Software Implementation of EMBLEM and R.EMBLEM

Contents

In this document, we give details of our implementation of EMBLEM and R.EMBLEM. Data types for each element in matrix used in EMBLEM and coefficient of polynomial used in R.EMBLEM is 32-bit `int`.

1 Modulus Operation

For modulus q in EMBLEM, we set q as power of two (i.e, $q = 2^k$). Therefore, reduction modulo q is as simple as ignoring $32 - (\log_2(q) - 1)$ most significant bits. For 24-bit q, reduction modulo q is computed as follows:

$$x = x \ \mathsf{0xffffff} (0 \leq x < q^2) \tag{1}$$

We set q as prime in R.EMBLEM to apply NTT-based polynomial multiplication. Let q be prime such that $q \equiv 1 \mod 2n$, where n is a dimension of the ring. In order to optimize reduction modulo q, we use modified form of Barrett reduction. Let $q = 12289$. Then $q < 2^{14} = 16384$. Let x be the number to be reduced, $0 \leq x < q^2$. We first obtain the quotient of x divided by 2^{14}. This can be done by simple bitwise operation:

$$c = x \gg 14 \tag{2}$$

Then we multiply by 12289 and subtract to x to obtain the result:

$$x = x - c \times 12289 \tag{3}$$

Since this reduction does not give exact result, $x \mod q$, additional reduction must be performed in order to get the full result. For efficiency, such final reduction only occur at the end of NTT operation. After the end of NTT or INTT in optimize implementation or reference implementation, we simply divide by q and add q if the result is negative to obtain the final result.

2 Data Generation

Matrices in EMBLEM or polynomials in R.EMBLEM are chosen randomly from the uniform distribution or in specific distributions. Detailed description is given as follows.

2.1 Matrix Generation

• **Public key matrix.** For public key matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, we call `randombytes()` function for each $A_{i,j}$, the i-th row and j-th column entry of $\mathbf{A}$, then reduce to modulo q.

• **Secret key matrix.** For secret key matrix $\mathbf{X} \in [-B, B]^{n \times k}$, we first call `randombytes()` function to generate octet string $\mathsf{rnd}[\,]$ of length $n \times k$ and obtain the remainder of the value when divided by $2B + 1$ and subtract $B + 1$ to obtain values in $[-B, B]$. If $B = 1$, then we can obtain the i-th row and j-th column of $\mathbf{X}$, denoted by $X_{i,j}$, by $X_{i,j} \leftarrow (\mathsf{rnd}[i, j]\%3) - 2$, and the result will be in [-1,1].

• **Sampling function.** In EMBLEM.CPA, the sampling function takes a 256-bit string r as an input and outputs $\mathbf{R} \in [-B, B]^{m \times v}$ and $(\mathbf{E_1}, \mathbf{E_2}) \in \mathcal{GD}_s^{v \times (n+k)}$. $\mathbf{E_1}$ and $\mathbf{E_2}$ are sampled from discrete Gaussian distribution, and we will explain this in **Error matrix** below.

The matrix $\mathbf{R}$ is generated as follows. For the simplicity, we assume that $B = 1$. First, compute the hash function with 256-bit string r as input, where the data type of the digest is `int`. Let *len* be the number of `int` arrays of the digest. Starting at the first index of digest array $\mathsf{digest}[i]$, each

element is divided by 3 and its remainder minus 2 is stored as matrix element of $\mathbf{R}$. For the i-th row and j-th column of R, denoted by $R_{i,j}$,

$$R_{i,j} \leftarrow (\mathsf{digest}[i]\%3) - 2 \tag{4}$$

Then the element of digest array is updated with a quotient divided by 3.

$$\mathsf{digest}[i] \leftarrow \mathsf{digest}[i]/3 \tag{5}$$

When $\mathsf{digest}[i]$ becomes zero, move to the next index and repeat the process. When all the elements have completed the process, we add 1 to r and compute the hash function with $r+1$ as input, and repeat the process again. When all elements of $\mathbf{R}$ are set, the process is terminated. At the end of the process, the original r is hashed again, and 32-bit of the digest is used as a *seed* to generate the same errors in EMBLEM. Pseudocode for generating matrix $\mathbf{R}$ and the *seed* is as follows:

Algorithm 1: Pseudocode for generating the matrix $\mathbf{R}$ and the *seed*

Input : 256-bit string r
Output: matrix $R \in [-1,1]^{m \times v}$, *seed*

```
cnt ← 0;
d ← r;
while cnt < m × v do
    digest ← H(d)
    for I = 0 to len do
        while digest[i] ! = 0 do
            R[cnt] ← (digest[i]%3) - 2;
            digest[i] ← digest[i]/3;
            cnt++;
        end
    end
    d[0]++;
end
digest ← H(r);
return digest[0]
```

• **Random matrix.** Elements of error matrix $\mathbf{E}$ are sampled from discrete Gaussian distribution. Let Sample_CDT() be Gaussian sampler. Then for each element $E_{i,j}$, the i-th row and j-th column of $\mathbf{E}$,

$$E_{i,j} \leftarrow \mathsf{Sample\_CDT}() \tag{6}$$

2.2 Polynomial Generation

• **Public key polynomial.** For public key polynomial $a \in \mathbb{Z}_q[x]/\langle x^n+1\rangle$, we call `randombytes()` functions for each coefficient of a and reduce to modulo q.

• **Secret key polynomial.** For secret key polynomial x, we call `randombytes()` function to generate octet string $\mathsf{rnd}[\ ]$ of length N and obtain the remainder of the value when divided by $2B+1$ and subtract $B+1$ to obtain values in $[-B, B]$. If $B = 1$, then we can obtain the i-th coefficient of x, denoted by x_i, by $x_i \leftarrow (\mathsf{rnd}[i]\%3) - 2$, and the result will be in [-1,1].

• **Sampling function.** In R.EMBLEM.CPA, the sampling function takes a 256-bit string z as an input and outputs the polynomials r, e_1, and e_2. The coefficients of e_1 and e_2 are sampled from discrete Gaussian distribution, and we will explain this in **Error polynomial** below.

The polynomial r is generated as follows. For the simplicity, we assume that $B = 1$. First, compute the hash function with 256-bit string z as input, where the data type of the digest is `int`. Let *len* be th number of `int` arrays of the digest. Starting at the first index of digest array $\mathsf{digest}[i]$, each element is divided by 3 and its remainder minus 2 is stored as the coefficient of r. For the i-th coefficient of r, denoted by r_i,

$$r_i \leftarrow (\mathsf{digest}[i]\%3) - 2 \tag{7}$$

Then the element of digest array is updated with a quotient divided by 3.

$$\mathsf{digest}[i] \leftarrow \mathsf{digest}[i]/3 \tag{8}$$

When $\mathsf{digest}[i]$ becomes zero, move to the next index and repeat the process. When all the elements have completed the process, we add 1 to z and compute the hash function with $z + 1$ as input, and repeat the process again. When all coefficients of r are set, the process is terminated. At the end of the process, the original z is hashed again, and 32-bit of the digest is used as a *seed* to generate the same error polynomials in EMBLEM.

• **Error polynomial.** Coefficients of an error polynomial e are sampled from discrete Gaussian distribution. Let Sample_CDT() be Gaussian sampler. Then for each coefficient e_i, the i-th coefficient of e,

$$e_i \leftarrow \mathsf{Sample\_CDT}() \tag{9}$$

3 Discrete Gaussian Sampler

We use inversion sampling method for insanitation of discrete Gaussian sampler. For $\sigma = 3$, cumulative density table T is created from probability density functions. We sample 9 bits uniformly at random, corresponding to a uniform random integer $x \in [0, 511]$ and additional 1 bit to determine the sign of the sampled value. For $\sigma = 25$, we sample 11 bits uniformly at random, corresponding to a uniform random integer $x \in [0, 2047]$ and additional 1 bit to determine the sign of the sampled value. Cumulative distribution table (CDT) of length 16 and 54 is used for $\sigma = 3$ and $\sigma = 25$, respectively.

4 Matrix Multiplication

For matrix multiplication $\mathbf{A} \times \mathbf{B}$ in reference implementation, we use row-wise matrix multiplication in order to reduce cache miss. That is, we consider $\mathbf{B}$ as in transposed format and execute matrix multiplication as row by row. For optimized implementation, matrix multiplication is done by removing pointers.

5 Number Theoretic Transform

The Number Theoretic Transform (NTT) is the specialized version of discrete Fourier transform in finite field $F = \mathbb{Z}_p$, integers modulo prime p, or ring. NTT used in R.EMBLEM is based on Cooly-Tukey butterfly, and inverse NTT (INTT) is based on Gentleman-Sande butterfly. Let ψ

be a primitive $2n$-th root of unity, $\psi^{1024} \equiv 1 \mod q$ such that $\psi^2 = w$ where $w^{512} \equiv 1 \mod q$. We generate ψ and ψ^{-1} table in bit reversed order. For $p = 12289$, we used $\psi = 1987$, and for $p = 40961$ we used $\psi = 1044$. Since NTT and INTT table is known to public, it suffices to store public and private keys in NTT domain for efficiency.