SOP REPORT

TITLE: STUDY OF STATE-OF-THE-ART POST-QUANTUM CRYPTOSYSTEMS

SUBMITTED TO: Dr. Tejasvi Alladi

AT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE,PILANI
(JAN 2023 - MAY 2023)

By:
Satvik Sinha 2020A7PS0993P
Kashish Mahajan 2020A7PS0995P

INTRODUCTION

With the rapid development and enhancement of quantum computers, it is becoming easier for them to break traditional cryptographic algorithms. Most of the traditional cryptographic algorithms are based on the prime factorisation of integers. Shor's algorithm, which runs on quantum computers can solve this problem in O(logn) time, hence the prime factorisation problem doesn't work with quantum computers. With more development, the traditional algorithms will be prone to quantum attacks. Therefore, post quantum cryptographic schemes must be used in order to tackle this challenge. These schemes are based on various mathematical problems like Lattices, Multi-variate polynomials, Hash based cryptography, Code-based crpytography etc.

The NTRU Encrypt Public Key Cryptosystem is an alternative to standard RSA and Elliptic Curve Cryptosytems which is resistant to quantum attacks. The cryptosystem is based on the shortest vector problem in a lattice. The encryption in NTRU encrypt is done by a robust algorithm which relies on the difficulty of factoring certain polynomials in a truncated polynomial ring into a quotient of two polynomials having very small coefficients.

The NTRU Encrypt Cryptosystem was developed by three mathematicians Jeffrey Hostein, Jill Pipher, and Joseph H. Silverman. In this SOP report, we have tried to implement the NTRU Encrypt in Python and also made some code optimizations in order to make its execution faster. The aim of this report is to provide the reader a brief of the work done by us throughout the semester,

THEORY

Group:

A group $(G, \cdot)$ is a set of elements where a binary operation is defined satisfying the following axioms:

• Closure: $\forall$ x, y $\in$ G, the product xy $\in$ G.
• Associativity: (xy)z = x(yz), $\forall$ x, y, z $\in$ G.
• Identity element: $\exists$ a unique identity element e $\in$ G such that ex = xe = x, $\forall$ x $\in$ G.
• Inverse element: $\forall$ x $\in$ G, $\exists$ y $\in$ G such that xy = yx = $x^{-1}x$ = e.

Ring:

A ring is a set $(R, +, \cdot)$ with two binary operators, addition and multiplication, which satisfies the following conditions:

• Additive associativity: (x + y) + z = x + (y + z), $\forall$ x, y, z $\in$ R.
• Additive commutativity: x + y = y + x, $\forall$ x, y $\in$ R.
• Additive identity element: $\exists$ an element 0 $\in$ R such that 0 + x = x + 0 = x, $\forall$ x $\in$ R.
• Additive inverse element: $\forall$ x $\in$ R $\exists$ −x $\in$ R such that x + (−x) = (−x) + x = 0.
• Left and right distributivity: $\forall$ x, y, z $\in$ R, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ and $(y + z) \cdot x = (y \cdot x) + (z \cdot x)$
• Multiplicative associativity: $\forall$ x, y, z $\in$ R, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Lattice:

In particular, for a linearly independent vector $v_1, \dots, v_n \in R^n$, the lattice generated is the set of vectors: $L(v_1, \dots, v_n) = (\mathbf{\Sigma}\, \alpha_i v_i \mid \alpha_i \in Z)$. The vectors $v_1, \dots, v_n$ are known as the basis of the lattice. The absolute value of the determinant of the vectors vi is denoted by d(L).

Polynomial Rings:

Polynomials rings are essential in the NTRU public-key algorithm in order to generate random polynomials. A polynomial ring is defined by a ring which contains the values the coefficients can obtain and a delimiter or maximum degree when polynomials over one variable are represented. A polynomial ring R[X] over the ring R in one variable X is formed by the set of all polynomials with coefficients in R. The elements of R[X] are the polynomials with the form: $F(X) = a_o + a_1X + a_2X^2 + \dots + a_nX^n = \Sigma\, a_iX^i$, where $a_i \in R$ and $0 \leq i \leq n$.

Truncated Polynomial Rings:

The NTRU public-key algorithm, uses random polynomials which are generated from a polynomial ring of the form $R[X] = Z[X]/(X^N - 1)$. The polynomials that form the ring R[X] have a degree smaller than N. The polynomials in the truncated ring R[X] are added in a regular way by adding their coefficients. The polynomial multiplication is a bit different since the resulting polynomial requires to satisfy the rule $X^N \equiv 1$. Said differently, the maximum degree of the resultant polynomial of a multiplication between two polynomials of the ring can not be greater than $N - 1$.

Invertibility in Truncated Polynomial Rings:

In order to be able to compute the inverse of a randomly chosen polynomial in a certain polynomial ring R_q defined as $R_q = (Z/qZ)[X]/(X^N - 1)$, it is important to take into account that not every polynomial might be invertible in the ring. The NTRU cryptosystem key generation is based on the computation of the inverse of a randomly generated polynomial from a polynomial ring.

IMPLEMENTATION

1. Key Generation

The first step in the NTRU-Encrypt cryptographic algorithm is to generate the public(h) and private keys(f,f_p) to encrypt the messages.

Choose random polynomials f and g with small coefficients. Then compute fp, i.e. the inverse of f (mod p) defined by:

$$f * f_p = 1 \pmod{p}\,.$$

Compute f_q, the inverse of f (mod q) that analogously satisfies the requirement:

$$f * f_q = 1 \pmod{q}\,.$$

Compute the polynomial:

$$h = g * p \cdot f_q\,.$$

The public key is h and the private key is the tuple (f,f_p).

Code Snippet for KeyGen:

```
    def key_gen(df,dg,deg,q,p):
    f = rand.randpol(df, df - 1, deg + 1)
    g = rand.randpol(dg, dg, deg + 1)
    print("f is:",f)
    print("g is:",g)
    N = len(f)
    D = [0]*(N+1)
    D[0] = -1
    D[N] = 1
    [gcd_f,s_f,t_f] = inverse.extEuclidPoly(f,D)
    finvp = inverse.modPoly(s_f,p)
    finvq = inverse.modPoly(s_f,q)
    while any(x is None for x in finvp) or any(x is None for x in
finvq) or finvp is None or finvq is None or all(x == 0 for x in
finvp) or all(x == 0 for x in finvq):
        print("inside while")
        f = rand.randpol(df, df - 1, deg + 1)
        [gcd_f,s_f,t_f] = inverse.extEuclidPoly(f,D)
        finvp = inverse.modPoly(s_f,p)
        finvq = inverse.modPoly(s_f,q)
    print("finvp is:",finvp)
    print("finvq is:",finvq)
    for i in range(len(g)):
        g[i] = g[i]*p
    h = polArithmetic.star_multiply(g,finvq,q)
    pk = [f,finvp]
    return h,pk
```

2. Encryption

The plaintext m is a polynomial with coefficients taken mod p. Note that convert the message m to a polynomial form is not part of NTRU public-key algorithm. Choose a blinding message r randomly from R with small coefficients.
The ciphertext is:

$$e = r * h + m \pmod{q}.$$

Code Snippet for Encryption:

```
    def encrypt(h,dr,msg,q,deg):
    r = rand.randpol(dr, dr, deg + 1)
    temp = polArithmetic.star_multiply(h,r,q)
    e = polArithmetic.polAdd(msg,temp,q)
    return e
```

3.Decryption

The decryption returns the message m from the encrypted message e using the Private Key (f,f_p). Compute

$$a = e * f \pmod{q},$$

choosing the coefficients of a to satisfy $-q/2 \le a_i < q/2$.

Reduce a modulo p:

$$b = a \pmod p\,.$$

Compute:

$$c = b * f_p \pmod p\,.$$

Then c mod p is equal to the plaintext m.

Code Snippet for Decryption:

```
def decrypt(pk, e, p,q):
a = polArithmetic.star_multiply(e,pk[0],q)
r = q/2
for i in range(len(a)):
    if a[i] < -1*r:
        a[i] = a[i] + q
    elif a[i] > r:
        a[i] = a[i] - q
for i in range(len(a)):
    a[i] = a[i]%p
c = polArithmetic.star_multiply(a,pk[1],p)
return c
```

Intermediate Computations:

1. **Inverse Calculation:**
   The polynomial inverse was caculated using the extended Euclidean algorithm for polynomials. The multiplication step was optimised using the numpy library's parallel multiplication support.

   ### Code Snippet for extended euclidean algo:

```
def extEuclidPoly(a,b):
switch = False
a=trim(a)
```

```
    b=trim(b)
    if len(a)<=len(b):
      a1, b1 = a, b
    else:
      a1, b1 = b, a
      switch = True
    Q,R=[],[]
    while b1 != [0]:
      [q,r]=divPoly(a1,b1)
      Q.append(q)
      R.append(r)
      a1=b1
      b1=r
    S=[0]*(len(Q)+2)
    T=[0]*(len(Q)+2)
    S[0],S[1],T[0],T[1] = [1],[0],[0],[1]

    for x in range(2, len(S)):
      S[x]=subPoly(S[x-2],multPoly(Q[x-2],S[x-1]))
      T[x]=subPoly(T[x-2],multPoly(Q[x-2],T[x-1]))

    gcdVal=R[len(R)-2]
    s_out=S[len(S)-2]
    t_out=T[len(T)-2]
    ### ADDITIONAL STEPS TO SCALE GCD SUCH THAT LEADING TERM
AS COEF OF 1:
    scaleFactor=gcdVal[len(gcdVal)-1]
    gcdVal=[x/scaleFactor for x in gcdVal]
    s_out=[x/scaleFactor for x in s_out]
    t_out=[x/scaleFactor for x in t_out]

    if switch:
      return [gcdVal,t_out,s_out]
    else:
      return [gcdVal,s_out,t_out]
```

2. Star Multiplication

Star Multiplication was carried out using the following function:

```
    def star_multiply(p1, p2, q):
```

```
    """Multiply two polynomials in Z_q[X]/(X**n - 1)"""
    [p1,p2] = inverse.resize(p1,p2)
    out = [0] * len(p1)
    for k in range(len(p1)):
        for i in range(len(p1)):
            if k >= i:
                out[k] += p1[i] * p2[k - i]
            else:
                out[k] += p1[i] * p2[len(p1) + k - i]
    return [x % q for x in out]
```

3. Random Polynomial Generation

A function for random polynomial generation was created, taking in input the number of coefficients to made positive and the number of coefficients to be made negative.

Code Snippet:

```
def randpol(positives,negatives,size):
    output = [0]*size
    for i in range(positives+negatives):
        if i < positives:
            output[i] = 1
        else:
            output[i] = -1
    random.shuffle(output)
    return output
```

FUTURE PROSPECTS

1) The multiplication in the inverse calculation can be further optimized using Fast-Fourier Transform multiplication techniques. Currently the multiplication has been optimzied using numpy library’s parallel multiplication support integrated in python.

2) Parallel processing can be achieved by running multiple threads at the same instance using the multithreading support of Python ecosystem. Currently a single thread is run upon execution of the program.

3) The encryption scheme can be combined with an authentication mechanism to generate a full proof authentication scheme.

REFERENCES

1) https://en.wikipedia.org/wiki/NTRUEncrypt

2) https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=3019&context=thesesdissertations

3) https://www.esat.kuleuven.be/cosic/publications/thesis-161.pdf

4) https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/grobschadl-lighteight-implmentation-NTRUE.pdf

5) https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9744589