathbf{m} \in R_p$, given the public key $\mathtt{pk}$, sample a polynomial $\mathbf{u}$ with ternary coefficients, and noise polynomials $\mathbf{e}_1, \mathbf{e}_2 \leftarrow \chi$ and compute the ciphertext in R_q^2:

$$\mathtt{ct} = [\mathtt{pk}[0] \cdot \mathbf{u} + p\mathbf{e}_1 + \mathbf{m}, \mathtt{pk}[1] \cdot \mathbf{u} + p\mathbf{e}_2]$$

An encryption of $\mathbf{m}$ will be denoted as $\mathcal{E}(\mathbf{m})$. An encryption as above is called "fresh" since no operations have been executed on the ciphertext. Note that by construction, a ciphertext satisfies

$$\mathtt{ct}[0] + \mathtt{ct}[1] \cdot \mathbf{s} = \mathbf{m} + p\mathbf{e}' \in R_q$$

for some small error term $\mathbf{e}'$. The term $p\mathbf{e}'$ is called the noise term of $\mathtt{ct}$.

- **Decryption** To decrypt a ciphertext $\mathtt{ct}$ of length k (typically $k = 2$), given the private key $\mathbf{s}$, compute the partial decryption, i.e. centered reduction

$$\mathbf{r} = [\sum_{i=0}^{k-1} \mathtt{ct}[i]\mathbf{s}^i]_q$$

and return the message $\mathbf{r} \bmod p \in R_p$. The decryption of $\mathtt{ct}$ will be denoted as $\mathcal{D}(\mathtt{ct})$. Note that decryption will only work correctly as long as the infinity norm of the partial decryption is strictly smaller than $q/2$.

- **Addition** Given two ciphertexts $\mathtt{ct}_1 = \mathcal{E}(\mathbf{m}_1)$ and $\mathtt{ct}_2 = \mathcal{E}(\mathbf{m}_2)$, one can compute an encryption of the sum of the two messages by defining

$$\mathtt{ct}_3 = \mathtt{ct}_1 \oplus \mathtt{ct}_2$$

where addition $\oplus$ is done componentwise.

- **Basic Multiplication** Given two ciphertexts $\mathtt{ct}_1 = \mathcal{E}(\mathbf{m}_1)$ of length k_1 and $\mathtt{ct}_2 = \mathcal{E}(\mathbf{m}_2)$ of length k_2, one can compute an encryption of the product of the two messages by setting

$$\mathtt{ct}_3 = \mathtt{ct}_1 \otimes \mathtt{ct}_2$$

where $\otimes$ denotes multiplication over the ring $R_q[Y]$ and each input ciphertext is considered as a degree $k_i - 1$ polynomial over $R_q[Y]$. In particular, the result $\mathtt{ct}_3$ will consist of $k_1 + k_2 - 1$ elements of R_q. For instance: assume the input are two normal ciphertexts: $\mathtt{ct} = [\mathbf{c}_0, \mathbf{c}_1]$ and $\mathtt{ct}' =$

$[\mathbf{d}_0, \mathbf{d}_1]$, then basic multiplication would give $[\mathbf{c}_0 \cdot \mathbf{d}_0, \mathbf{c}_0 \cdot \mathbf{d}_1 + \mathbf{c}_1 \cdot \mathbf{d}_0, \mathbf{c}_1 \cdot \mathbf{d}_1]$, so consisting of 3 elements in R_q. Note that it is easy to see that this would decrypt to the product of the messages: indeed, by definition we have $\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} = \mathbf{m}_1 + p\mathbf{e}_1$ and $\mathbf{d}_0 + \mathbf{d}_1 \cdot \mathbf{s} = \mathbf{m}_2 + p\mathbf{e}_2$, so clearly

$$(\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}) \cdot (\mathbf{d}_0 + \mathbf{d}_1 \cdot \mathbf{s}) = \mathbf{m}_1\mathbf{m}_2 + p\mathbf{e}'$$

for some $\mathbf{e}'$.

Although the above encryption scheme works and allows to add and multiply ciphertexts together, its homomorphic capabilities are rather limited (See Task 1 to see this in practice). There are two compounding factors:

- The noise term after each operation has grown, and especially so after multiplication.
- The result of the basic multiplication of two regular ciphertexts does not look like a regular ciphertext in that it has 3 components instead of 2 for a fresh ciphertext. So we will need to find a method to transform such 3-component ciphertexts back into their regular 2-component shape.

Modulus switching The first problem can be addressed by modulus switching: it takes as input a ciphertext wrt. some ciphertext modulus q and returns a valid ciphertext wrt. a different (typically smaller) modulus q'. For simplicity, in this project, we will assume that the ciphertext modulus is of the form q_b^ℓ for some fixed base modulus q_b and we want to mod switch over q_b once, so going from q_b^ℓ to $q_b^{\ell-1}$. The ciphertext modulus will therefore vary between q_b and q_b^L for some maximum L which we call `max_level` depending on how many switches have been performed. A ciphertext that is defined modulo q_b^ℓ is said to be at level ℓ.

Recall that a level ℓ ciphertext satisfies

$$\mathtt{ct}[0] + \mathtt{ct}[1] \cdot \mathbf{s} = \mathbf{m} + p\mathbf{e} \bmod q_b^\ell$$

with $p\mathbf{e}$ the noise term. Since we want to end up with a ciphertext modulo $q_b^{\ell-1}$, we could simply try to scale everything over q_b and "formally" obtain

$$\frac{\mathtt{ct}[0]}{q_b} + \frac{\mathtt{ct}[1]}{q_b} \cdot \mathbf{s} = \frac{\mathbf{m}}{q_b} + p\frac{\mathbf{e}}{q_b} \bmod q_b^{\ell-1}.$$

There are 2 main issues with the above: first, in general the coefficients of $\mathtt{ct}[0]$ and $\mathtt{ct}[1]$ will not be divisible by q_b and secondly, the message $\mathbf{m}$ gets multiplied with the inverse of $q_b \bmod p$. The second problem is easy to solve by taking $q_b \equiv 1 \bmod p$. The first needs a bit more work: note that we can always add smallish multiples of p to any component of the ciphertext and its decryption would not change; indeed, the resulting difference in decryption will also be a

multiple of p and just ends up in the noise term. To make the ciphertext parts divisible by q_b and not change the underlying plaintext it thus suffices to first add

$$\delta = [p[-\mathtt{ct}[0]/p]_{q_b}, p[-\mathtt{ct}[1]/p]_{q_b}]$$

to the ciphertext and then have an exact division by q_b. Verify that indeed both components of the sum of $\mathtt{ct} + \delta$ are 0 mod q_b.

A ciphertext will from now on consist of a Magma Tuple, consisting of 2 elements: ((ct0,ct1),l)

- The second element gives the current level ℓ of the ciphertext. At the start this will always be `max_level` and during computations it will go down depending on the number of modulus switches.
- The first element then defines the actual ciphertext as above, so a sequence of k polynomials defined modulo q_b^ℓ where ℓ is the current level and q_b the base modulus defined in the system parameters.

Key switching Key switching allows to transform a ciphertext encrypted under one public key into a valid ciphertext encrypted under a 2nd public key. As an example, recall that after the basic multiply we have a 3-term ciphertext that satisfies:

$$\mathtt{ct}[0] + \mathtt{ct}[1]\mathbf{s} + \mathtt{ct}[2]\mathbf{s}^2 = \mathbf{m} + p\mathbf{e} \bmod q$$

and we really would like to replace the term $\mathtt{ct}[2]\mathbf{s}^2$ by a normal looking ciphertext for the private key $\mathbf{s}$, i.e. we would like to find $\mathbf{d}_0, \mathbf{d}_1$ such that

$$\mathbf{d}_0 + \mathbf{d}_1\mathbf{s} = \mathtt{ct}[2]\mathbf{s}^2 + p\mathbf{e}' \bmod q$$

for some small noise polynomial $\mathbf{e}'$. This would then allow us to replace the 3-term ciphertext by an equivalent 2-term ciphertext $[\mathtt{ct}[0] + \mathbf{d}_0, \mathtt{ct}[1] + \mathbf{d}_1]$. The basic idea is to give out an "encryption" of $\mathbf{s}^2$ under the key $\mathbf{s}$, i.e. provide the ciphertext $\mathtt{ksk} = \mathcal{E}(\mathbf{s}^2)$. Since by definition of encryption we have

$$\mathtt{ksk}[0] + \mathtt{ksk}[1]\mathbf{s} = \mathbf{s}^2 + p\mathbf{e}'' \bmod q$$

for some small $\mathbf{e}''$, we would almost get what we want by multiplying by $\mathtt{ct}[2]$. Unfortunately, $\mathtt{ct}[2]$ just looks like a random polynomial with coefficients modulo q, which would destroy the smallness of the resulting noise term $\mathtt{ct}[2]\mathbf{e}''$. To make this idea work, we simply have to chop $\mathtt{ct}[2]$ into smaller pieces, and work with each piece individually. More in detail, recall that the maximum ciphertext modulus is of the form q_b^L, so we could just write

$$\mathtt{ct}[2] = \sum_{i=0}^{T} \mathtt{ct}[2]^{(i)} q_b^{wi}$$

where w is some word size, and $T = \lceil (L-1)/w \rceil$ and each "slice" $\texttt{ct}[2]^{(i)}$ has infinity norm smaller than q_b^w. The key switching key would then consist of "encryptions" for each i, i.e.

$$\texttt{ksk} = [\mathcal{E}(q_b^{wi}\mathbf{s}^2)]_{i=0}^{T}\,.$$

Note that `ksk` consists of $T+1$ "encryptions" that need to be combined with the slices of $\texttt{ct}[2]$, to finally derive the equivalent ciphertext

$$[\texttt{ct}[0], \texttt{ct}[1]] + \sum_{i=0}^{T} \texttt{ct}[2]^{(i)}\texttt{ksk}[i]\,.$$

Note that the sum runs over pairs of polynomials which get added to the pair $[\texttt{ct}[0], \texttt{ct}[1]]$.

In the project we will simply use $w = 1$ in the implementation.

2.1 Basics of the BGV cryptosystem

Task 1 [5 marks]

Complete the BGV implementation that was started in the file `basicBGV.m`. We also provide a test file `testBGV.m` that contains some basic checks to see if your implementation is behaving as it should. It is instructive to look at the partial implementation to see what a ciphertext looks like. Note that although we can use quotient rings in Magma to define R_q directly, we decided to work in $\mathbb{Z}[x]$ and reduce mod $f(x)$ and q manually whenever needed. The reason for this approach is that the modulus attached to a ciphertext changes during its lifetime, so instead of creating many rings R_q for all different q we just use elements in $\mathbb{Z}[x]$ and remember the level of a given ciphertext.

There are two parameter sets that we will use: a toy parameter set and then a standard parameter set. The main difference is the degree of the polynomial $f(x)$ and the `max_level`. You can select which set to use by defining `toy_set := false` for the standard parameter set. For development and debugging it is recommended to stick with the toy parameter set since it is faster. Once you are confident things work. you can use the standard parameter set.

The file already contains the following functions to get you started:

- `BGVKeyGen() -> sk::RngUPolElt, pk::SeqEnum` which generates a secret key `sk` and public key `pk`.
- `BGVEncrypt(m::RngUPolElt, pk::SeqEnum) -> c::Tup` which takes in a message, i.e. a polynomial in $\mathbb{Z}[x]$ with coefficients smaller than p and returns a ciphertext, where the first element of the tuple represents the actual encryption, and the second element the level of the ciphertext.

- `BGVDecrypt(c::Tup, sk::RngUPolElt) -> m::RngUPolElt` decrypts a ciphertext given the private key `sk` and returns a polynomial representing the message.
- `BGVModSwitch(c::Tup, t::RngIntElt) -> cc::Tup` which takes in a ciphertext c at given level $\ell = c[1]$ and mod switches down over t levels, resulting in a ciphertext on level $\ell - t$. If $t \geq \ell - 1$ or $t < 1$ it throws an error.
- `BGVNoiseBound(c::Tup, sk::RngUPolElt) -> bound::FldReElt` which returns the $\log_2$ of the infinity norm of the partial decryption of the ciphertext `ct` given by
$$\mathbf{r} = [\sum_{i=0}^{k-1} \mathtt{ct}[i]\mathbf{s}^i]_{q_b^\ell}$$
with $\ell = c[1]$ the level of the ciphertext and k the number of components in $\mathtt{ct} = c[0]$.
- `BGVKeySwitchingKeyGen(sk2::RngUPolElt, sk1::RngUPolElt) -> ksk::SeqEnum` generates the key switching key, i.e. contains "encryptions" of $q_b^i \mathtt{sk}_2$ for $i = 0, \ldots, \mathtt{max\_level} - 1$.

Complete the BGV cryptosystem by implementing the following APIs:

1.a `BGVAdd(c1::Tup, c2::Tup) -> cc::Tup` which takes in 2 ciphertexts (not necessarily at the same level) and outputs the sum of the two ciphertexts at a level which is the mimimum of the two input levels.

1.b `BGVBasicMul(c1::Tup, c2::Tup) -> cc::Tup` which takes in 2 ciphertexts (not necessarily at the same level) and outputs the basic multiplication of the two ciphertexts at a level which is the mimimum of the two input levels. Note that the ciphertext will contain $k_1 + k_2 - 1$ elements over $R_{q_b^\ell}$ where ℓ is the output level and k_1 and k_2 the number of elements in the input.

1.c `BGVKeySwitch(g::RngUPolElt, ell::RngIntElt, ksk::SeqEnum) -> cc::Tup` which takes in a single part g of a ciphertext (think `ct`[2] above) at level ℓ, so defined modulo q_b^ℓ, and key switches this using the key switching key given in `ksk` resulting in a 2-term "ciphertext". Note that the level of the resulting ciphertext will be `max_level` since this is inherited from the key switching key.

1.d `BGVMul(c1::Tup, c2::Tup, ksk:SeqEnum) -> cc::Tup` which takes in 2 standard ciphertexts (not necessarily at the same level) and outputs the multiplication of the two ciphertexts at a level which is the mimimum of the two input levels as a standard ciphertext. In particular, this function combines `BGVBasicMul` with `BGVKeySwitch`.

1.e Using the function `BGVNoiseBound`, make a plot of the noises for the following scenario's: Using the standard parameter set given in `basicBGV.m`, generate a fresh encryption `ct` of a random message $\mathbf{m} \in R_p$ and compute

- $\texttt{ct}^k$ using `BGVBasicMul` for $k = 1 \ldots 16$. Note that in this case the ciphertext size will grow with each multiply since there is no key switching being done.
- $\texttt{ct}^k$ using `BGVMul` for $k = 1 \ldots 16$.
- $\texttt{ct}^k$ using `BGVMul` followed each time by a single `BGVModSwitch` for $k = 1 \ldots 10$. The reason this stops at 10 is due to `max_level` in the standard parameter set.
- $\texttt{ct}^{2^k}$ (yes that is a power of 2^k in the exponent, this is not a typo) using `BGVMul` followed each time by a single `BGVModSwitch` for $k = 1 \ldots 10$ by **repeated squaring**.

Explain the noise growth in the different scenario's, give the maximum number of multiplications each strategy can handle before decryption error and explain what the best strategy is to compute a product of the form

$$\prod_{i=0}^{n-1} \texttt{ct}_i$$

for n ciphertexts $\texttt{ct}_i$. For the standard parameter set, what is the maximum number of fresh ciphertexts that can be multiplied together before a decryption failure occurs?

3 CRT and SIMD arithmetic

In the previous section, you implemented a basic version of the BGV cryptosystem. The plaintext space consists of the ring $R_p = \mathbb{Z}[x]/(p, f(x))$ for some smallish prime p and $f(x) = x^n + 1$ with $n = 2^k$. In applications however, one is typically interested in operations on elements of $\mathbb{F}_p$ directly, instead of on such polynomials. A trivial method to do so, is to simply restrict to polynomials with only the constant coefficient non-zero, but this is wasteful since a general plaintext element can contain n such elements as its coefficients. Note that adding two plaintext polynomials can be interpreted as adding the corresponding vectors of coefficients, but that this not hold when considering multiplication (the result looks more like a convolution of the two vectors).

Some parameter sets allow to manipulate arrays of elements in $\mathbb{F}_p$ both for addition as well as multiplication. This is called SIMD arithmetic for Single Instruction Multiple Data and allows to execute a set of operations on an array of elements in parallel. Recall that the Chinese Remainder Theorem gives the following ring isomorphism:

$$R_p = \mathbb{Z}[x]/(p, f(x)) \cong \oplus_{i=1}^{d} \mathbb{Z}[x]/(p, f_i(x))$$

where the $f_i(x)$ are the irreducible factors of $f(x) \bmod p$. Since the above is a ring isomorphism, we can use the trivial method from above (only use polynomials with non-zero constant) for each of the factors, and thus immediately obtain a method to manipulate arrays of d elements in $\mathbb{F}_p$ in parallel.

The maximal packing is achieved for polynomials $f(x)$ that split completely modulo p into different factors, i.e. each $f_i(x)$ has degree 1 or equivalently, that $f(x)$ has n roots over $\mathbb{F}_p$. In the example parameters, we have $p = 2^{16} + 1$ so $\mathbb{F}_p$ contains all roots of unity of order 2^{16}. This also implies that polynomials of the form $f(x) = x^n + 1$ with $n = 2^k$ will split completely for k up to 15, and thus definitely for the two parameter sets in the project. This shows that using CRT we can really compute element wise on arrays of length n over $\mathbb{F}_p$ using a 3 step process:

- encode array $[m_1, \ldots, m_d] \in \mathbb{F}_p^d$ into a single element of $\mathbf{m} \in R_p$ using CRT, given the d factors f_i of $f(x) \bmod p$
- perform ring operations on R_p
- decode element $\mathbf{m} \in R_p$ to an array of d elements using the inverse CRT, given the same d factors f_i of $f(x) \bmod p$

Task 2 [1 marks]

Implement the encoding and decoding function above that takes as input an array of d elements in $\mathbb{F}_p$ and an array containing the d factors of $f(x) \bmod p$. Note that CRT really depends on the order of the factors, so it is important to pass the same sequence to both encoding as well as decoding to obtain a consistent result.

2.a `BGVEncode(ms::SeqEnum, fs::SeqEnum) -> m::RngUPolElt` which takes as input a sequence `ms` of d elements in $\mathbb{F}_p$, a sequence of d elements in $\mathbb{F}_p[x]$ which are the factors of $f(x) \bmod p$ and returns the result of applying CRT as an element in R_p, represented by a polynomial in $\mathbb{F}_p[x]$.

2.b `BGVDecode(m::RngUPolElt, fs::SeqEnum) -> ms::SeqEnum` which decodes the output of the previous function.

Make sure to test encoding and decoding, by confirming that they are indeed ring isomorphisms, i.e. that adding / multiplying encoded arrays as elements of the ring R_p indeed corresponds to pointwise addition and multiplication on these arrays. Note that this is just a test on the plaintext space, so there is no need to encrypt/decrypt anything, it is only a test of the encoding. Of course, since BGV gives us operations on the ring R_p, the combination with the encoding allows us to manipulate encryptions of arrays over $\mathbb{F}_p$.

Bonus task - optional for 1 mark

Although the above encoding already allows us to elementwise add and multiply arrays over $\mathbb{F}_p$, there are still a few shortcomings: first, we need to remember the ordering of the factors $f_i(x)$, and secondly, we are currently unable to permute the elements in the array. Both issues can be solved using automorphisms of the ring R. Since in our case $f(x) = x^n + 1$ equals the $2n$-th cyclotomic polynomial, it is well known that the group of automorphisms on R is given by:

$$\tau_j : R \to R : a(x) \to a(x^j) \quad \text{for} \quad j \in \mathbb{Z}_{2n}^{\times},$$

in particular, $0 < j < 2n$ and $\gcd(j, 2n) = 1$. Furthermore, the structure and generators of $\mathbb{Z}_{2n}^{\times}$ are well known, namely, the group is isomorpic to $\langle -1 \rangle \times \langle 5 \rangle, \cdot$ where the first factor is generated by -1 of order 2 and the second factor by 5 of order $n/2$. In particular, any element of $\mathbb{Z}_{2n}^{\times}$ can be written as $(-1)^s 5^t$ with $0 \leq s < 2$ and $0 \leq t < n/2$.

Task 3 [1 marks] - bonus task - optional

3.a In the case where $f(x)$ splits completely, choose a single root α and set $f_1(x) = x - \alpha$. Using the above decomposition of $\mathbb{Z}_{2n}^{\times}$, derive a well defined order for the other factors $f_i(x)$, such that the n elements in $\mathbb{F}_p$ are arranged in a $2 \times n/2$ matrix, where the factor $f_1(x)$ corresponds to the element in position $(0, 0)$ and the automorphism corresponding to -1 swaps the rows of the matrix, and the automorphism corresponding to 5 causes a circular shift **to the right** over the columns.

3.b Using the above ordering, implement a matrix encoding function that takes as input a $2 \times (n/2)$ matrix of elements in $\mathbb{F}_p$, the starting root α and returns an element in R_p via the CRT and the constructed ordering above:
`BGVMatrixEncode(mat::ModMatFldElt, alpha::FldFinElt) -> m::RngUPolElt`

3.c `BGVMatrixDecode(m::RngUPolElt, alpha::FldFinElt) -> mat::ModMatFldElt`
which decodes the output of the previous function.

Make sure to test that the above are each other's inverse, but also that encoding a matrix, then applying the automorphism corresponding to -1 and decoding indeed corresponds to swapping the rows, and similarly, applying the automorphism corresponding to 5 corresponds to a circular shift of the columns. Again note that all these operations are done on the plaintext space; there is no need to encrypt/decrypt anything. It is also possible to apply automorphisms τ_j to a ciphertext, which would have the same result on the underlying plaintext, but you would also need to key switch from $\tau_j(\mathbf{s})$ to $\mathbf{s}$.

4 Attacking BGV using lattice reduction

The public key of the BGV scheme really corresponds to a single sample of RLWE. Since we really have

$$\texttt{pk}[0] = -\texttt{pk}[1] \cdot \mathbf{s} + p\mathbf{e}$$

in the ring R_q, we can derive n LWE samples from this 1 RLWE sample. In the fully general RLWE setting, there are no restrictions on $\mathbf{s}$ and giving out a single sample does not fully determine $\mathbf{s}$. In fact, it is easy to see that in this case any choice for $\mathbf{e}$ would result in a possible solution for $\mathbf{s} = -(\texttt{pk}[0] - p\mathbf{e})/\texttt{pk}[1]$ as long as $\texttt{pk}[1]$ is invertible in R_q.

Furthermore, the standard attack on LWE (and also on RLWE after expanding single samples into n LWE samples) proceeds by assembling all m samples in an $m \times n$ matrix A and an $m \times 1$ matrix B (both are public) such that

$$A \cdot S + E = B \bmod q \tag{1}$$

where S is the $n \times 1$ matrix containing the coefficients of the secret key and E the $m \times 1$ matrix containing the individual error terms. Both E and S are secret. If we then consider the lattice

$$L := \{z \in \mathbb{Z}^m \mid \exists x \in \mathbb{Z}^n \text{ such that } \ z = Ax \bmod q\}$$

then we can look for the closest vector to B inside the lattice L, i.e. solve a CVP. If the norm of E is not too large, this vector will be exactly $A \cdot S \bmod q$, and then we can recover S using simple Gaussian elimination.

The problem with the above approach is that it does **not** apply to our setting, since we only have a **single** RLWE sample (corresponding to the public key), but with ternary secret key (which guarantees a unique solution in most cases, see task below). Since the secret itself is ternary, its norm is small and independent of q, so maybe we can try to construct a lattice such that the secret really corresponds to (part of) a short vector directly.

Looking at the first row of Equation (1), we can rewrite it as

$$A[1:] \cdot S + pE[1] - B[1] = 0 \bmod q\,. \tag{2}$$

Furthermore, we know S is ternary, $E[1]$ is unknown but small since sampled from $[-B, B]$ and $B[1]$ is given (the B denotes the matrix here, not the noise bound). Note that we have taken out p explicitly since only $E[1]$ is guaranteed to be very small, in contrast to $pE[1]$ which is quite a bit bigger.

Task 4 [5 marks]

4.a Give a simple counting argument to derive a bound involving n, B and q such that a single RLWE sample (e.g. the public key) uniquely determines the underlying ternary key $\mathbf{s}$. Note that we assume the errors are sampled from the uniform distribution on $[-B, B]$.

4.b Describe how from the single equation in (2) you can derive an $(n+2)$-dimensional lattice that contains a short-ish vector, part of which gives the secret key immediately. Describe how you would compute a basis for this lattice, give a formula for its volume, derive from this a formula for the expected length of the shortest vector, and conclude for which parameters you expect the vector of interest (i.e. the one containing the secret key) to really be the shortest vector in this lattice and can thus be recovered as such. What happens when you do not take the factor p out of the error term?

4.c Implement your lattice attack from 4.b as a Magma function `BGVLatticeAttack(pk::SeqEnum, ell::RngIntElt) -> sk::RngUPolElt` that takes as input a single public key, a level `ell` which defines $q = q_b^\ell$ modulo which q the public key has to be considered and uses this to setup the lattice from 4.b, and tries to recover the secret key by finding a short vector in said lattice. Although Magma has a `ShortestVectors` command, it is much more efficient to use BKZ-reduction with block size 20 say. This gives a BKZ reduced basis, and you can use the first vector as one of the (supposedly) shortest vectors in the lattice. Hint: note that if you find a short vector, the negative of that vector will have the same norm.

4.d Use your implementation from 4.c on the toy parameter set, i.e. $N = 64$, but for increasing moduli sizes, i.e. for $q = q_b^k$ with $k = 1, \ldots, 8$. Determine for which k your algorithm really recovers the secret key. Does this align with the bound you derived in 4.b?

4.e **(Bonus task - optional for 1 mark)** Describe how you would extend the attack to take more than 1 equation into account. What is the main difference compared to using a single equation?

5 Decryption failures and an active attack

One of the main problems of basic cryptosystems such as the BGV cryptosystem as described in Section 2, is that there are no restrictions on what is seen as a "valid" ciphertext, i.e. a ciphertext that was really obtained as an encryption of a message.

Such cryptosystems can be attacked by an active attack: in this case, an attacker can ask a user to decrypt anything that looks like a ciphertext and receive what the user thinks is the decryption of this ciphertext. The attacker therefore exploits the user as a "Decryption Oracle". Although it might look strange that you give an attacker the power to ask a user to decrypt anything the attacker wants, this functionality is often available in practice in authentication protocols, i.e. in a protocol where the user has to prove he knows the secret key.

Task 5 [5 marks]

5.a Show that there is a trivial key recovery attack (at least as long as $p > 2$) that recovers the full private key (which is ternary) in a **single** decryption query. Hint: look at the BGV decryption equation and make a clever choice for `ct`[0] and `ct`[1]. Implement this attack as `BGVTrivialKeyRecovery(sk::RngUPolElt) -> s::RngUPolElt` which takes as input a secret key (the only reason for this is so you can call `BGVDecrypt` to mimic the decryption oracle, you should not use it for anything else), constructs a special ciphertext, and recovers the private key using a single decryption query. How would you adjust the attack for $p = 2$ and how many queries do you need in this case?

5.b The attack in 5.a can be easily avoided by rejecting certain special ciphertexts. However, the fact that the scheme is homomorphic combined with the presence of decryption failures when the noise in a ciphertext reaches a certain bound, implies that a very general attack is still able to recover the secret key, even though the input ciphertext now really does look like a valid ciphertext. Given a valid ciphertext `ct`, devise a method that derives a related ciphertext `ct'` which causes a decryption failure in a chosen coefficient of the plaintext. How would you create a borderline case: a valid ciphertext such that a minimal change, i.e. adding 1 in a single coefficient, causes a decryption failure? When you look at the decryption equation, what information do you learn from such borderline case? How many queries does it take to create such borderline ciphertext for a single coefficient? Can you create such ciphertext for all coefficients at the same time?

5.c Implement the attack from 5.b in a function called `BGVActiveAttack(pk::SeqEnum, sk::RngUPolElt) -> s::RngUPolElt, nbq::RngIntElt` that starts from a public key, derives by repeated querying the decryption oracle, a borderline ciphertext (for all coefficients if possible) and uses this borderline ciphertext to recover the secret key. The function also returns the total number `nbq` of queries to the decryption oracle. Recall that the public key can be interpreted as a valid ciphertext encrypting the message 0. The secret key `sk` is only passed as input so you can mimic the decryption oracle by calling `BGVDecrypt`. You should not use it in any other place, also not to verify correctness of the result of your attack (you should think of a different method to verify if a candidate key is really the correct private key).

6 NTTs

Recall that the DFT (Discrete Fourier Transform) takes as input a primitive N-th root of unity $\omega \in \mathcal{R}$ in some ring $\mathcal{R}$ and a polynomial $g \in \mathcal{R}[x]$ and

computes

$$DFT_\omega : g(x) \to [g(w^i)]_{i=0}^{N-1} .$$

When the ring $\mathcal{R}$ is finite, for instance $\mathbb{Z}/(q\mathbb{Z})$, the DFT is often called the Number Theoretic Transform (NTT). We can also define the NTT on an input array $[g_0, \ldots, g_{N-1}]$ where we then interpret this array as the coefficients of the polynomial $g(x)$ and apply the NTT to $g(x)$. For $N = 2^d$, the NTT can be computed efficiently using the analogon of the FFT algorithm (the only difference being the root of unity is now in a finite ring).

Furthermore, as explained in the lectures, the NTT can be used to speed up multiplication of polynomials: given two polynomials $a(x), b(x) \in \mathcal{R}[x]$ of degree $< N/2$ we can recover their product as:

$$c(x) = a(x)b(x) = INTT_\omega(NTT_\omega(a(x)) \cdot NTT_\omega(b(x)))$$

where $\cdot$ denotes pointwise multiplication of vectors and INTT the inverse NTT. Since the ω^i are precisely the roots of the polynomial $x^N - 1$ (note there really is a $-$ there), it is also not hard to see that the NTT is consistent with arithmetic in the polynomial ring $\mathcal{R}[x]/(x^N - 1)$, i.e. we get reduction modulo $(x^N - 1)$ for free.

Task 6 [4 marks]

6.a Implement a basic recursive version of the NTT and the INTT for arrays defined over some ring $\mathcal{R}$ containing an N-th root of unity. We will take $\mathcal{R} = \mathbb{Z}/(q\mathbb{Z})$ where q is such that an N-th root of unity exists. In the examples, we will use $q = q_b^8$. Note that for the INTT you can just call the NTT itself as subroutine.

- `RecNTT(a::SeqEnum, omega::RngIntResElt, N::RngIntElt) -> b::SeqEnum`
- `RecINTT(a::SeqEnum, omega::RngIntResElt, N::RngIntElt) -> b::SeqEnum`

which takes as input a length N array of some ring $\mathcal{R}$, a primitive N-th root of unity $\omega \in \mathcal{R}$ and returns the N-point NTT (resp. INTT). Note that we want the following identity:

$$\texttt{INTT}(\texttt{NTT}(\texttt{a}, \texttt{omega}, \texttt{N}), \texttt{omega}, \texttt{N}) = \texttt{a} ,$$

in particular, calling the functions with the same ω gives the inverse operation.

6.b Implement a basic schoolbook multiplication routine to multiply two polynomials of degree $< N/2$ over some ring $\mathcal{R}$

- `SchoolBookMult(a::RngUPolElt, b::RngUPolElt) -> c::RngUPolElt`; here you are not allowed to use the built-in polynomial arithmetic; you are only allowed to use operations on elements in the ring $\mathcal{R}$.

- `PrimitiveNthRoot(ell::RngIntElt,N::RngIntElt) -> omega::RngIntResElt` which takes as input a level ℓ and computes a primitive N-th root of unity modulo q_b^ℓ. This function will only be called for $N = 2^k$ with $k = 1, \ldots, 15$. It should return an element of the ring $\mathbb{Z}/(q_b^\ell\mathbb{Z})$.
- `FastMult(a::RngUPolElt, b::RngUPolElt, omega::RngIntResElt, N::RngIntElt) -> c::RngUPolElt` a fast version using the NTT where you pass in two polynomials of degree $< N/2$, an N-th root of unity ω in the coefficient ring $\mathcal{R}$.

Using the ring $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ with $q = q_b^8$, compare the efficiency of both methods to multiply two polynomials of degree strictly less than 2^k for $k = 2, \ldots, 13$.

6.c The NTT allows to work in the polynomial ring $\mathbb{Z}[x]$ modulo $(x^N - 1)$ (and modulo q). However, in the BGV cryptosystem we really want to work modulo $(x^n + 1)$ with $n = 2^k$, so there is $+1$ instead of a -1. Can you think of a method to multiply two polynomials modulo $(x^n + 1)$ by only using an NTT of length n?

6.d In task 2.a and 2.b you implemented the CRT method to encode and decode vectors of elements in $\mathbb{F}_p$. Assuming that $f(x) = x^n + 1$ splits completely modulo p, can you find a faster method to implement 2.a and 2.b?

References

[1] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.