ANALYSIS OF THE CRYSTALS-KYBER IMPLEMENTATION

ENCUENTRO DE ÁLGEBRA COMPUTACIONAL Y APLICACIONES

Diego Rojas Rodríguez[1,2*], Luis Hernández Encinas [1†]

[1]Institute of Physical and Information Technologies (ITEFI), Spanish National Research Council (CSIC)
[2]Complutense University of Madrid (UCM)
*diegroja@ucm.es, †luis.h.encinas@csic.es
†0000-0001-6980-2683

EACA 2024

1. INTRODUCTION AND KYBER IMPLEMENTATION

CRYSTALS-Kyber (or just Kyber) is a lattice-based KEM (Key Encapsulation Mechanism) whose security relies on the hardness of the Ring Learning With Errors (RLWE) problem. RLWE is a version of the Learning With Errors (LWE) problem defined on the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where $n \leq 256$ is the polynomial degree and $q = 3329 = 13 \cdot 28 + 1$ is a prime number.

Kyber is composed by two main primitives: a Public Key Encryption (PKE) with its corresponding key generation, cipher and decipher algorithms, and a set of functions that conform the KEM via the Fujisaki-Okamoto transform resulting key generation, encapsulation, and decapsulation algorithms. Additionally, PKE uses several functions, among which we highlight *Compress* (and *Decompress*) and NTT (Number Theoretic Transform). The reference implementation in Kyber's proposal was done in language C. This decision might be motivated due to C is the fastest programming language when coding at a low level, even though it may carry some side effects given the lack of native operations for dealing with polynomials.

2. *COMPRESS* AND *DECOMPRESS* FUNCTIONS

The *Compress* and *Decompress* functions are defined as follows:

$$Compress_q(x,d) = \frac{2^d}{q}x \mod 2^d, \quad Decompress_q(x,d) = \frac{q}{2^d}x \mod q,$$

where q is the prime used in the definition of the ring R_q, d is an integer parameter determined by the security of Kyber's implementation and x is the coefficient to be compressed. Both functions aim to discard some bits of the keys whose influence is negligible, and thus shortening key size and as they have similar properties. Some operations, not defined natively in C, needed to be adapted giving raise to the following disparities (see Figure 1):

1. The function compresses all coefficients of the polynomial at once.
2. Each signed polynomial coefficient is mapped to its positive representative in binary (C2 operation).
3. Multiplication by 2^d is efficiently performed by arithmetically shifting to the left d positions.
4. Kyber_Q/2 is added and later divided by q (integer division)
5. The resulting number is then operated with an AND (&) with the number 15 (1111), equivalent to performing the $\mod 2^4$ operation.

Figure 1 – poly_compress function in poly.c

```
void poly_compress(uint8_t r[KYBER_POLYCOMPRESSEDBYTES], const poly *a)
{
  unsigned int i,j;
  int16_t u;
  uint8_t t[8];

#if (KYBER_POLYCOMPRESSEDBYTES == 128)
  for(i=0;i<KYBER_N/8;i++) {
    for(j=0;j<8;j++) {
      // map to positive standard representatives
      u  = a->coeffs[8*i+j];
      u += (u >> 15) & KYBER_Q; //Si u < 0 => u += 1
      t[j] = ((((uint16_t)u << 4) + KYBER_Q/2)/KYBER_Q) & 15;
    }
```

The result after performing one iteration of the loop is equivalent to the function:

$$poly\_compress(x,d) = |x|\frac{2^d}{q} + \frac{q}{2} \mod 2^d.$$

3. NTT (NUMBER THEORETIC TRANSFORM)

It is known that an essential operation and one of the most time consuming for the RLWE problem is the polynomial multiplication over R_q. To improve its efficiency the NTT (Number Theoretic Transform) is used. To multiply two polynomials over R_q they are transformed to the NTT domain, then operated and finally transformed back to the original domain. We have $x^{256} + 1 = \prod_{i=0}^{127} x^2 - \zeta^{2i+1} = \prod_{i=0}^{127} x^2 - \zeta^{2br_7(i)+1}$ where each ζ^i represents a 256-th root of unity. Using the NTT and its inverse, NTT^{-1}, it is possible to efficiently compute the product of two polynomials in R_q, given that $\text{NTT}^{-1}(\hat{f} \cdot \hat{g}) = \text{NTT}^{-1}(\hat{h}) = h = f \cdot g$. The implementation of the NTT is shown in Figure 2, where the roots of unity are previously calculated and stored in the matrix zetas.

Moreover, we can see that in the mathematical definition, as well as in the implementation, the process is iterative. In Figure 2, len represents the distance to which the operations are performed, this is why we define $k = 7 - \log_2(len)$.

3. NTT (NUMBER THEORETIC TRANSFORM) (CONT.)

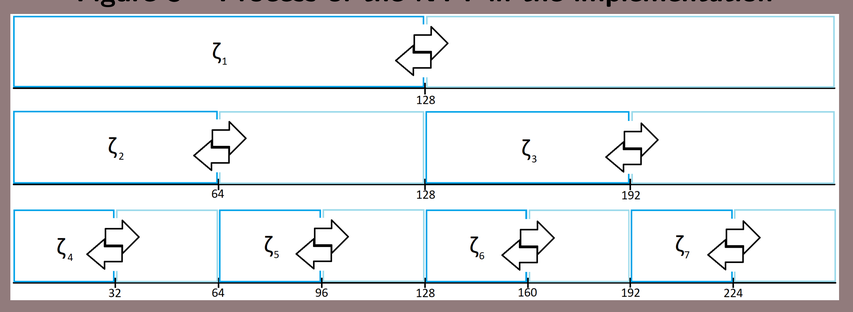
Figure 2 – NTT process in the implementation in C

```
void ntt(int16_t r[256]) {
  unsigned int len, start, j, k;
  int16_t t, zeta;

  k = 1;
  for(len = 128; len >= 2; len >>= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas[k++];
      for(j = start; j < start + len; j++) {
        t = fqmul(zeta, r[j + len]);
        r[j + len] = r[j] - t;
        r[j] = r[j] + t;
```

Given that $len = 2^7 - k$, the further in the process the smaller is the operating distance. Variable $start$ represents over which index of the vector len steps are made. Finally, j represents the index of the vector over which each individual operation is executed, always between j and $j + len$. The $start$ takes values which are multiples of len from 0 to 256, meaning the nested loop is executed 2^k times for each value of len. The last loop performs the multiplication of $zeta$ and the corresponding value $r[j + len]$, in the interval $[start + len, 2 \cdot len]$ with $len - start$ iterations. Lastly, the transform adds each value to its symmetrical over len and subtracts it to the indexed value in the previous step. Figure 3 shows how the loop is executed in the implementation.

Figure 3 – Process of the NTT in the implementation



It can be appreciated how each iteration of k is represented with a row and each value of start with a colour. Variable j would be the one iterating over each coloured block, indexing one by one $j = j + 1$. After a coloured block, $start$ is increased $start = j + len$ until $start = start + 2 \cdot len$ that will start the next row. The arrows point which blocks are swapped, for each index i with its representative in the other block (starting from 0 in each block). Next step would be proposing a mathematical function that execute the process of the implementation in order to compare it to the original NTT. Representing each index of the vector by i and each iteration of the loop by k, knowing that $len = 2^7 - k$, we define the auxiliary recursive function $\varphi_{i,k}$ as follows:

$$\varphi_{i,0} = f_i + \zeta f_{i+128}, 0 \leq i < 128, \quad \varphi_{i-128} - \zeta f_i, 128 \leq i < 256,$$

$$\varphi_{i,k} = \varphi_{i,k-1} + \zeta^{2^k + \frac{i}{2^{8-k}}}\varphi_{i+2^{7-k},k-1}, \quad \varphi_{i,k} = \varphi_{i-2^{7-k},k-1} + \zeta^{2^k + \frac{i}{2^{8-k}}}\varphi_{i,k-1}.$$

Finally, the starting call would be $\hat{f}_i = \varphi_{i,6}$. This way the base cases would represent the first iteration of the loop, when $len = 128$, and the following iterations, $0 < k < 7$, are determined by their parity in each iteration of $start$, or $start + len$. That is, $\frac{i}{2^{7-k}} \mod 2$ represents the division of the vector in intervals of length len and the $\mod 2$ indicates whether its the left or right side. Each of the 128 $zeta$ values is represented given that k and i can be at most 6 and 255 respectively. Ultimately, one can appreciate that the definition in the code greatly differs from the original one. This could be one of the main reasons why side-channel attacks usually target the NTT in Kyber.

4. CONCLUSIONS

The differences shown between the mathematical definition, presented in the reference document of Kyber, and the official implementation, made by the original authors, are non negligible. These differences could lead to vulnerabilities that allow side-channel attacks that exploit them. We have highlighted the differences we spotted in our first analysis.

ACKNOWLEDGEMENTS

This work was supported in part by P2QProMeTe project (PID2020-112586RB-I00), funded by MCIN/AEI/10.13039/501100011033, and in part by QURSA project (TED2021-130369B-C33) funded by MCIN/AEI/10.13039/501100011033, both co-funded by the European Union "NextGenerationEU"/PRTR.