Implementation of LWE-based encryption

Report by:
Yevhen Perehuda – 1190035053

University of Luxembourg

1 July, 2022

Abstract

Homomorphic encryption is known among cryptographers as the "Swiss Army Knife" due to the variety of applications it may be used in, from cloud storage to non-interactive zero-knowledge proof (NIZK). We implemented the elementary version of "somewhat homomorphic" BV11 scheme[1], which is example of second generation of Fully Homomorphic Encryption(FHE), in Python using the Sage library. The second generation differs from the previous schemes by the appearance of relinearization and modulus reduction (as alternative to bootstrapping). Therefore, in addition to the main task, we decided to add another modulus reduction, as an essential part of the algorithm. Also in this paper, we explain why the scheme works based implemented code.

1 Basic LWE encryption

Firstly, we choose our parameters q, n and m. As q is order of the field it should be prime. Based on that every time of algorithm we generate q as cryptological "safe prime".

For a proof of security and simplicity we take dimensions parameters as $n = m$ based on the leftover hash lemma[1].

Additional constraint is that q is recommended to be sufficiently larger than n.

1.1 Key generation

For simplicity we take $\vec{s} \in \{0,1\}^n$ from binomial distribution, where $s_1 = 1$. Analogically, we choose vector of noise $\vec{e}$, based also on parameter k. Based on that we can find matrix A$\in {\mathbb{F}_q}^{m \times n}$ consisting of public-key $\vec{a_i}$ such that:

$$\mathbf{A} \cdot \vec{s} = \vec{e} \mod q \tag{1}$$

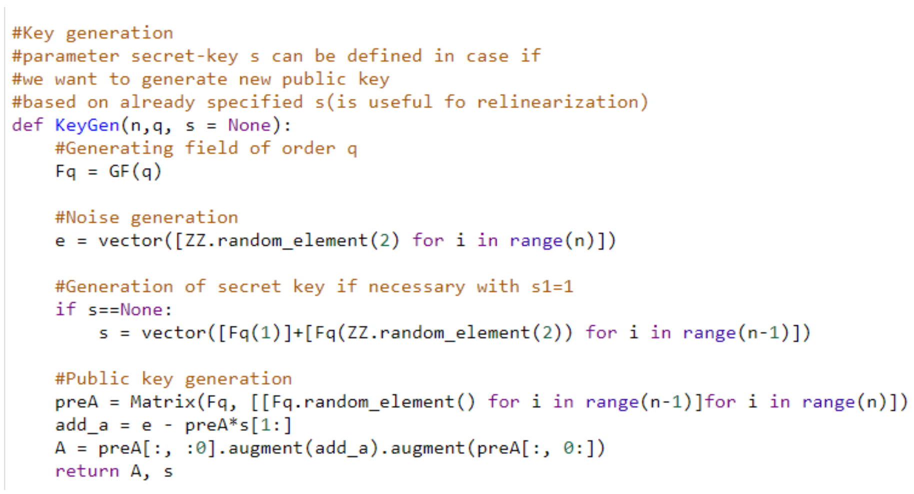
there are many possible matrix, in theory, values of it should be distributed uniformly. For simplicity we chose for each public-key $\vec{a_i} \in {\mathbb{F}_q}^n$ randomly $n-1$ elements and the last one is calculate to satisfy Equation 4. Code with keys generation is shown on Figure 1.

1.2 Encryption and decryption

Encryption is quite similar to other homomorphic schemes. To encrypt message $m \in \{0,1\}$ **we use matrix of public-keys A (like in DGHV scheme[2])**:

$$\vec{c} = (m, 0, \dots, 0) + 2\vec{u} \cdot \mathbf{A} \tag{2}$$

where $\vec{u} \in \{0,1\}^m$(in our case the same as n) randomly chosen for binomial distribution.



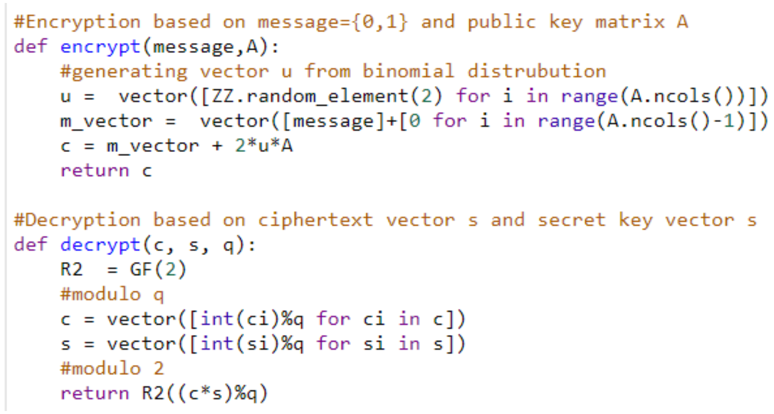
```
#Key generation
#parameter secret-key s can be defined in case if
#we want to generate new public key
#based on already specified s(is useful fo relinearization)
def KeyGen(n,q, s = None):
    #Generating field of order q
    Fq = GF(q)

    #Noise generation
    e = vector([ZZ.random_element(2) for i in range(n)])

    #Generation of secret key if necessary with s1=1
    if s==None:
        s = vector([Fq(1)]+[Fq(ZZ.random_element(2)) for i in range(n-1)])

    #Public key generation
    preA = Matrix(Fq, [[Fq.random_element() for i in range(n-1)]for i in range(n)])
    add_a = e - preA*s[1:]
    A = preA[:, :0].augment(add_a).augment(preA[:, 0:])
    return A, s
```

Figure 1: Key generation code

So, for decryption we calculate product of $\vec{c}$ and $\vec{s}$ and apply modulo q and modulo 2:

$$\begin{gathered}
\langle\vec{c},\vec{s}\rangle = (m,0,\ldots,0)\cdot\vec{s} + 2\vec{u}\cdot\mathbf{A}\cdot\vec{s} \equiv \\
(m\cdot 1(because\ s_1 = 1) + 2\vec{u}\cdot\vec{e}) \mod q = \\
(m + 2\langle\vec{u},\vec{e}\rangle) \mod q \equiv \\
m \mod q \mod 2
\end{gathered} \tag{3}$$

To apply modulos q successfully and obtain correct decryption, the product of noise $\vec{e}$ and vector $\vec{u}$ - $\langle\vec{u},\vec{e}\rangle$ **should be less than** $q/4$. However, in our case $\vec{e}$ and $\vec{u}$ are very simple and sparse so we can neglect this constraint in our implementation as we defined q as 16-bits safe prime. Methods which corresponds to this part you can see on Figure 2.



```
#Encryption based on message={0,1} and public key matrix A
def encrypt(message,A):
    #generating vector u from binomial distrubution
    u =  vector([ZZ.random_element(2) for i in range(A.ncols())])
    m_vector =  vector([message]+[0 for i in range(A.ncols()-1)])
    c = m_vector + 2*u*A
    return c

#Decryption based on ciphertext vector s and secret key vector s
def decrypt(c, s, q):
    R2  = GF(2)
    #modulo q
    c = vector([int(ci)%q for ci in c])
    s = vector([int(si)%q for si in s])
    #modulo 2
    return R2((c*s)%q)
```

Figure 2: Encryption and decryption modules

2 Ciphertext multiplication and relinearization

If we will naively multiply two ciphertexts $\vec{c_1}$ and $\vec{c_2}$ we will obtain a new ciphertext vector or in other words polynomial $c(\vec{x}) = c_1(\vec{x}) * c_2(\vec{x})$. For $\vec{s}$, $c(\vec{s}) = m_1 m_2 + 2e' \mod q$. However, **number of coefficients in this case will be n^2 instead of n as before**. So $c(\vec{x})$ is quadratic polynomial. And to solve this problem relinearization is used.

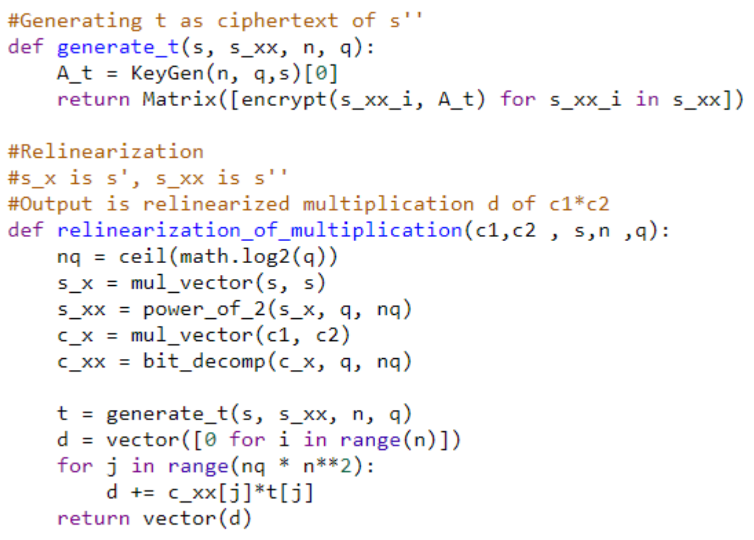
Relinearization is typical method for cryptology. For example, it is used in attack on filtered LFSR(stream cipher). The idea is that every multiplied pair of coefficients we masking as new coefficient(quadratic term).

So for that we generate ciphertext $\vec{t}$ (encryption of powers of 2 of products $\vec{(s')} = (s_i \cdot s_j)_{i,j}$), which consist of polynomials $t_{j,k,l}(\vec{x}) = 2^l * x_j * x_k + L_{j,k,l}(\vec{x})$, where $0 \leq l < n_q$, n_q is size of bits of q and $0 \leq j, k < n$. L is linear term.

This ciphertext $\vec{t}$ helps us take back to n dimension, because after we multiply it by $\vec{c''}$ (which is binary decomposition of $\vec{(c')} = (c_i \cdot c_j)_{i,j}$). This will remove the quadratic terms, using a binary decomposition(we can say that it is some kind memorization of place of different quadratic terms). After only linear terms left, which we aggregate to dimension n. So in the end we have ciphertext with the same dimension.

Another interesting observation is that **during relinearization we can reduce dimension even less from n**.

Code for relinearization can be observed on Figure 3.



```
#Generating t as ciphertext of s''
def generate_t(s, s_xx, n, q):
    A_t = KeyGen(n, q,s)[0]
    return Matrix([encrypt(s_xx_i, A_t) for s_xx_i in s_xx])

#Relinearization
#s_x is s', s_xx is s''
#Output is relinearized multiplication d of c1*c2
def relinearization_of_multiplication(c1,c2 , s,n ,q):
    nq = ceil(math.log2(q))
    s_x = mul_vector(s, s)
    s_xx = power_of_2(s_x, q, nq)
    c_x = mul_vector(c1, c2)
    c_xx = bit_decomp(c_x, q, nq)

    t = generate_t(s, s_xx, n, q)
    d = vector([0 for i in range(n)])
    for j in range(nq * n**2):
        d += c_xx[j]*t[j]
    return vector(d)
```

Figure 3: Relinearization module

3 (Extra)Modulus switching

As we have already working ”somewhat” homomorphic encryption, we decided to implement **modulus reduction** to convert it to FHE. It gives us much shorter ciphertexts and lower decryption complexity(than during bootstrapping), which will enable us to apply the bootstrapping theorem to obtain full homomorphism.

The goal is to allow for evaluator, who does not know the secret key s but instead only knows a bound on its length, can transform a ciphertext c modulo q into a different ciphertext modulo p while preserving correctness. This helps to decrease noise by p/q factor.

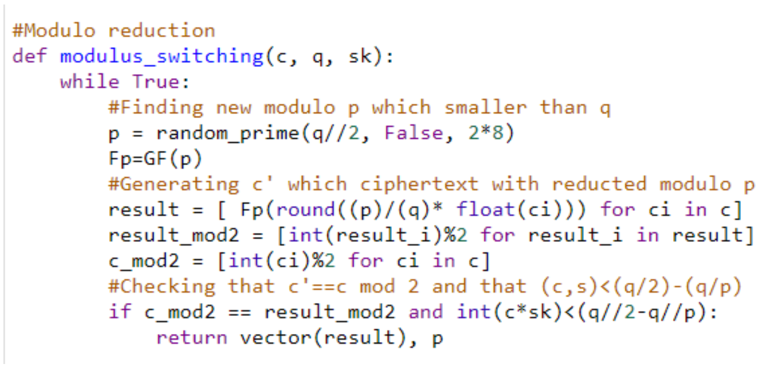
This can be achieve this by multiplying ciphertext $\vec{c}$ by fraction p/q and rounding afterwards. So will get $\vec{c'}$ which is the closest integer vector to $p/q \cdot \vec{c}$. Then if $\vec{c'} = \vec{c} \mod 2$, $\vec{c'}$ is final resulted modulo reduced ciphertext with decreased noise by p/q:

$$|\langle \vec{s}, \vec{c'} \rangle|_p = |\langle \vec{s}, \vec{c} \rangle|_q \mod 2 \tag{4}$$

This approach works because:

- we know that $\langle \vec{s}, \vec{c} \rangle \mod q = \langle \vec{s}, \vec{c} \rangle - kq$ and $\langle \vec{s}, \vec{c'} \rangle \mod p = \langle \vec{s}, \vec{c'} \rangle - kp$.
- As p and q are odd (because they should be order of the field), thus $kq \equiv kp \mod 2$.
- Since $\vec{c'} \equiv \vec{c} \mod 2$ then $\langle \vec{s}, \vec{c} \rangle \equiv \langle \vec{s}, \vec{c'} \rangle \mod 2$
- So, we can define $\langle \vec{s}, \vec{c'} \rangle \mod p = \langle \vec{s}, \vec{c'} - kp \equiv \vec{s}, \vec{c} - kq \mod 2 = \vec{s}, \vec{c} \mod q$

Code for this part is on Figure 4.



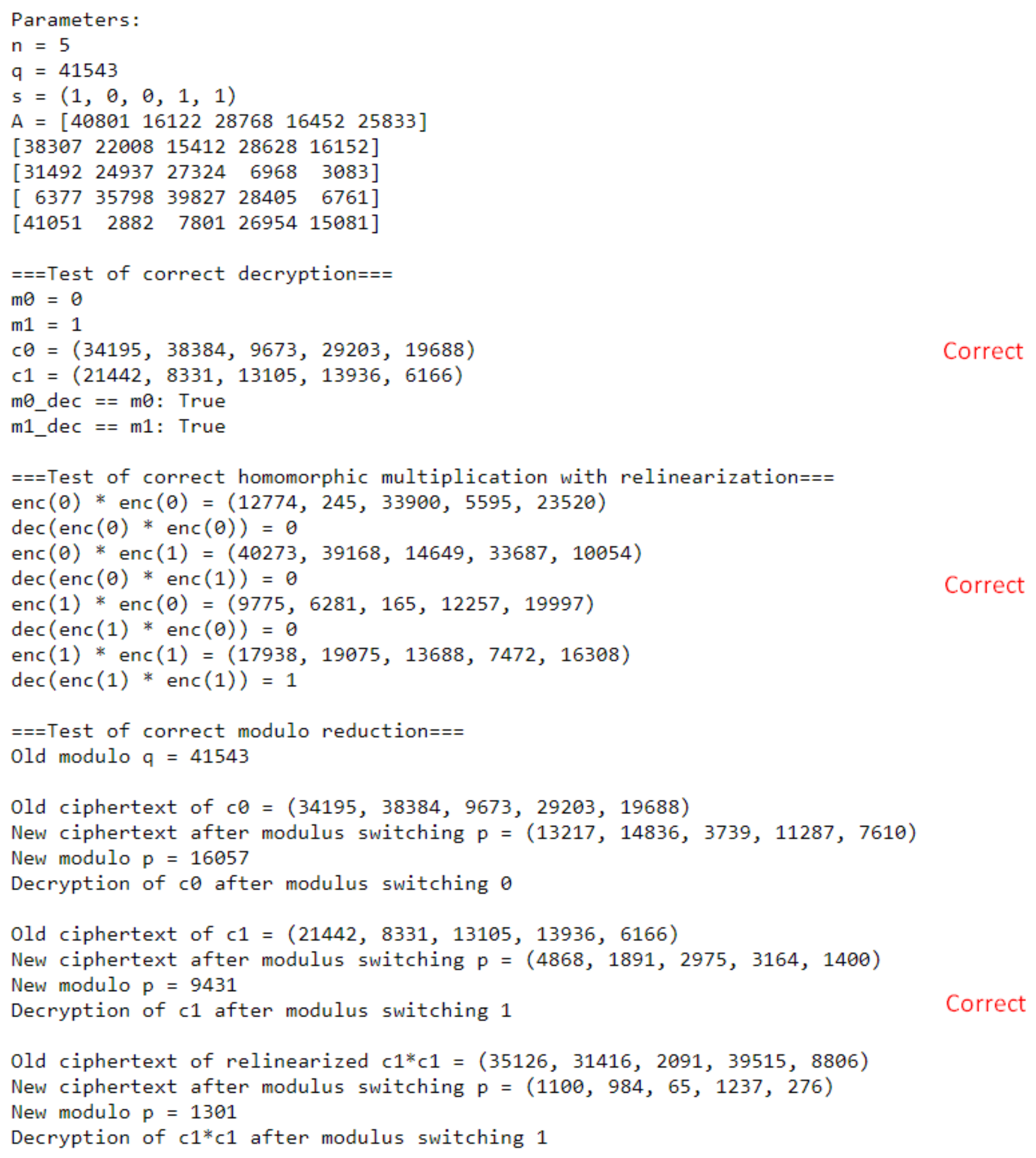
```
#Modulo reduction
def modulus_switching(c, q, sk):
    while True:
        #Finding new modulo p which smaller than q
        p = random_prime(q//2, False, 2*8)
        Fp=GF(p)
        #Generating c' which ciphertext with reducted modulo p
        result = [ Fp(round((p)/(q)* float(ci))) for ci in c]
        result_mod2 = [int(result_i)%2 for result_i in result]
        c_mod2 = [int(ci)%2 for ci in c]
        #Checking that c'==c mod 2 and that (c,s)<(q/2)-(q/p)
        if c_mod2 == result_mod2 and int(c*sk)<(q//2-q//p):
            return vector(result), p
```

Figure 4: Code for module switching

Decryption succeeds as long as the magnitude of the noise stays smaller than q/2. Thus another important constraint is that: $\langle \vec{c}, \vec{s} \rangle < q/2 - (q/p) * \mathcal{L}_1$, where $\mathcal{L}_1(\vec{s})$ is Manhattan norm of $\vec{s}$ (in our case it is negligible, as we took $\vec{s}$ as vector of zeroes and ones).

4 Results

On Figure 5 you can see the results of testing - test of correct decryption and encryption, correct homomorphic multiplication with relinearization, and test of correct homomorphic modulo switching.



```
Parameters:
n = 5
q = 41543
s = (1, 0, 0, 1, 1)
A = [40801 16122 28768 16452 25833]
[38307 22008 15412 28628 16152]
[31492 24937 27324  6968  3083]
[ 6377 35798 39827 28405  6761]
[41051  2882  7801 26954 15081]

===Test of correct decryption===
m0 = 0
m1 = 1
c0 = (34195, 38384, 9673, 29203, 19688)
c1 = (21442, 8331, 13105, 13936, 6166)
m0_dec == m0: True
m1_dec == m1: True

===Test of correct homomorphic multiplication with relinearization===
enc(0) * enc(0) = (12774, 245, 33900, 5595, 23520)
dec(enc(0) * enc(0)) = 0
enc(0) * enc(1) = (40273, 39168, 14649, 33687, 10054)
dec(enc(0) * enc(1)) = 0
enc(1) * enc(0) = (9775, 6281, 165, 12257, 19997)
dec(enc(1) * enc(0)) = 0
enc(1) * enc(1) = (17938, 19075, 13688, 7472, 16308)
dec(enc(1) * enc(1)) = 1

===Test of correct modulo reduction===
Old modulo q = 41543

Old ciphertext of c0 = (34195, 38384, 9673, 29203, 19688)
New ciphertext after modulus switching p = (13217, 14836, 3739, 11287, 7610)
New modulo p = 16057
Decryption of c0 after modulus switching 0

Old ciphertext of c1 = (21442, 8331, 13105, 13936, 6166)
New ciphertext after modulus switching p = (4868, 1891, 2975, 3164, 1400)
New modulo p = 9431
Decryption of c1 after modulus switching 1

Old ciphertext of relinearized c1*c1 = (35126, 31416, 2091, 39515, 8806)
New ciphertext after modulus switching p = (1100, 984, 65, 1237, 276)
New modulo p = 1301
Decryption of c1*c1 after modulus switching 1
```

Figure 5: Result of executed tests

References

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

[2] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 24–43. Springer, 2010.