Multivariate Encryption Schemes Based on Polynomial Equations over Real Numbers

Design Project (CS F376)

Report

By

Siddharth Kapoor

2018A7PS0232P

Professor in-charge

Dr. Abhishek Mishra

Introduction

The Multivariate Polynomial (MP) problem, defined as, given a prime number q, positive integers m, n and F(x) is a system of m polynomials over a finite field F_q in n variables $x = (x_1, x_2, \ldots, x_n)$, then finding a solution of $F(x) = 0$ in the finite field F_q^n is an NP-Complete problem, with no known efficient quantum algorithms. This makes any public key cryptosystem based on the MP problem a candidate for future post-quantum public key cryptographic schemes (MPKC Schemes).

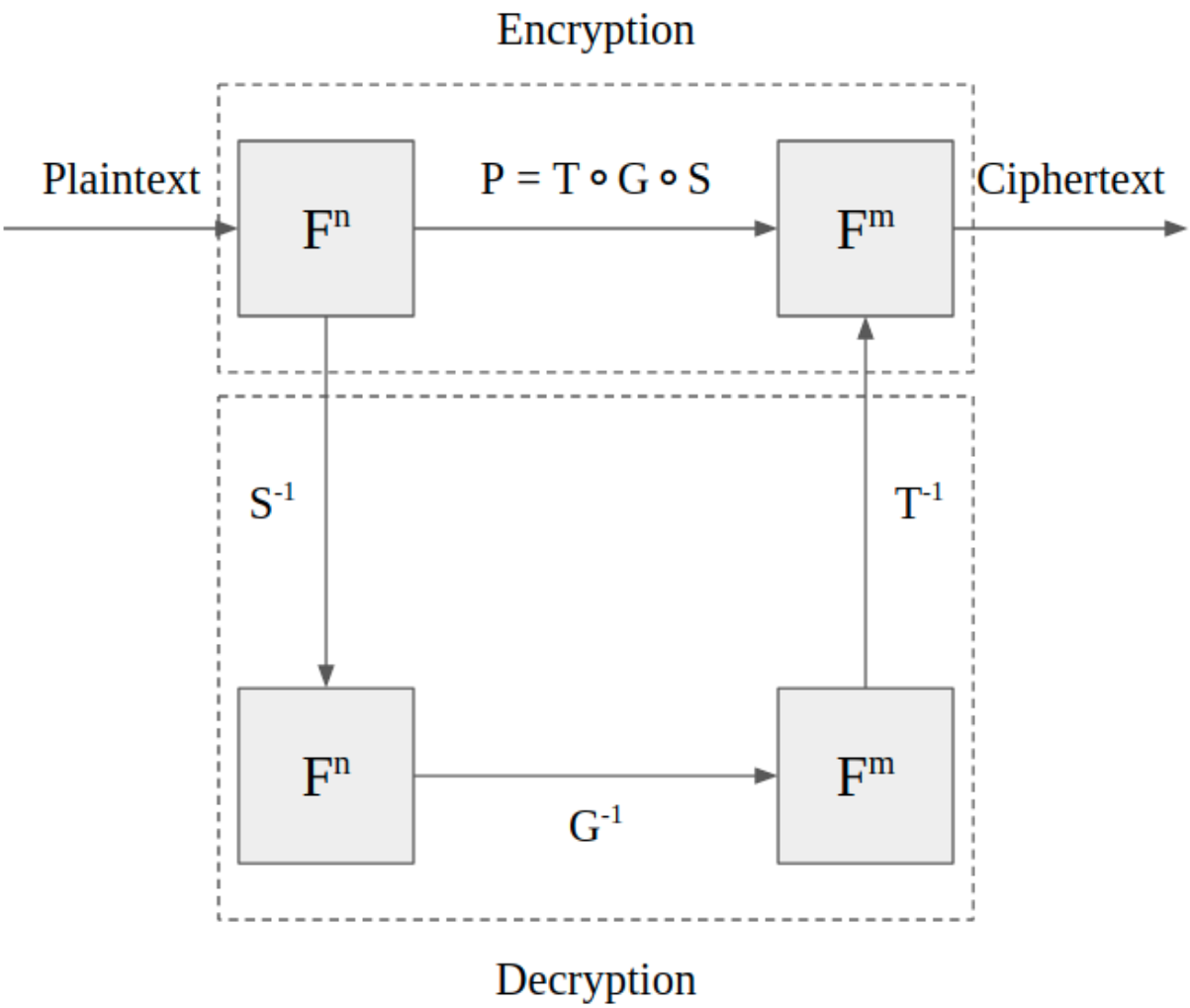
Some variants of the MP problem which are also used in some proposed cryptosystems are defined below.

- Constrained MP Problem: In the constrained MP problem, the solution space for $F(x) = 0$ is constrained to the set of integers in the range $(-L/2, L/2]$, where L is an odd positive integer.
- MQ Problem: When the system of polynomial equations F(x) comprises only quadratic polynomials, then the MP problem is called the Multivariate Quadratic (MQ) Problem.
- Constrained MQ Problem: Finding the solution of a system of quadratic polynomials $F(x) = 0$ over a constrained input space, is called the constrained MQ problem.

The Polynomial Encryption over Real Numbers (PERN) public key cryptosystem is based on the constrained MQ problem, but can also be generalized to the constrained MP problem.

Key Generation, Encryption and Decryption for General MPKC Schemes

- Key Generation: Most MPKC schemes follow a bipolar structure.
  - First choose the Central Map, i.e., an easily invertible one-to-one multivariate polynomial map $G(x) : F_q^n \rightarrow F_q^m$.
  - Then choose random affine isomorphisms $S : F_q^n \rightarrow F_q^n$ and $T : F_q^m \rightarrow F_q^m$. An affine isomorphism is a linear transformation that maintains lines and parallelism when applied to a vector space.
  - Compute the composition of T, G and S. $F(x) = T \circ G \circ S : F_q^n \rightarrow F_q^m$.
  - F(x) is the public key, and G(x), S, T are the private key.
- Encryption: For a plaintext $m \in F_q^n$, the cipher text c is computed by evaluating the public key polynomials over the input message, $c = F(m)$.
- Decryption: For a ciphertext $c \in F_q^m$, first compute $b_1 = T^{-1}(c)$, followed by $b_2 = G^{-1}(b_1)$, and finally $m' = S^{-1}(b_2)$ in this order. The m' obtained coincides with the plaintext m.



Polynomial Encryption over Real Numbers (PERN)

PERN is based on an earlier proposed scheme called the pq-method, which was based on the constrained MQ problem. However, PERN has been generalized to be based on the constrained MP problem as its underlying hard problem.

The decryption of PERN involves solving 2n polynomial equations in n variables. Therefore, techniques based on solving

nonlinear equations over the real numbers are used to solve the system of equations, with the constraint that the solution has integer components.

The affine isomorphism S is replaced by the identity map. Therefore, the number of monomials in G and F can be freely adjusted, allowing the key length to be freely adjusted as well.

Both the above points allow the scheme to be generalized to polynomials of any degree and not just quadratic polynomials.

- **Key Generation Algorithm**
  - Let L, L_G be odd prime integers, let n be a positive integer, and E a finite subset of $(Z_{\geq 0})^n$.
  - Choose multivariate polynomial systems of n polynomials $\Phi(x)$, $\Psi(x)$ with random coefficients from $I_{LG} = (-L_G/2, L_G/2] \cap Z$.
  - Compute M_Φ, M_Ψ which are the largest possible values in the codomains of $\Phi(x)$, $\Psi(x)$ respectively.
  - Choose a large odd prime number q such that $q > 4M_\Phi.M_\Psi$.
  - Choose n positive integers $r_1,..., r_n$ such that $M_\Phi < r_i < q$ & $2M_\Phi < |lar_q(r_ik)|$; $k \in \{1, \ldots, 2M_\Psi\}$.
  - Compute $\Psi_R(x) = (r_1.\psi_1(x),..., r_n.\psi_n(x))$, and $G(x) = (g_1(x),..., g_n(x)) = (\Phi(x) + \Psi_R(x)) \bmod q$.
  - Choose a random affine isomorphism T on F_q^n.
  - Compute $F(x) = T \circ G(x)$.

- The secret key is $\Phi(x)$, $\Psi(x)$, $\{r_1,..., r_n\}$, T, and the public key is F(x).

- **Encryption Algorithm**
  - Let $m \in I_L^n$ be the plaintext message.
  - Compute ciphertext $c = F(m) \in F_q^n$.
- **Decryption Algorithm**
  - Let $c \in F_q^n$ be a cipher text.
  - Compute $c' = (c_1,..., c_n) = T^{-1}(c)$.
  - For all $i = 1,..., n$, find $\tilde{b}_i \in [-M_\Psi, M_\Psi]$ satisfying $|lar_q(c_i' - r_i.\tilde{b}_i)| < M_\Phi$ and compute $\tilde{a}_i = lar_q(c_i' - r_i.\tilde{b}_i) \in Z$.
  - Solve the nonlinear equation system with a box constraint I_L^n, $\Phi(x) = (\tilde{a}_1,..., \tilde{a}_n)$, $\Psi(x) = (\tilde{b}_1,..., \tilde{b}_n)$.
  - The solution is denoted by $m' \in I_L^n$.
  - m' coincides with the plaintext m.

Solving Constrained Nonlinear Polynomials with Integer Coefficients

As defined in the decryption algorithm, compute the 2n integers $\tilde{a}_i$ and $\tilde{b}_i$ and set $\Phi(x) = (\tilde{a}_1,..., \tilde{a}_n)$, $\Psi(x) = (\tilde{b}_1,..., \tilde{b}_n)$. Hence, define $H(x) = (h_1(x), h_2(x),..., h_{2n}(x)) = ((\Phi(x) - (\tilde{a}_1,..., \tilde{a}_n)) \| (\Psi(x) - (\tilde{b}_1,..., \tilde{b}_n)))$. Therefore, when x is equal to the plaintext message m, then H(x) is equal to 0. Therefore, m can be considered a solution to the equation H(x) = 0. There may exist

other possible solutions to $H(x) = 0$, however, by Bezout's theorem, the probability of another such solution in R^n is very low.

By using methods to solve a system of nonlinear equations over real numbers, and using the fact that the solution must have integer components, a solution to the polynomial equation system $H(x) = 0$ can be approximated upto an error margin of 0.5 before rounding and verifying the solution.

Let $\theta(x)$ be defined as $\theta(x) = ½ * ||H(x)||_2^2 = ½ * (h_1(x)^2 + h_2(x)^2 + \cdots + h_{2n}(x)^2)$. An approximate solution to $H(x) = 0$ can be obtained by solving the least squares optimization problem of minimizing $\theta(x)$. Line search methods, such as Steepest Gradient Descent, Newton's Method, Levenberg-Marquardt Method etc., can be used to solve this optimization problem. In the paper, it was found that the Levenberg-Marquardt Method is the most efficient line search algorithm to solve the optimization problem.

The algorithm to obtain the solution to the equation $H(x) = 0$, using the Levenberg-Marquardt Method is described below:

- Given $H(x)$, an odd number L (same as the bound for the message space), and parameters $\alpha, \beta, \gamma \in (0, 1)$.
- Choose a random initial point $x_0 \in [-(L-1)/2, (L-1)/2]^n$
- Compute $J_H(x_0)$ which is equal to the Jacobian Matrix of $H(x)$ at the point x_0.
- Compute $e = -H(x_0)J_H(x_0)$.
- Compute $S = J_H(x_0)^T J_H(x_0)$.
- Solve for d_0 in the linear equation $d_0 S = e$. Therefore, $d_0 = eS^{-1}$.

- Compute the minimal non-negative integer l, such that $\theta(x_0+\beta^l) - \theta(x_0) \leq -\alpha\beta^l e d_0^T$. Hence, set $t_0 = \beta^l$.
- Set $x_0 = x_0 + t_0 d_0$. If the maximum of the absolute values of the components of $t_0 d_0$ is less than γ ($||t_0 d_0||_\infty < \gamma$), then continue to the next step, else go back to the previous step and continue finding the appropriate integer l. ($t_0 d_0 < \gamma$ signifies that x_0 is close enough to a stationary point).
- Set $\tilde{x}_0 = \text{round}(x_0)$, i.e., $\tilde{x}_0$ is obtained by component wise rounding of x_0 to integers.
- If $H(\tilde{x}_0) = 0$, then the decrypted message is equal to $\tilde{x}_0$, else go back to step 1 and choose a different initial point x_0.

Implementation Details

The cryptosystem as described in the paper has been implemented in C++. Some of the important snippets have been described below:

Struct PrivateKey:

```
typedef struct PrivateKey {
   ll n; // Number of Variables
   ll l; // Defines Range of Input Message
   ll lg; // Defines Range of Polynomial Coefficients
   std::vector<std::vector<ll>> phi; // Polynomial System Phi
   ll mPhi; // Largest Value in Codomain of Phi
   std::vector<std::vector<ll>> psi; // Polynomial System Psi
   ll mPsi; // Largest Value in Codomain of Psi
   std::vector<ll> r; // r values r1,..., rn
   std::vector<std::vector<ll>> T; // Random Affine Transformation T
   ll q; // Prime q
} PrivateKey;
```

Defines the structure of the private key.

Struct PublicKey:

```
typedef struct PublicKey {
```

```
    std::vector<std::vector<ll>> F; // Coefficients of the Public
Polynomial System
} PublicKey;
```

Defines the structure of the public key.

LeastAbsoluteRemainder():

```
// Output in range (-b / 2, b / 2]
ll LeastAbsoluteRemainder(ll a, ll b) {
    ll lar = a % b;
    lar = (lar + b) % b;
    lar = lar > b / 2 ? lar - b : lar;
    return lar;
}
```

Returns the remainder of a, on division by b, with the smallest absolute value. Returns an output in the range of $-b/2 < \text{output} \leq b/2$.

NumberOfMonomials():

```
// Compute nCr
ll Combinations(ll n, ll r) {
    if(r > n - r) r = n - r;
    if(r == 0) return 1;
    ll c = 1;
    for(ll i = 1; i <= r; ++i) {
        c *= n - r + i;
        c /= i;
    }
    return c;
}

// Compute Number of Possible Monomials for a Polynomial with n Variables
and d Degree
ll NumberOfMonomials(ll n, ll d) {
    // for(ll i = 1; i <= d; ++i) {
    //  ll c = Combinations(n + i - 1, i);
    //  num += c;
    // }
    ll num = Combinations(n + d, d); // The Above Code can be Simplified to
this using Combinatorial Identities
    return num;
}
```

Returns the maximum number of monomials that can be present in a polynomial in n variables with maximum degree d. The number of monomials in n variables of some degree i is equal to ${}^{n+i-1}C_i$. Therefore, the maximum number of monomials in a polynomial of maximum degree d, in n variables, is equal to $\sum {}^{n+i-1}C_i$ (i = 0, 1,…, d) = ${}^{n+d}C_d$.

GetNextPrime():

```
// Checks if a Number is Prime
bool IsPrime(ll num) {
   ll sqnum = std::sqrt(num);
   for(ll i = 2; i <= sqnum; ++i) {
       if(num % i == 0) return false;
   }
   return true;
}

// Finds the Next Prime Greater Than base
ll GetNextPrime(ll base) {
   ll nextPrime = base + 1;
   if(nextPrime <= 2) nextPrime = 2;
   else if(nextPrime % 2 == 0) nextPrime++;
   while(!IsPrime(nextPrime)) {
       nextPrime += 2;
   }
   return nextPrime;
}
```

Returns the next largest prime number greater than the given base number, i.e., p > base, where p is a prime number.

MatrixMultiplication():

```
// Multiply 2 Matrices of Any Dimensions (Integers)
std::vector<std::vector<ll>>
MatrixMultiplication(std::vector<std::vector<ll>> matA,
std::vector<std::vector<ll>> matB) {

   std::vector<std::vector<ll>> productMat;
   if(matA[0].size() != matB.size()) {
       std::cout << "ERROR : Matrix dimensions do not match for
```

```
multiplication.\n";
        return productMat;
    }

    for(ll i = 0; i < matA.size(); ++i) {
        std::vector<ll> product;
        ll tmpProd;
        for(ll c = 0; c < matB[0].size(); ++c) {
            tmpProd = 0;
            for(ll j = 0; j < matA[i].size(); ++j) {
                tmpProd += matA[i][j] * matB[j][c];
            }
            product.push_back(tmpProd);
        }
        productMat.push_back(product);
    }
    return productMat;
}
```

Multiplies 2 rectangular matrices consisting of integer values. Is overloaded with matrix multiplication of rectangular matrices with floating point values as well.

MatrixInverse():

```
// Returns the GCD and Coefficients of GCD(a, b)
std::tuple<ll, ll, ll> ExtendedEuclideanGCD(ll a, ll b) {
    std::tuple<ll, ll, ll> coefficients;
    if(b == 0) {
        coefficients = std::make_tuple(a, 1, 0);
    } else {
        ll x; // Coefficient of a
        ll y; // Coefficient of b
        ll gcd; // GCD of a and b
        std::tie(gcd, x, y) = ExtendedEuclideanGCD(b, Modulo(a, b));
        coefficients = std::make_tuple(gcd, y, x - y * (a / b));
    }
    return coefficients;
}

// Computes the Positive Multiplicative Inverse of b over a
ll MultiplicativeInverse(ll a, ll b) {
    ll inverse = 0;
    std::tie(std::ignore, std::ignore, inverse) = ExtendedEuclideanGCD(a,
b); // Only get the coefficient of b
```

```
    inverse = Modulo(inverse, a); // Convert negative coefficients to
positive ones
    return inverse;
}

// Invert a Square Matrix in the Integer Field Fq
std::vector<std::vector<ll>> MatrixInverse(std::vector<std::vector<ll>>
mat, ll q) {

    // Initializing the Augmented Matrix
    std::vector<std::vector<ll>> augmentedMat;
    for(ll i = 0; i < mat.size(); ++i) {
        std::vector<ll> row(2 * mat[i].size());
        for(ll j = 0; j < mat[i].size(); ++j) {
            row[j] = mat[i][j];
            row[j + mat[i].size()] = i == j ? 1 : 0;
        }
        augmentedMat.push_back(row);
    }

    // Running Gaussian Elimination Algorithm to obtain the Inverse
    for(ll i = 0; i < augmentedMat.size(); ++i) {

        ll inv = MultiplicativeInverse(q, augmentedMat[i][i]);
        for(ll j = 0; j < augmentedMat[i].size(); ++j) {
            augmentedMat[i][j] = Modulo(augmentedMat[i][j] * inv, q);
        }

        for(ll j = 0; j < augmentedMat.size(); ++j) {
            if(j == i) continue; // Skip the current row
            ll scale = Modulo(0 - augmentedMat[j][i], q);
            for(ll k = 0; k < augmentedMat[j].size(); ++k) {
                augmentedMat[j][k] = Modulo(augmentedMat[j][k] + scale *
augmentedMat[i][k], q);
            }
        }
    }

    // Extracting the Inverse from the Augmented Matrix
    std::vector<std::vector<ll>> inverseMat;
    for(ll i = 0; i < augmentedMat.size(); ++i) {
        std::vector<ll> row;
        for(ll j = augmentedMat[i].size() / 2; j < augmentedMat[i].size();
++j) {
            row.push_back(augmentedMat[i][j]);
        }
        inverseMat.push_back(row);
```

```
    }

    return inverseMat;
}
```

Computes the inverse of a square matrix over a finite field F_q using the Gaussian elimination method. The multiplicative inverses are calculated using the extended euclidean algorithm for computing GCD. The function is overloaded to compute the inverse of square matrices over the real numbers as well.

ComputeJacobianMatrix():

```
// Computes the Partial Derivatives of a Single Polynomial over All
Variables
std::vector<ld> ComputePartialDerivatives(std::vector<ll> coefficients,
std::vector<ld> x, ll degree) {
    // Only works for Quadratic Polynomial Equations
    std::vector<ld> xCopy = x;
    xCopy.push_back(1.0); // Pushing the Constant Term

    std::vector<ld> partialDerivatives;
    for(ll i = 0; i < x.size(); ++i) { // Only Compute the Partial
Derivatives w.r.t. the Variables and not the Constant Term
        ld result = 0.0;
        for(ll j = 0, l = 0; j < xCopy.size(); ++j) {
            for(ll k = 0; k < xCopy.size(); ++k, ++l) {
                if(j == k) {
                    if(i == j) {
                        result += 2.0 * coefficients[l] * xCopy[j]; //
derivative of xj^2 w.r.t. xj
                    } else {
                        result += 0.0; // derivative of xj^2 w.r.t. xi
                    }
                } else {
                    if(i == j) {
                        result += coefficients[l] * xCopy[k]; // derivative
of xj.xk w.r.t. xj
                    } else if(i == k) {
                        result += coefficients[l] * xCopy[j]; // derivative
of xj.xk w.r.t. xk
                    } else {
                        result += 0.0; // derivative of xj.xk w.r.t. xi
                    }
```

```
                }
            }
        }
        partialDerivatives.push_back(result);
    }

    return partialDerivatives;
}

// Computes the Jacobian of the Polynomial System
std::vector<std::vector<ld>>
ComputeJacobianMatrix(std::vector<std::vector<ll>> function,
std::vector<ld> x, ll degree) {
    std::vector<std::vector<ld>> jacobian;
    for(ll i = 0; i < function.size(); ++i) {
        jacobian.push_back(ComputePartialDerivatives(function[i], x,
degree));
    }
    return jacobian;
}
```

Computes the jacobian matrix for a system of polynomials at some point x_0. The Jacobian matrix of a system of polynomials is the matrix obtained by evaluating the partial derivatives of each polynomial with respect to each variable x_i, at the given point x_0.

$$\mathbf{J_f}(x, y) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}$$

The partial derivatives can be calculated for multivariate quadratic polynomials only.

ComputePolynomialSystemOutput():

```
// Compute the Result of a Single Polynomial on some input x (Integers)
ll ComputePolynomialOutput(std::vector<ll> coefficients, std::vector<ll>
x, ll degree) {
    x.push_back(1); // Appending the Constant Term (Degree 0 Term) in the
```

```
Input --> x = [x1, x2, ..., xn, 1]
   ll result = 0;
   // TODO: Re-implement for Generalized Degree
   // Current Implementation Works only for Degree 2 Polynomials
   for(ll i = 0, k = 0; i < x.size(); ++i) {
       for(ll j = i; j < x.size(); ++j, ++k) {
           result += x[i] * x[j] * coefficients[k];
       }
   }
   return result;
}

// Compute the Result of the Polynomial System on some input x (Integers)
std::vector<ll> ComputePolynomialSystemOutput(std::vector<std::vector<ll>>
coefficients, std::vector<ll> x, ll degree) {
   std::vector<ll> result;
   for(ll i = 0; i < coefficients.size(); ++i) {
       ll out = ComputePolynomialOutput(coefficients[i], x, degree);
       result.push_back(out);
   }
   return result;
}
```

This method computes the outputs of a system of n polynomials on the same integer input x. It makes use of the method ComputePolynomialOutput() which computes the output for a single polynomial on the input x. The method can only evaluate quadratic multivariate polynomials and not of higher degree polynomials. It assumes the polynomials to have a lexicographic monomial ordering. The method has been overloaded for evaluating the polynomials on floating point numbers as well.

GenerateKeyPair():

```
// Generate Public Private Key Pair
std::pair<PublicKey, PrivateKey> GenerateKeyPair(ll n, ll l, ll lg, ll
degree) {

   std::vector<std::vector<ll>> phiCoefficients = GenerateCoefficients(n,
n, degree, lg); // Coefficients for the Phi Polynomial System
   std::vector<std::vector<ll>> psiCoefficients = GenerateCoefficients(n,
n, degree, lg); // Coefficients for the Psi Polynomial System
```

```
    ll mPhi = GetMaxInCodomain(phiCoefficients, n, degree, l); // Largest
Value in Codomain of Phi
    ll mPsi = GetMaxInCodomain(psiCoefficients, n, degree, l); // Largest
Value in Codomain of Psi

    ll q; // Large Prime q
    std::vector<ll> rValues; // r Values
    std::tie(q, rValues) = GetRValues(n, mPhi, mPsi); // r Values and Prime
q

    std::vector<std::vector<ll>> centralMapG =
GetCentralMap(phiCoefficients, psiCoefficients, rValues, q, n); // The
Central Map G

    std::vector<std::vector<ll>> affineT = GetAffineTransformation(n, q);
// Random Affine Transformation T

    std::vector<std::vector<ll>> polynomialMapF =
GetFinalPolynomialMap(affineT, centralMapG, q); // Final Polynomial Map
and Public Key F

    // Save to publicKey
    std::cout << "Saving Public Key...\n";
    PublicKey publicKey;
    publicKey.F = polynomialMapF;
    WritePublicKeyToFile(publicKey, "PublicKey.txt");

    // Save to privateKey
    std::cout << "Saving Private Key...\n";
    PrivateKey privateKey;
    privateKey.n = n;
    privateKey.l = l;
    privateKey.lg = lg;
    privateKey.phi = phiCoefficients;
    privateKey.mPhi = mPhi;
    privateKey.psi = psiCoefficients;
    privateKey.mPsi = mPsi;
    privateKey.r = rValues;
    privateKey.T = affineT;
    privateKey.q = q;
    WritePrivateKeyToFile(privateKey, "PrivateKey.txt");

    return std::make_pair(publicKey, privateKey);
}
```

Generates the public-private key pair for the PERN public key

cryptosystem. It calls different methods corresponding to each step of the key generation algorithm and saves the keys to their respective text files.

GenerateCoefficients():

```
// Generate Coefficients for n Polynomials in m Variables and d Degree in
range (-r / 2, r / 2)
std::vector<std::vector<ll>> GenerateCoefficients(ll n, ll m, ll d, ll r)
{
   std::vector<std::vector<ll>> coefficients;
   ll numMonomials = NumberOfMonomials(m, d);
   ll lowLimit = LeastAbsoluteRemainder(r / 2 + 1, r);
   ll upLimit = LeastAbsoluteRemainder(r / 2, r);
   for(ll i = 0; i < n; ++i) {
       coefficients.push_back(GenerateRandom(numMonomials, lowLimit,
upLimit));
   }
   return coefficients;
}
```

Generates random coefficients in the range (-r/2, r/2) of n polynomials in m variables and degree d.

GetMaxInCodomain():

```
// Return Max Possible Value in Codomain of Polynomial System
ll GetMaxInCodomain(std::vector<std::vector<ll>> coefficients, ll n, ll
degree, ll l) {

   // Replacing the Coefficients with their Absolute Values
   for(ll i = 0; i < coefficients.size(); ++i) {
       for(ll j = 0; j < coefficients[i].size(); ++j) {
           coefficients[i][j] = std::abs(coefficients[i][j]);
       }
   }

   // Setting the Values of All the Variables to l / 2 to Maximize Output
   std::vector<ll> x;
   for(ll i = 0; i < n; ++i) {
       x.push_back(l / 2);
   }

   // Returning the Max Value from the Result of All the Polynomials in
```

```
the System
   std::vector<ll> result = ComputePolynomialSystemOutput(coefficients, x,
degree);
   ll max = 0;
   for(ll i = 0; i < result.size(); ++i) {
       max = std::max(max, result[i]);
   }
   return max;
}
```

Returns the maximum possible value in the codomain of a system of polynomials, if the input is constrained to the integers in the range [-l/2, l/2]. The maximum value is obtained by first replacing each coefficient for all the polynomials with their absolute values and evaluating the polynomials at the input (l/2, l/2,…,l/2)^n. The maximum of the outputs of all the polynomials is then returned.

GetRValues():

```
// Returns the r Values for the given Prime q. Else Returns Empty List

std::vector<ll> ComputeRValues(ll n, ll q, ll mPhi, ll mPsi) {
   std::vector<ll> rValues;
   ll base = 2 * mPhi;
   ll prevBase;

   for(ll i = 1; i <= n; ++i) {
       prevBase = base;
       for(ll r = base + 1; r < q; ++r) {
           bool flag = true;
           for(ll k = 1; k <= 2 * mPsi; ++k) {
               if(std::abs(LeastAbsoluteRemainder(r * k, q)) <= 2 * mPhi)
{
                   flag = false;
                   break;
               }
           }
           if(flag) {
               rValues.push_back(r);
               base = r;
               break;
           }
```

```
        }
        // If No More r Values are Found, Break the Loop
        if(base == prevBase) {
            break;
        }
    }

    // If n Distinct r Values are not Found, Empty the Vector
    if(rValues.size() != n) {
        rValues.clear();
    }
    return rValues;
}

// Returns the r Values and Prime q by Automatically Updating the Prime q
std::tuple<ll, std::vector<ll>> GetRValues(ll n, ll mPhi, ll mPsi) {
    ll base = 4 * mPhi * mPsi; // Prime q > 4 * mPhi * mPsi
    ll q; // Large Prime Number
    std::vector<ll> rValues; // List of the r Values;
    while(rValues.size() != n) {
        q = GetNextPrime(base);
        rValues.clear();
        rValues = ComputeRValues(n, q, mPhi, mPsi);
        base = q;
    }
    return std::make_tuple(q, rValues);
}
```

Gets n integers $r_1, r_2,\ldots,r_n$ and a prime q, as required by the step in the key generation algorithm step:

- Choose n positive integers $r_1,..., r_n$ such that $M_\Phi < r_i < q$ & $2M_\Phi < |lar_q(r_ik)|$; $k \in \{1, \ldots, 2M_\Psi\}$.

The prime q is increased until we can compute n distinct integers satisfying the given constraints.

GetCentralMap():

```
// Returns the Central Map G = (phi + r * psi) mod q
std::vector<std::vector<ll>> GetCentralMap(std::vector<std::vector<ll>>
phiCoefficients, std::vector<std::vector<ll>> psiCoefficients,
std::vector<ll> rValues, ll q, ll n) {
    std::vector<std::vector<ll>> centralMap;
```

```
    ll numMonomials = phiCoefficients[0].size(); // Both phi and psi have
Same Number of Monomials
    for(ll i = 0; i < n; ++i) {
        std::vector<ll> polyi;
        for(ll j = 0; j < numMonomials; ++j) {
            ll coeff = phiCoefficients[i][j] + psiCoefficients[i][j] *
rValues[i];
            coeff = Modulo(coeff, q);
            polyi.push_back(coeff);
        }
        centralMap.push_back(polyi);
    }
    return centralMap;
}
```

Computes the central map G(x) as described in the key generation algorithm:

- Compute $\Psi_R(x) = (r_1.\psi_1(x),..., r_n.\psi_n(x))$, and $G(x) = (g_1(x),..., g_n(x)) = (\Phi(x) + \Psi_R(x)) \bmod q$.

GetAffineTransformation():

```
// Generate a Random Affine Transformation over the Field (Fq)^n
std::vector<std::vector<ll>> GetAffineTransformation(ll n, ll q) {
    std::vector<std::vector<ll>> affineT;
    for(ll i = 0; i < n; ++i) {
        affineT.push_back(GenerateRandom(n, 0, q - 1));
    }
    return affineT;
}
```

Generates a random affine transformation T over the finite field F_q, which obscures the central map.

GetFinalPolynomialMap():

```
// Computes the Final Central Map (the Public Key) F = T o G(x)
std::vector<std::vector<ll>>
GetFinalPolynomialMap(std::vector<std::vector<ll>> affineT,
std::vector<std::vector<ll>> centralMapG, ll q) {
    std::vector<std::vector<ll>> polynomialMapF;
    polynomialMapF = MatrixMultiplication(affineT, centralMapG);
```

```
    for(ll i = 0; i < polynomialMapF.size(); ++i) {
        for(ll j = 0; j < polynomialMapF[i].size(); ++j) {
            polynomialMapF[i][j] = Modulo(polynomialMapF[i][j], q);
        }
    }
    return polynomialMapF;
}
```

Generates the final public polynomial map F(x) which is the composition of the affine transformation T and the central map G(x). $F(x) = T \circ G(x)$.

EncryptMessage():

```
// Encrypting the Input Message Using the Public Key
std::vector<ll> EncryptMessage(PublicKey publicKey, std::vector<ll>
message, ll degree) {
    std::vector<ll> cipherText = ComputePolynomialSystemOutput(publicKey.F,
message, degree);
    return cipherText;
}
```

Encrypts the input message text and outputs the cipher text by evaluating the public polynomial system F(x) on the message.

DecryptCipherText():

```
// Decrypting the Cipher Text Using the Private Key
std::vector<ll> DecryptCipherText(PrivateKey privateKey, std::vector<ll>
cipherText, ll degree) {
    // Brings the Cipher Text to the Finite Field Fq
    for(ll i = 0; i < cipherText.size(); ++i) {
        cipherText[i] = Modulo(cipherText[i], privateKey.q);
    }

    // Computing T^-1(c)
    std::vector<std::vector<ll>> inverseT = MatrixInverse(privateKey.T,
privateKey.q);
    std::vector<ll> inverseCipherText =
DeVectorize(MatrixMultiplication(inverseT, Vectorize(cipherText))); //
ComputePolynomialSystemOutput() not used as T is a system of linear
polynomials

    // Computing the RHS of Phi(x) and Psi(x)
```

```
    std::vector<ll> aValues;
    std::vector<ll> bValues;
    std::tie(aValues, bValues) = GetABValues(privateKey,
inverseCipherText); // Get the RHS of Phi(x) and Psi(x)

    // Solving the Non Linear Polynomial System using Levenberg-Marquardt
Method
    std::vector<ll> decryptedMessage =
SolveNonLinearEquationSystem(privateKey, aValues, bValues, degree);

    return decryptedMessage;
}
```

Decrypts the cipher text obtained from encryption and returns the original message text. The method first converts the cipher text to the finite field F_q. It then obtains $T^{-1}(c)$ by simply multiplying the cipher text with the matrix inverse of the affine transformation T. It then calls methods corresponding to each step in the decryption algorithm.

GetABValues():

```
// Compute the a Values (RHS of Phi(x)) and the b Values (RHS of Psi(x))
std::tuple<std::vector<ll>, std::vector<ll>> GetABValues(PrivateKey
privateKey, std::vector<ll> inverseCipherText) {
    std::vector<ll> aValues;
    std::vector<ll> bValues;
    for(ll i = 0; i < privateKey.n; ++i) {
        for(ll b = 0; b <= privateKey.mPsi; ++b) {
            if(std::abs(LeastAbsoluteRemainder(inverseCipherText[i] - b *
privateKey.r[i], privateKey.q)) < privateKey.mPhi) {
                bValues.push_back(b);

aValues.push_back(LeastAbsoluteRemainder(inverseCipherText[i] - b *
privateKey.r[i], privateKey.q));
                break;
            } else if(std::abs(LeastAbsoluteRemainder(inverseCipherText[i]
+ b * privateKey.r[i], privateKey.q)) < privateKey.mPhi) {
                bValues.push_back(0 - b);

aValues.push_back(LeastAbsoluteRemainder(inverseCipherText[i] + b *
privateKey.r[i], privateKey.q));
                break;
            }
```

```
        }
    }
    return std::make_tuple(aValues, bValues);
}
```

Obtains the 2n integers $\tilde{a}_i$, $\tilde{b}_i$, which are the solutions for the polynomial systems $\Phi(x)$ and $\Psi(x)$ respectively. Therefore, $\Phi(x) = (\tilde{a}_1,..., \tilde{a}_n)$, $\Psi(x) = (\tilde{b}_1,..., \tilde{b}_n)$. The procedure to obtain these values is described in the decryption algorithm:

- For all $i = 1,..., n$, find $\tilde{b}_i \in [-M_\Psi, M_\Psi]$ satisfying $|lar_q(c_i - r_i.\tilde{b}_i)| < M_\Phi$ and compute $\tilde{a}_i = lar_q(c_i - r_i.\tilde{b}_i) \in Z$.
- Solve the nonlinear equation system with a box constraint I_L^n, $\Phi(x) = (\tilde{a}_1,..., \tilde{a}_n)$, $\Psi(x) = (\tilde{b}_1,..., \tilde{b}_n)$.

SolveNonLinearEquationSystem():

```
// Compute Norm of a Vector = 0.5 * (vec[1]^2 + vec[2]^2 + ... + vec[n]^2)
ld ComputeNorm(std::vector<ld> vec) {
    ld norm = 0.0;
    for(ll i = 0; i < vec.size(); ++i) {
        norm += vec[i] * vec[i];
    }
    norm *= 0.5;
    return norm;
}

// Returns the Max Absolute Component of a Vector
ld MaxAbsoluteComponent(std::vector<ld> vec) {
    ld max = 0.0;
    for(ll i = 0; i < vec.size(); ++i) {
        max = std::abs(vec[i]) > max ? std::abs(vec[i]) : max;
    }
    return max;
}

// Solve the Non Linear Polynomial Equation System using the
Levenberg-Marquardt Method
std::vector<ll> SolveNonLinearEquationSystem(PrivateKey privateKey,
std::vector<ll> aValues, std::vector<ll> bValues, ll degree) {

    ld alpha = ALPHA; // Search Parameter
```

```
    ld beta = BETA; // Search Parameter
    ld gamma = GAMMA; // Search Parameter

    std::vector<std::vector<ll>> h; // H(x) = Concatenation of Phi(x) -
aValues and Psi(x) - bValues
    for(ll i = 0; i < privateKey.phi.size(); ++i) {
        h.push_back(privateKey.phi[i]);
        h[h.size() - 1][privateKey.phi[i].size() - 1] -= aValues[i]; //
Phi(x) - aValues
    }
    for(ll i = 0; i < privateKey.psi.size(); ++i) {
        h.push_back(privateKey.psi[i]);
        h[h.size() - 1][privateKey.psi[i].size() - 1] -= bValues[i]; //
Psi(x) - bValues
    }

    std::vector<ll> decryptedMessage;

    bool flag = false;
    ld lowLimit = - privateKey.l / 2.0;
    ld upLimit = privateKey.l / 2.0;
    std::vector<ld> x0;
    do {
        x0.clear();
        x0 = GenerateRandom(privateKey.n, lowLimit, upLimit); // Initial
Random Guess in Range [-l / 2, l / 2]
        std::vector<std::vector<ld>> hResult =
{ComputePolynomialSystemOutput(h, x0, degree)}; // Computing h(x0) : (1 x
2n)
        std::vector<std::vector<ld>> hJacobian = ComputeJacobianMatrix(h,
x0, degree); // Computing Jacobian of h(x) at x0 : (2n x n)
        std::vector<std::vector<ld>> negativeIdentityMatrix = {{-1.0}}; //
Negative Identity Matrix : (1 x 1)

        std::vector<std::vector<ld>> e =
MatrixMultiplication(MatrixMultiplication(negativeIdentityMatrix,
hResult), hJacobian); // e = -h(x0) * Jh(x0) : (1 x n)
        std::vector<std::vector<ld>> s =
MatrixMultiplication(MatrixTranspose(hJacobian), hJacobian); // s =
Jh(x0)^T * Jh(x0) : (n x n)
        std::vector<std::vector<ld>> d0 = MatrixMultiplication(e,
MatrixInverse(s)); // d0 * s = e : (1 x n)
        std::vector<std::vector<ld>> ed0Product = MatrixMultiplication(e,
MatrixTranspose(d0)); // e * d0^T : (1 x 1)

        ld t0 = 0.0; // Step Size
        std::vector<ld> x1; // x1 = x0 + t0 * d0
```

```
        std::vector<std::vector<ld>> hResultNew; // h(x1)
        ld thetaOldX = ComputeNorm(hResult[0]); // theta(x0) = Norm of
h(x0)
        ld thetaNewX = 0.0; // theta(x1)
        for(ll i = 0; i >= 0; ++i) {
            ld stepSize = std::pow(beta, i);
            x1.clear();
            for(ll j = 0; j < x0.size(); ++j) {
                x1.push_back(x0[j] + stepSize * d0[0][j]);
            }

            hResultNew.clear();
            hResultNew = {ComputePolynomialSystemOutput(h, x1, degree)};
            thetaNewX = ComputeNorm(hResultNew[0]);
            ld delta = (-alpha) * stepSize * ed0Product[0][0];
            if(thetaNewX - thetaOldX <= delta) {
                t0 = stepSize;
            } else {
                continue;
            }

            std::vector<ld> deltaX; // x1 = x0 + deltaX
            for(ll i = 0; i < x0.size(); ++i) {
                deltaX.push_back(x1[i] - x0[i]);
            }
            ld maxComponent = MaxAbsoluteComponent(deltaX);
            if(maxComponent < gamma) {
                break;
            }
        }

        decryptedMessage.clear();
        for(ll i = 0; i < x1.size(); ++i) {
            decryptedMessage.push_back((ll) std::round(x1[i]));
        }

        std::vector<ll> hVerify = ComputePolynomialSystemOutput(h,
decryptedMessage, degree);
        flag = true;
        for(ll i = 0; i < hVerify.size(); ++i) {
            if(hVerify[i] != 0) {
                flag = false;
                break;
            }
        }
    } while(!flag);
```

```
    return decryptedMessage;
}
```

This method solves the system of polynomial equations $\Phi(x) = (\tilde{a}_1,..., \tilde{a}_n)$, $\Psi(x) = (\tilde{b}_1,..., \tilde{b}_n)$ to obtain the original message. The function uses the Levenberg-Marquardt line search method to compute the solutions. The steps of the algorithm are described in the previous section.

Results

The program was compiled using the g++ compiler using the -Ofast flag for maximum compiler optimization for speed of execution. The code is many orders of magnitude slower than the authors' code, according to the results obtained by them in the original paper. The code had to be run at much smaller security parameters than the recommended values to keep the execution time low. The results have been tabulated in the table below and shows the average time over 10 runs each, for every security parameter setting:

Security Parameters (n, l, l_G)	Key Generation Time (s)	Encryption (μs)	Decryption (s)
(3, 7, 5)	2.2042	2.0842	0.0094
(4, 7, 5)	7.624	2.7401	0.1461

(5, 7, 5)	38.6222	3.5258	2.0588

Challenges

Implementing the cryptosystem from scratch in C++, without the help of scientific mathematical libraries, was by far the biggest challenge in the implementation of this project. All the matrix operations (multiplication, transpose, inverse) and other mathematical operations (extended euclidean GCD, multiplicative inverse, partial derivatives, jacobian matrix) were implemented from scratch.

Understanding and implementing the paper required the understanding of several concepts of abstract algebra, such as Group, Ring and Field theory, and methods to solve a system of nonlinear equations, such as the Grobner Basis technique and Buchburger's Algorithm. Understanding of optimization techniques such as the line search methods, like Steepest Gradient Descent, Newton-Raphson method and Levenberg-Marquardt method, was also required.

Limitations

In the current state, the code to implement the paper has several limitations.

- The code is several orders of magnitude slower than the code used by the authors to obtain their results. The Key Generation algorithm, specifically the GetRValues() method, is the biggest bottleneck in the entire process,

taking several seconds to execute even on small parameters. The method can be optimized by pre-computing and storing the primes in a large range, so that the program can simply look up the next prime from a table, instead of iterating through every integer, checking if it's prime or not. The polynomial and matrix operations can also be optimized by using well established and tested scientific mathematical libraries.

- It can only be used with quadratic multivariate polynomial systems and has not been generalized for higher degree polynomials.
- The parameters ALPHA, BETA and GAMMA for the Levenberg-Marquardt method were tuned manually, and can be further tuned for much better decryption speeds. It was observed that the parameter ALPHA had little effect on the decryption time, while BETA and GAMMA had a significant effect, with the speeds getting better as the values were decreased upto a certain level.

Appendix

Compilation instructions:

Run the following command to compile the code. It is assumed that the <bits/stdc++.h>, <fstream> and <chrono> library is installed.

$ g++ -Ofast -o pern pern.cpp

This creates a binary executable named pern in the current directory.

Execution instructions:

The binary executable can be run directly from the command line.

$ ./pern

It can also be provided with an input file to read the inputs directly.

$ ./pern < input.txt

Sample Input:

The first line of input consists of the security parameters n, l, l_G in that order. The second prompt for the input happens after the generation of the public-private key pair. This time the program expects n integer values in the range [-l/2, l/2]. This is the input message which will be encrypted. A sample input file below:

```
5 7 5
-1 1 3 0 -2
```

Sample Private Key File:

```
n:
5

l:
7

lg:
5

Phi:
1 0 2 0 1 1 -2 -1 1 1 -2 -1 -2 -2 -1 0 2 2 -2 -2 -2
0 2 0 1 2 2 -1 0 2 1 -2 -1 2 0 -1 -2 0 0 -2 0 -2
-1 0 0 2 2 -1 2 -1 1 -1 1 2 1 2 2 0 0 -1 -1 1 2
2 2 1 -1 2 1 -2 2 -2 0 1 1 -2 0 2 1 1 -1 0 2 -1
1 2 0 0 0 -2 -1 -2 -2 -2 0 2 -2 1 2 1 -2 -2 -1 1 -2

MPhi:
194

Psi:
2 1 -2 -2 -1 1 1 -2 -2 -2 2 2 0 1 1 2 2 -1 2 -1 1
-1 0 1 0 0 1 0 1 -1 -2 2 -1 -2 -2 -2 -2 1 0 -2 -2 1
0 -2 0 0 2 -1 1 -1 -2 -1 2 0 2 -1 -1 -2 -2 -2 1 -2 2
-2 0 -1 0 1 0 2 1 0 1 0 0 -2 -2 2 1 1 0 1 0 -1
0 1 0 2 1 0 -1 0 -2 0 2 0 2 -1 -2 -1 0 -1 -2 0 -2

MPsi:
235

r Values:
389 39295 45909 71976 78590

T:
4027 124484 42148 68249 92159
4055 124433 64213 79126 45297
153480 165495 35052 40608 135648
108682 158845 41516 92649 145437
40553 166876 9682 151201 69963

q:
183247
```

Sample Public Key File:

```
F:
46344 107034 74163 182241 67516 131208 111589 178290 125217 164244 72341
39550 174567 97973 35571 20412 132425 89500 165132 20582 132796
```

```
89022 74142 23216 37664 173166 60318 91046 137415 173132 66604 163047
10968 154154 156149 78917 35773 153721 140482 22353 181156 5454
2436 60332 95409 79044 168932 130056 30753 15372 144954 53897 65996 120187
154988 168028 54567 4613 37440 141730 49426 12552 86193
128481 41573 141151 162403 34766 182836 125803 44802 122622 70830 72907
38549 99446 54528 141036 160098 102682 154132 39975 85730 41135
12534 74223 111989 135159 50915 173335 173094 32393 12082 99151 123868
53040 48066 69962 15194 175838 142317 14302 78877 70843 75730
```

Sample Output:

The output shows the cipher text, the decrypted plain text, and the time taken to generate the key pair, encrypt the input message, and decrypt the cipher text.

```
Enter Security Parameters (n L Lg):
5 7 5

Generating Key Pair...
Saving Public Key...
Saving Private Key...
Time taken to generate Key Pair = 39.817s

Enter Message to Encrypt (Enter 5 numbers in the range [-3, 3]):
-1 1 3 0 -2

Encrypting Input Message...
Encrypted Cipher Text: 737942 -104501 394765 214063 79400
Time taken to Encrypt Message = 4.705us

Decrypting Cipher Text...
Decrypted Message: -1 1 3 0 -2

Time taken to Decrypt Message = 2.066s
```

Code:

Save the code in a file named pern.cpp

```
/**
* Author: Siddharth Kapoor (2018A7PS0232P)
* Email: f20180232@pilani.bits-pilani.ac.in
* BITS Pilani
*/

#include <bits/stdc++.h>
#include <chrono>
#include <fstream>

#define ll long long int
#define ld long double
#define DEGREE 2
#define ALPHA 10e-8
#define BETA 10e-5
#define GAMMA 10e-5

typedef struct PublicKey {
   std::vector<std::vector<ll>> F; // Coefficients of the Public
Polynomial System
} PublicKey;

typedef struct PrivateKey {
   ll n; // Number of Variables
   ll l; // Defines Range of Input Message
   ll lg; // Defines Range of Polynomial Coefficients
   std::vector<std::vector<ll>> phi; // Polynomial System Phi
   ll mPhi; // Largest Value in Codomain of Phi
   std::vector<std::vector<ll>> psi; // Polynomial System Psi
   ll mPsi; // Largest Value in Codomain of Psi
   std::vector<ll> r; // r values r1,..., rn
   std::vector<std::vector<ll>> T; // Random Affine Transformation T
   ll q; // Prime q
} PrivateKey;

// Output in range [0, b - 1]
ll Modulo(ll a, ll b) {
   ll mod = a % b;
   mod = (mod + b) % b;
   return mod;
}
```

```cpp
// Output in range (-b / 2, b / 2]
ll LeastAbsoluteRemainder(ll a, ll b) {
   ll lar = a % b;
   lar = (lar + b) % b;
   lar = lar > b / 2 ? lar - b : lar;
   return lar;
}

// Compute nCr
ll Combinations(ll n, ll r) {
   if(r > n - r) r = n - r;
   if(r == 0) return 1;
   ll c = 1;
   for(ll i = 1; i <= r; ++i) {
       c *= n - r + i;
       c /= i;
   }
   return c;
}

// Compute Number of Possible Monomials for a Polynomial with n Variables
and d Degree
ll NumberOfMonomials(ll n, ll d) {
   // for(ll i = 1; i <= d; ++i) {
   //  ll c = Combinations(n + i - 1, i);
   //  num += c;
   // }
   ll num = Combinations(n + d, d); // The Above Code can be Simplified to
this using Combinatorial Identities
   return num;
}

// Generate List of Random Integers in the range [lowLimit, upLimit]
std::vector<ll> GenerateRandom(ll n, ll lowLimit, ll upLimit) {
   std::vector<ll> list;
   ll range = upLimit - lowLimit + 1;
   for(ll i = 0; i < n; ++i) {
       ll random = (std::rand() % range) + lowLimit;
       list.push_back(random);
   }
   return list;
}

// Generate List of Random Real Numbers in the range [lowLimit, upLimit]
std::vector<ld> GenerateRandom(ll n, ld lowLimit, ld upLimit) {
   std::vector<ld> list;
   ll range = (ll) upLimit - (ll) lowLimit + 1;
```

```
    for(ll i = 0; i < n; ++i) {
        ld random = (std::rand() % range) + lowLimit + ((ld) std::rand()) /
RAND_MAX;
        list.push_back(random);
    }
    return list;
}

// Generate Coefficients for n Polynomials in m Variables and d Degree in
range (-r / 2, r / 2)
std::vector<std::vector<ll>> GenerateCoefficients(ll n, ll m, ll d, ll r)
{
    std::vector<std::vector<ll>> coefficients;
    ll numMonomials = NumberOfMonomials(m, d);
    ll lowLimit = LeastAbsoluteRemainder(r / 2 + 1, r);
    ll upLimit = LeastAbsoluteRemainder(r / 2, r);
    for(ll i = 0; i < n; ++i) {
        coefficients.push_back(GenerateRandom(numMonomials, lowLimit,
upLimit));
    }
    return coefficients;
}

// Compute the Result of a Single Polynomial on some input x (Integers)
ll ComputePolynomialOutput(std::vector<ll> coefficients, std::vector<ll>
x, ll degree) {
    x.push_back(1); // Appending the Constant Term (Degree 0 Term) in the
Input --> x = [x1, x2, ..., xn, 1]
    ll result = 0;
    // TODO: Re-implement for Generalized Degree
    // Current Implementation Works only for Degree 2 Polynomials
    for(ll i = 0, k = 0; i < x.size(); ++i) {
        for(ll j = i; j < x.size(); ++j, ++k) {
            result += x[i] * x[j] * coefficients[k];
        }
    }
    return result;
}

// Compute the Result of the Polynomial System on some input x (Integers)
std::vector<ll> ComputePolynomialSystemOutput(std::vector<std::vector<ll>>
coefficients, std::vector<ll> x, ll degree) {
    std::vector<ll> result;
    for(ll i = 0; i < coefficients.size(); ++i) {
        ll out = ComputePolynomialOutput(coefficients[i], x, degree);
        result.push_back(out);
    }
```

```
    return result;
}

// Compute the Result of a Single Polynomial on some input x (Real
Numbers)
ld ComputePolynomialOutput(std::vector<ll> coefficients, std::vector<ld>
x, ll degree) {
    x.push_back(1.0); // Appending the Constant Term (Degree 0 Term) in the
Input --> x = [x1, x2, ..., xn, 1.0]
    ld result = 0;
    // TODO: Re-implement for Generalized Degree
    // Current Implementation Works only for Degree 2 Polynomials
    for(ll i = 0, k = 0; i < x.size(); ++i) {
        for(ll j = i; j < x.size(); ++j, ++k) {
            result += x[i] * x[j] * coefficients[k];
        }
    }
    return result;
}

// Compute the Result of the Polynomial System on some input x (Real
Numbers)
std::vector<ld> ComputePolynomialSystemOutput(std::vector<std::vector<ll>>
coefficients, std::vector<ld> x, ll degree) {
    std::vector<ld> result;
    for(ll i = 0; i < coefficients.size(); ++i) {
        ld out = ComputePolynomialOutput(coefficients[i], x, degree);
        result.push_back(out);
    }
    return result;
}

// Return Max Possible Value in Codomain of Polynomial System
ll GetMaxInCodomain(std::vector<std::vector<ll>> coefficients, ll n, ll
degree, ll l) {

    // Replacing the Coefficients with their Absolute Values
    for(ll i = 0; i < coefficients.size(); ++i) {
        for(ll j = 0; j < coefficients[i].size(); ++j) {
            coefficients[i][j] = std::abs(coefficients[i][j]);
        }
    }

    // Setting the Values of All the Variables to 1 / 2 to Maximize Output
    std::vector<ll> x;
    for(ll i = 0; i < n; ++i) {
        x.push_back(l / 2);
```

```
    }

    // Returning the Max Value from the Result of All the Polynomials in
the System
    std::vector<ll> result = ComputePolynomialSystemOutput(coefficients, x,
degree);
    ll max = 0;
    for(ll i = 0; i < result.size(); ++i) {
        max = std::max(max, result[i]);
    }
    return max;
}

// Checks if a Number is Prime
bool IsPrime(ll num) {
    ll sqnum = std::sqrt(num);
    for(ll i = 2; i <= sqnum; ++i) {
        if(num % i == 0) return false;
    }
    return true;
}

// Finds the Next Prime Greater Than base
ll GetNextPrime(ll base) {
    ll nextPrime = base + 1;
    if(nextPrime <= 2) nextPrime = 2;
    else if(nextPrime % 2 == 0) nextPrime++;
    while(!IsPrime(nextPrime)) {
        nextPrime += 2;
    }
    return nextPrime;
}

// Returns the r Values for the given Prime q. Else Returns Empty List
std::vector<ll> ComputeRValues(ll n, ll q, ll mPhi, ll mPsi) {

    std::vector<ll> rValues;
    ll base = 2 * mPhi;
    ll prevBase;

    for(ll i = 1; i <= n; ++i) {
        prevBase = base;
        for(ll r = base + 1; r < q; ++r) {
            bool flag = true;
            for(ll k = 1; k <= 2 * mPsi; ++k) {
                if(std::abs(LeastAbsoluteRemainder(r * k, q)) <= 2 * mPhi)
{
```

```
                    flag = false;
                    break;
                }
            }
            if(flag) {
                rValues.push_back(r);
                base = r;
                break;
            }
        }
        // If No More r Values are Found, Break the Loop
        if(base == prevBase) {
            break;
        }
    }

    // If n Distinct r Values are not Found, Empty the Vector
    if(rValues.size() != n) {
        rValues.clear();
    }
    return rValues;
}

// Returns the r Values and Prime q by Automatically Updating the Prime q
std::tuple<ll, std::vector<ll>> GetRValues(ll n, ll mPhi, ll mPsi) {
    ll base = 4 * mPhi * mPsi; // Prime q > 4 * mPhi * mPsi
    ll q; // Large Prime Number
    std::vector<ll> rValues; // List of the r Values;
    while(rValues.size() != n) {
        q = GetNextPrime(base);
        rValues.clear();
        rValues = ComputeRValues(n, q, mPhi, mPsi);
        base = q;
    }
    return std::make_tuple(q, rValues);
}

// Returns the Central Map G = (phi + r * psi) mod q
std::vector<std::vector<ll>> GetCentralMap(std::vector<std::vector<ll>>
phiCoefficients, std::vector<std::vector<ll>> psiCoefficients,
std::vector<ll> rValues, ll q, ll n) {
    std::vector<std::vector<ll>> centralMap;
    ll numMonomials = phiCoefficients[0].size(); // Both phi and psi have
Same Number of Monomials
    for(ll i = 0; i < n; ++i) {
        std::vector<ll> polyi;
        for(ll j = 0; j < numMonomials; ++j) {
```

```
            ll coeff = phiCoefficients[i][j] + psiCoefficients[i][j] *
rValues[i];
            coeff = Modulo(coeff, q);
            polyi.push_back(coeff);
        }
        centralMap.push_back(polyi);
    }
    return centralMap;
}

// Generate a Random Affine Transformation over the Field (Fq)^n
std::vector<std::vector<ll>> GetAffineTransformation(ll n, ll q) {
    std::vector<std::vector<ll>> affineT;
    for(ll i = 0; i < n; ++i) {
        affineT.push_back(GenerateRandom(n, 0, q - 1));
    }
    return affineT;
}

// Multiply 2 Matrices of Any Dimensions (Integers)
std::vector<std::vector<ll>>
MatrixMultiplication(std::vector<std::vector<ll>> matA,
std::vector<std::vector<ll>> matB) {

    std::vector<std::vector<ll>> productMat;
    if(matA[0].size() != matB.size()) {
        std::cout << "ERROR : Matrix dimensions do not match for
multiplication.\n";
        return productMat;
    }

    for(ll i = 0; i < matA.size(); ++i) {
        std::vector<ll> product;
        ll tmpProd;
        for(ll c = 0; c < matB[0].size(); ++c) {
            tmpProd = 0;
            for(ll j = 0; j < matA[i].size(); ++j) {
                tmpProd += matA[i][j] * matB[j][c];
            }
            product.push_back(tmpProd);
        }
        productMat.push_back(product);
    }
    return productMat;
}

// Multiply 2 Matrices of Any Dimensions (Real Numbers)
```

```
std::vector<std::vector<ld>>
MatrixMultiplication(std::vector<std::vector<ld>> matA,
std::vector<std::vector<ld>> matB) {

   std::vector<std::vector<ld>> productMat;
   if(matA[0].size() != matB.size()) {
       std::cout << "ERROR : Matrix dimensions do not match for
multiplication.\n";
       return productMat;
   }

   for(ll i = 0; i < matA.size(); ++i) {
       std::vector<ld> product;
       ll tmpProd;
       for(ll c = 0; c < matB[0].size(); ++c) {
           tmpProd = 0;
           for(ll j = 0; j < matA[i].size(); ++j) {
               tmpProd += matA[i][j] * matB[j][c];
           }
           product.push_back(tmpProd);
       }
       productMat.push_back(product);
   }
   return productMat;
}

// Computes the Final Central Map (the Public Key) F = T o G(x)
std::vector<std::vector<ll>>
GetFinalPolynomialMap(std::vector<std::vector<ll>> affineT,
std::vector<std::vector<ll>> centralMapG, ll q) {
   std::vector<std::vector<ll>> polynomialMapF;
   polynomialMapF = MatrixMultiplication(affineT, centralMapG);
   for(ll i = 0; i < polynomialMapF.size(); ++i) {
       for(ll j = 0; j < polynomialMapF[i].size(); ++j) {
           polynomialMapF[i][j] = Modulo(polynomialMapF[i][j], q);
       }
   }
   return polynomialMapF;
}

// Returns the GCD and Coefficients of GCD(a, b)
std::tuple<ll, ll, ll> ExtendedEuclideanGCD(ll a, ll b) {
   std::tuple<ll, ll, ll> coefficients;
   if(b == 0) {
       coefficients = std::make_tuple(a, 1, 0);
   } else {
       ll x; // Coefficient of a
```

```
        ll y; // Coefficient of b
        ll gcd; // GCD of a and b
        std::tie(gcd, x, y) = ExtendedEuclideanGCD(b, Modulo(a, b));
        coefficients = std::make_tuple(gcd, y, x - y * (a / b));
    }
    return coefficients;
}

// Computes the Positive Multiplicative Inverse of b over a
ll MultiplicativeInverse(ll a, ll b) {
    ll inverse = 0;
    std::tie(std::ignore, std::ignore, inverse) = ExtendedEuclideanGCD(a,
b); // Only get the coefficient of b
    inverse = Modulo(inverse, a); // Convert negative coefficients to
positive ones
    return inverse;
}

// Invert a Square Matrix in the Integer Field Fq
std::vector<std::vector<ll>> MatrixInverse(std::vector<std::vector<ll>>
mat, ll q) {

    // Initializing the Augmented Matrix
    std::vector<std::vector<ll>> augmentedMat;
    for(ll i = 0; i < mat.size(); ++i) {
        std::vector<ll> row(2 * mat[i].size());
        for(ll j = 0; j < mat[i].size(); ++j) {
            row[j] = mat[i][j];
            row[j + mat[i].size()] = i == j ? 1 : 0;
        }
        augmentedMat.push_back(row);
    }

    // Running Gaussian Elimination Algorithm to obtain the Inverse
    for(ll i = 0; i < augmentedMat.size(); ++i) {

        ll inv = MultiplicativeInverse(q, augmentedMat[i][i]);
        for(ll j = 0; j < augmentedMat[i].size(); ++j) {
            augmentedMat[i][j] = Modulo(augmentedMat[i][j] * inv, q);
        }

        for(ll j = 0; j < augmentedMat.size(); ++j) {
            if(j == i) continue; // Skip the current row
            ll scale = Modulo(0 - augmentedMat[j][i], q);
            for(ll k = 0; k < augmentedMat[j].size(); ++k) {
                augmentedMat[j][k] = Modulo(augmentedMat[j][k] + scale *
augmentedMat[i][k], q);
```

```
            }
        }
    }

    // Extracting the Inverse from the Augmented Matrix
    std::vector<std::vector<ll>> inverseMat;
    for(ll i = 0; i < augmentedMat.size(); ++i) {
        std::vector<ll> row;
        for(ll j = augmentedMat[i].size() / 2; j < augmentedMat[i].size();
++j) {
            row.push_back(augmentedMat[i][j]);
        }
        inverseMat.push_back(row);
    }

    return inverseMat;
}

// Invert a Square Matrix (Real Numbers)
std::vector<std::vector<ld>> MatrixInverse(std::vector<std::vector<ld>>
mat) {

    // Initializing the Augmented Matrix
    std::vector<std::vector<ld>> augmentedMat;
    for(ll i = 0; i < mat.size(); ++i) {
        std::vector<ld> row(2 * mat[i].size());
        for(ll j = 0; j < mat[i].size(); ++j) {
            row[j] = mat[i][j];
            row[j + mat[i].size()] = i == j ? 1.0 : 0.0;
        }
        augmentedMat.push_back(row);
    }

    // Running Gaussian Elimination Algorithm to obtain the Inverse
    for(ll i = 0; i < augmentedMat.size(); ++i) {

        ld inv = 1.0 / augmentedMat[i][i];
        for(ll j = 0; j < augmentedMat[i].size(); ++j) {
            augmentedMat[i][j] *= inv;
        }

        for(ll j = 0; j < augmentedMat.size(); ++j) {
            if(j == i) continue; // Skip the current row
            ld scale = 0 - augmentedMat[j][i];
            for(ll k = 0; k < augmentedMat[j].size(); ++k) {
                augmentedMat[j][k] += scale * augmentedMat[i][k];
            }
```

```
        }
    }

    // Extracting the Inverse from the Augmented Matrix
    std::vector<std::vector<ld>> inverseMat;
    for(ll i = 0; i < augmentedMat.size(); ++i) {
        std::vector<ld> row;
        for(ll j = augmentedMat[i].size() / 2; j < augmentedMat[i].size();
++j) {
            row.push_back(augmentedMat[i][j]);
        }
        inverseMat.push_back(row);
    }

    return inverseMat;
}

// Convert a 1 x n Matrix into an n x 1 Vector
std::vector<std::vector<ll>> Vectorize(std::vector<ll> matA) {
    std::vector<std::vector<ll>> vecA;
    for(ll i = 0; i < matA.size(); ++i) {
        std::vector<ll> element = {matA[i]};
        vecA.push_back(element);
    }
    return vecA;
}

// Convert an n x 1 Vector into a 1 x n Matrix
std::vector<ll> DeVectorize(std::vector<std::vector<ll>> vecA) {
    std::vector<ll> matA;
    for(ll i = 0; i < vecA.size(); ++i) {
        matA.push_back(vecA[i][0]);
    }
    return matA;
}

// Compute the a Values (RHS of Phi(x)) and the b Values (RHS of Psi(x))
std::tuple<std::vector<ll>, std::vector<ll>> GetABValues(PrivateKey
privateKey, std::vector<ll> inverseCipherText) {
    std::vector<ll> aValues;
    std::vector<ll> bValues;
    for(ll i = 0; i < privateKey.n; ++i) {
        for(ll b = 0; b <= privateKey.mPsi; ++b) {
            if(std::abs(LeastAbsoluteRemainder(inverseCipherText[i] - b *
privateKey.r[i], privateKey.q)) < privateKey.mPhi) {
                bValues.push_back(b);
```

```
aValues.push_back(LeastAbsoluteRemainder(inverseCipherText[i] - b *
privateKey.r[i], privateKey.q));
                break;
            } else if(std::abs(LeastAbsoluteRemainder(inverseCipherText[i]
+ b * privateKey.r[i], privateKey.q)) < privateKey.mPhi) {
                bValues.push_back(0 - b);

aValues.push_back(LeastAbsoluteRemainder(inverseCipherText[i] + b *
privateKey.r[i], privateKey.q));
                break;
            }
        }
    }
    return std::make_tuple(aValues, bValues);
}

// Computes the Partial Derivatives of a Single Polynomial over All
Variables
std::vector<ld> ComputePartialDerivatives(std::vector<ll> coefficients,
std::vector<ld> x, ll degree) {
    // Only works for Quadratic Polynomial Equations
    std::vector<ld> xCopy = x;
    xCopy.push_back(1.0); // Pushing the Constant Term

    std::vector<ld> partialDerivatives;
    for(ll i = 0; i < x.size(); ++i) { // Only Compute the Partial
Derivatives w.r.t. the Variables and not the Constant Term
        ld result = 0.0;
        for(ll j = 0, l = 0; j < xCopy.size(); ++j) {
            for(ll k = 0; k < xCopy.size(); ++k, ++l) {
                if(j == k) {
                    if(i == j) {
                        result += 2.0 * coefficients[l] * xCopy[j]; //
derivative of xj^2 w.r.t. xj
                    } else {
                        result += 0.0; // derivative of xj^2 w.r.t. xi
                    }
                } else {
                    if(i == j) {
                        result += coefficients[l] * xCopy[k]; // derivative
of xj.xk w.r.t. xj
                    } else if(i == k) {
                        result += coefficients[l] * xCopy[j]; // derivative
of xj.xk w.r.t. xk
                    } else {
                        result += 0.0; // derivative of xj.xk w.r.t. xi
                    }
```

```
                }
            }
        }
        partialDerivatives.push_back(result);
    }

    return partialDerivatives;
}

// Computes the Jacobian of the Polynomial System
std::vector<std::vector<ld>>
ComputeJacobianMatrix(std::vector<std::vector<ll>> function,
std::vector<ld> x, ll degree) {
    std::vector<std::vector<ld>> jacobian;
    for(ll i = 0; i < function.size(); ++i) {
        jacobian.push_back(ComputePartialDerivatives(function[i], x,
degree));
    }
    return jacobian;
}

// Returns the Transpose of a Matrix
std::vector<std::vector<ld>> MatrixTranspose(std::vector<std::vector<ld>>
mat) {
    std::vector<std::vector<ld>> transpose;
    for(ll i = 0; i < mat[0].size(); ++i) {
        std::vector<ld> row;
        for(ll j = 0; j < mat.size(); ++j) {
            row.push_back(mat[j][i]);
        }
        transpose.push_back(row);
    }
    return transpose;
}

// Compute Norm of a Vector = 0.5 * (vec[1]^2 + vec[2]^2 + ... + vec[n]^2)
ld ComputeNorm(std::vector<ld> vec) {
    ld norm = 0.0;
    for(ll i = 0; i < vec.size(); ++i) {
        norm += vec[i] * vec[i];
    }
    norm *= 0.5;
    return norm;
}

// Returns the Max Absolute Component of a Vector
ld MaxAbsoluteComponent(std::vector<ld> vec) {
```

```
    ld max = 0.0;
    for(ll i = 0; i < vec.size(); ++i) {
        max = std::abs(vec[i]) > max ? std::abs(vec[i]) : max;
    }
    return max;
}

// Solve the Non Linear Polynomial Equation System using the
Levenberg-Marquardt Method
std::vector<ll> SolveNonLinearEquationSystem(PrivateKey privateKey,
std::vector<ll> aValues, std::vector<ll> bValues, ll degree) {

    ld alpha = ALPHA; // Search Parameter
    ld beta = BETA; // Search Parameter
    ld gamma = GAMMA; // Search Parameter

    std::vector<std::vector<ll>> h; // H(x) = Concatenation of Phi(x) -
aValues and Psi(x) - bValues
    for(ll i = 0; i < privateKey.phi.size(); ++i) {
        h.push_back(privateKey.phi[i]);
        h[h.size() - 1][privateKey.phi[i].size() - 1] -= aValues[i]; //
Phi(x) - aValues
    }
    for(ll i = 0; i < privateKey.psi.size(); ++i) {
        h.push_back(privateKey.psi[i]);
        h[h.size() - 1][privateKey.psi[i].size() - 1] -= bValues[i]; //
Psi(x) - bValues
    }

    std::vector<ll> decryptedMessage;

    bool flag = false;
    ld lowLimit = - privateKey.l / 2.0;
    ld upLimit = privateKey.l / 2.0;
    std::vector<ld> x0;
    do {
        x0.clear();
        x0 = GenerateRandom(privateKey.n, lowLimit, upLimit); // Initial
Random Guess in Range [-l / 2, l / 2]
        std::vector<std::vector<ld>> hResult =
{ComputePolynomialSystemOutput(h, x0, degree)}; // Computing h(x0) : (1 x
2n)
        std::vector<std::vector<ld>> hJacobian = ComputeJacobianMatrix(h,
x0, degree); // Computing Jacobian of h(x) at x0 : (2n x n)
        std::vector<std::vector<ld>> negativeIdentityMatrix = {{-1.0}}; //
Negative Identity Matrix : (1 x 1)
```

```
        std::vector<std::vector<ld>> e =
MatrixMultiplication(MatrixMultiplication(negativeIdentityMatrix,
hResult), hJacobian); // e = -h(x0) * Jh(x0) : (1 x n)
        std::vector<std::vector<ld>> s =
MatrixMultiplication(MatrixTranspose(hJacobian), hJacobian); // s =
Jh(x0)^T * Jh(x0) : (n x n)
        std::vector<std::vector<ld>> d0 = MatrixMultiplication(e,
MatrixInverse(s)); // d0 * s = e : (1 x n)
        std::vector<std::vector<ld>> ed0Product = MatrixMultiplication(e,
MatrixTranspose(d0)); // e * d0^T : (1 x 1)

        ld t0 = 0.0; // Step Size
        std::vector<ld> x1; // x1 = x0 + t0 * d0
        std::vector<std::vector<ld>> hResultNew; // h(x1)
        ld thetaOldX = ComputeNorm(hResult[0]); // theta(x0) = Norm of
h(x0)
        ld thetaNewX = 0.0; // theta(x1)
        for(ll i = 0; i >= 0; ++i) {
            ld stepSize = std::pow(beta, i);
            x1.clear();
            for(ll j = 0; j < x0.size(); ++j) {
                x1.push_back(x0[j] + stepSize * d0[0][j]);
            }

            hResultNew.clear();
            hResultNew = {ComputePolynomialSystemOutput(h, x1, degree)};
            thetaNewX = ComputeNorm(hResultNew[0]);
            ld delta = (-alpha) * stepSize * ed0Product[0][0];
            if(thetaNewX - thetaOldX <= delta) {
                t0 = stepSize;
            } else {
                continue;
            }

            std::vector<ld> deltaX; // x1 = x0 + deltaX
            for(ll i = 0; i < x0.size(); ++i) {
                deltaX.push_back(x1[i] - x0[i]);
            }
            ld maxComponent = MaxAbsoluteComponent(deltaX);
            if(maxComponent < gamma) {
                break;
            }
        }

        decryptedMessage.clear();
        for(ll i = 0; i < x1.size(); ++i) {
            decryptedMessage.push_back((ll) std::round(x1[i]));
```

```
        }

        std::vector<ll> hVerify = ComputePolynomialSystemOutput(h,
decryptedMessage, degree);
        flag = true;
        for(ll i = 0; i < hVerify.size(); ++i) {
            if(hVerify[i] != 0) {
                flag = false;
                break;
            }
        }
    } while(!flag);

    return decryptedMessage;
}

// Write Public Key to File
void WritePublicKeyToFile(PublicKey publicKey, std::string filename) {

    std::ofstream publicKeyFile ("PublicKey.txt");

    publicKeyFile << "F:\n";
    for(ll i = 0; i < publicKey.F.size(); ++i) {
        for(ll j = 0; j < publicKey.F[i].size(); ++j) {
            publicKeyFile << publicKey.F[i][j] << " ";
        }
        publicKeyFile << "\n";
    }

    return;
}

// Write Private Key to File
void WritePrivateKeyToFile(PrivateKey privateKey, std::string filename) {

    std::ofstream privateKeyFile (filename);

    privateKeyFile << "n:\n" << privateKey.n << "\n\n";

    privateKeyFile << "l:\n" << privateKey.l << "\n\n";

    privateKeyFile << "lg:\n" << privateKey.lg << "\n\n";

    privateKeyFile << "Phi:\n";
    for(ll i = 0; i < privateKey.phi.size(); ++i) {
        for(ll j = 0; j < privateKey.phi[i].size(); ++j) {
            privateKeyFile << privateKey.phi[i][j] << " ";
```

```
        }
        privateKeyFile << "\n";
    }

    privateKeyFile << "\nMPhi:\n" << privateKey.mPhi << "\n\n";

    privateKeyFile << "Psi:\n";
    for(ll i = 0; i < privateKey.psi.size(); ++i) {
        for(ll j = 0; j < privateKey.psi[i].size(); ++j) {
            privateKeyFile << privateKey.psi[i][j] << " ";
        }
        privateKeyFile << "\n";
    }

    privateKeyFile << "\nMPsi:\n" << privateKey.mPsi << "\n\n";

    privateKeyFile << "r Values:\n";
    for(ll i = 0; i < privateKey.r.size(); ++i) {
        privateKeyFile << privateKey.r[i] << " ";
    }
    privateKeyFile << "\n\n";

    privateKeyFile << "T:\n";
    for(ll i = 0; i < privateKey.T.size(); ++i) {
        for(ll j = 0; j < privateKey.T[i].size(); ++j) {
            privateKeyFile << privateKey.T[i][j] << " ";
        }
        privateKeyFile << "\n";
    }

    privateKeyFile << "\nq:\n" << privateKey.q << "\n";

    return;
}

// Generate Public Private Key Pair
std::pair<PublicKey, PrivateKey> GenerateKeyPair(ll n, ll l, ll lg, ll
degree) {

    std::vector<std::vector<ll>> phiCoefficients = GenerateCoefficients(n,
n, degree, lg); // Coefficients for the Phi Polynomial System
    std::vector<std::vector<ll>> psiCoefficients = GenerateCoefficients(n,
n, degree, lg); // Coefficients for the Psi Polynomial System

    ll mPhi = GetMaxInCodomain(phiCoefficients, n, degree, l); // Largest
Value in Codomain of Phi
    ll mPsi = GetMaxInCodomain(psiCoefficients, n, degree, l); // Largest
```

```
Value in Codomain of Psi

    ll q; // Large Prime q
    std::vector<ll> rValues; // r Values
    std::tie(q, rValues) = GetRValues(n, mPhi, mPsi); // r Values and Prime
q

    std::vector<std::vector<ll>> centralMapG =
GetCentralMap(phiCoefficients, psiCoefficients, rValues, q, n); // The
Central Map G

    std::vector<std::vector<ll>> affineT = GetAffineTransformation(n, q);
// Random Affine Transformation T

    std::vector<std::vector<ll>> polynomialMapF =
GetFinalPolynomialMap(affineT, centralMapG, q); // Final Polynomial Map
and Public Key F

    // Save to publicKey
    std::cout << "Saving Public Key...\n";
    PublicKey publicKey;
    publicKey.F = polynomialMapF;
    WritePublicKeyToFile(publicKey, "PublicKey.txt");

    // Save to privateKey
    std::cout << "Saving Private Key...\n";
    PrivateKey privateKey;
    privateKey.n = n;
    privateKey.l = l;
    privateKey.lg = lg;
    privateKey.phi = phiCoefficients;
    privateKey.mPhi = mPhi;
    privateKey.psi = psiCoefficients;
    privateKey.mPsi = mPsi;
    privateKey.r = rValues;
    privateKey.T = affineT;
    privateKey.q = q;
    WritePrivateKeyToFile(privateKey, "PrivateKey.txt");

    return std::make_pair(publicKey, privateKey);
}

// Encrypting the Input Message Using the Public Key
std::vector<ll> EncryptMessage(PublicKey publicKey, std::vector<ll>
message, ll degree) {
    std::vector<ll> cipherText = ComputePolynomialSystemOutput(publicKey.F,
message, degree);
```

```
    return cipherText;
}

// Decrypting the Cipher Text Using the Private Key
std::vector<ll> DecryptCipherText(PrivateKey privateKey, std::vector<ll>
cipherText, ll degree) {
    // Brings the Cipher Text to the Finite Field Fq
    for(ll i = 0; i < cipherText.size(); ++i) {
        cipherText[i] = Modulo(cipherText[i], privateKey.q);
    }

    // Computing T^-1(c)
    std::vector<std::vector<ll>> inverseT = MatrixInverse(privateKey.T,
privateKey.q);
    std::vector<ll> inverseCipherText =
DeVectorize(MatrixMultiplication(inverseT, Vectorize(cipherText))); //
ComputePolynomialSystemOutput() not used as T is a system of linear
polynomials

    // Computing the RHS of Phi(x) and Psi(x)
    std::vector<ll> aValues;
    std::vector<ll> bValues;
    std::tie(aValues, bValues) = GetABValues(privateKey,
inverseCipherText); // Get the RHS of Phi(x) and Psi(x)

    // Solving the Non Linear Polynomial System using Levenberg-Marquardt
Method
    std::vector<ll> decryptedMessage =
SolveNonLinearEquationSystem(privateKey, aValues, bValues, degree);

    return decryptedMessage;
}

int main() {

    std::srand(std::time(0)); // Setting a New Seed Value on Every Run

    std::cout << "Enter Security Parameters (n L Lg):\n";
    ll n; // Number of Variables in System of Polynomial Equations
    ll l; // Odd Positive Integer
    ll lg; // Odd Positive Integer
    ll degree = DEGREE; // Maximum Degree of the Monomials in the
Polynomial System
    std::cin >> n >> l >> lg;

    /**
     * ################################## Generating Public Private Key
```

```
Pair ################################
    */

   std::cout << "\nGenerating Key Pair...\n";
   auto startTimeKG = std::chrono::high_resolution_clock::now();

   std::pair<PublicKey, PrivateKey> keyPair = GenerateKeyPair(n, l, lg,
degree);
   PublicKey publicKey = keyPair.first;
   PrivateKey privateKey = keyPair.second;

   auto stopTimeKG = std::chrono::high_resolution_clock::now();
   auto durationKG =
std::chrono::duration_cast<std::chrono::milliseconds>(stopTimeKG -
startTimeKG);
   double execTime = durationKG.count() / 1000.0;
   std::cout << "Time taken to generate Key Pair = " << execTime <<
"s\n\n";

   /**
    * ################################ Encrypting the Message using the
Public Key ################################
    */

   std::cout << "Enter Message to Encrypt (Enter " << n << " numbers in
the range [" << std::floor(-l / 2.0) + 1 << ", " << std::floor(l / 2.0) <<
"]):\n";
   std::vector<ll> message(n); // Input to Encrypt
   for(ll i = 0; i < n; ++i) {
       std::cin >> message[i];
   }

   std::cout << "\nEncrypting Input Message...\n";
   auto startTimeEnc = std::chrono::high_resolution_clock::now();

   std::vector<ll> cipherText = EncryptMessage(publicKey, message,
degree);

   auto stopTimeEnc = std::chrono::high_resolution_clock::now();
   auto durationEnc =
std::chrono::duration_cast<std::chrono::nanoseconds>(stopTimeEnc -
startTimeEnc);
   execTime = durationEnc.count() / 1000.0;

   std::cout << "Encrypted Cipher Text: ";
   for(ll i = 0; i < cipherText.size(); ++i) {
       std::cout << cipherText[i] << " ";
```

```
    }
    std::cout << "\n";
    std::cout << "Time taken to Encrypt Message = " << execTime <<
"us\n\n";

    /**
     * ################################## Decrypting the Cipher Text using
the Private Key ##################################
     */

    std::cout << "Decrypting Cipher Text...\n";
    auto startTimeDec = std::chrono::high_resolution_clock::now();

    std::vector<ll> decryptedMessage = DecryptCipherText(privateKey,
cipherText, degree);

    auto stopTimeDec = std::chrono::high_resolution_clock::now();
    auto durationDec =
std::chrono::duration_cast<std::chrono::milliseconds>(stopTimeDec -
startTimeDec);
    execTime = durationDec.count() / 1000.0;

    std::cout << "Decrypted Message: ";
    for(ll i = 0; i < decryptedMessage.size(); ++i) {
        std::cout << decryptedMessage[i] << " ";
    }
    std::cout << "\n";
    std::cout << "Time taken to Decrypt Message = " << execTime << "s\n";

    return 0;
}
```