



Module-Lattice-based Key-Encapsulation Mechanism Standard

Autores: *Daniel Ruiz Guirales, Andres Penagos Betancur*

Laboratorio de Electrónica Digital 3

Departamento de Ingeniería Electrónica y de Telecomunicaciones

Universidad de Antioquia

Resumen

En el ámbito de la seguridad informática, un mecanismo llamado KEM es esencial para que dos partes puedan establecer una clave secreta compartida a través de una red pública. Este mecanismo, conocido como ML-KEM, se basa en algoritmos complejos que abordan problemas de seguridad computacional. Se ha demostrado que ML-KEM es resistente incluso contra ataques de computación cuántica, lo que lo hace crucial en un mundo digital en constante evolución. Además, este estándar presenta diferentes conjuntos de parámetros, cada uno con su propio equilibrio entre seguridad y rendimiento, lo que proporciona flexibilidad a los usuarios en función de sus necesidades específicas de seguridad.

Palabras clave: Algebra Lineal, Cartografía, Encriptación, Generación de Clave, Seguridad Computacional.

Introducción

Este estándar aborda el propósito y alcance del Mecanismo de Encapsulación de Clave basado en Módulo-Reticulado, o ML-KEM. Un ML-KEM es un conjunto de algoritmos diseñados para establecer una clave secreta compartida entre dos partes que se comunican a través de un canal público. Se destaca que los esquemas de establecimiento de clave actuales aprobados por NIST pueden ser vulnerables a ataques de computadoras cuánticas avanzadas, lo que ha generado la necesidad de alternativas seguras como ML-KEM. Este mecanismo se deriva de una versión específica del CRYSTALS-KYBER KEM, presentado en el proyecto de estandarización de criptografía post-cuántica de NIST. El estándar también de-

talla los algoritmos y conjuntos de parámetros de ML-KEM, con el objetivo de proporcionar información suficiente para su implementación y validación. Además, se presentan tres conjuntos de parámetros de ML-KEM, cada uno con sus propios compromisos entre la seguridad y el rendimiento, todos aprobados para proteger sistemas de comunicación no clasificados del Gobierno Federal de los EE. UU.

Marco teórico

Para el apartado teórico de esta implementación, es fundamental definir los parámetros de entrada y salida de cada algoritmo, así como su utilidad y propósito.

Dentro de este estándar, contamos con 12 Algoritmos base que desempeñan funciones fundamentales. Estos algoritmos base incluyen:

Función BitstoBytes

La función **BitstoBytes** se emplea para transformar un conjunto de bits en un conjunto de bytes. Esta conversión resulta útil en escenarios donde se requiere representar datos en forma binaria de manera más eficiente, o cuando se necesita manipular datos a nivel de bits en lugar de bytes.

Los parámetros de esta función son:

- **b:** Un puntero que señala un arreglo de enteros sin signo que representa el conjunto de bits que se desea convertir a bytes.
- **b-len:** Un entero que indica la longitud del arreglo de bits b.
- **B:** Un puntero que apunta a un arreglo de caracteres sin signo donde se almacenarán los bytes resultantes de la conversión. Es importante destacar que este arreglo debe tener suficiente espacio

para contener los bytes generados a partir de los bits en b .

Función `BytestoBits`

La función `BytestoBits` descompone un conjunto de bytes en un conjunto de bits.

Los parámetros de esta función son:

- **B**: Un puntero que apunta a un arreglo de caracteres sin signo que contiene los bytes que se desean convertir en bits.
- **b-len**: Un entero que indica la longitud del arreglo de bytes B .
- **b**: Un puntero que señala un arreglo de enteros sin signo donde se almacenarán los bits resultantes de la conversión. Es crucial destacar que este arreglo debe tener suficiente capacidad para contener los bits generados a partir de los bytes en B .

Funciones `ByteEncode` y `ByteDecode`

Los algoritmos `ByteEncode` y `ByteDecode` se emplean para la serialización y deserialización de arreglos de enteros módulo m . Todos los arreglos serializados tendrán una longitud de 256. `ByteEncode` convierte un arreglo de enteros de d bits en un arreglo de $32 * d$ bytes, mientras que `ByteDecode` realiza la operación inversa, transformando un arreglo de $32 * d$ bytes en un arreglo de enteros de d bits.

Función `ByteEncode`

La función `ByteEncode` acepta dos parámetros:

- **const unsigned int *F**: Es un puntero que apunta a un arreglo de enteros sin signo. Este arreglo representa los coeficientes de los polinomios que se desean convertir en su forma serializada.
- **unsigned char *B**: Es un puntero que señala un arreglo de caracteres sin signo. Aquí es donde se almacenarán los bytes resultantes de la serialización de los coeficientes de los polinomios.

Función `ByteDecode`

La función `ByteDecode` tiene dos parámetros:

- **unsigned char *B**: Es un puntero que apunta a un arreglo de caracteres sin signo. Este arreglo contiene los bytes que se desean deserializar, es

decir, los datos que se convertirán de su representación en bytes a su forma original de enteros.

- **const unsigned int *F**: Es un puntero que señala un arreglo de enteros sin signo. Aquí es donde se almacenarán los enteros resultantes de la deserialización de los bytes.

Funciones `Compress` y `Decompress`

La función `Compress` toma un entero x del conjunto de números enteros módulo q y lo asigna a un entero en el conjunto de números enteros módulo 2^d . Realiza este proceso descartando los bits menos significativos de la entrada y ajustando el resultado al entero más cercano en el conjunto 2^d .

La función `Decompress` toma un entero y del conjunto de números enteros módulo 2^d y lo asigna a un entero en el conjunto de números enteros módulo q . Realiza este proceso añadiendo ceros a los bits menos significativos de la entrada y ajustando el resultado al entero más cercano en el conjunto q .

$$\begin{aligned} \text{Compress}_d : \mathbb{Z}_q &\longrightarrow \mathbb{Z}_{2^d} \\ x &\longmapsto \lceil (2^d/q) \cdot x \rceil. \\ \text{Decompress}_d : \mathbb{Z}_{2^d} &\longrightarrow \mathbb{Z}_q \\ y &\longmapsto \lceil (q/2^d) \cdot y \rceil. \end{aligned}$$

La función `Compress` toma un número entero sin signo de 16 bits x como entrada y devuelve un número entero sin signo de 16 bits como salida.

La función `Decompress` recibe un número entero sin signo de 16 bits y como entrada y devuelve un número entero sin signo de 16 bits como salida.

Función `sampleNTT`

La función `sampleNTT` convierte una secuencia de bytes en un polinomio en el dominio de Transformada de Fourier Rápida (NTT).

Argumentos de la función:

- **B**: Un puntero que señala un arreglo de caracteres sin signo que contiene los datos de entrada en formato de bytes.
- **a**: Un puntero que apunta a un arreglo de enteros sin signo donde se almacenarán los resultados de

la transformación NTT. Es importante que este arreglo tenga al menos 256 elementos para contener los resultados completos de la transformación.

Función SamplePolyCBD

La función **SamplePolyCBD** recibe una secuencia de bytes como entrada y genera un polinomio en el anillo de enteros módulo q , donde q es un número primo. Implementa un esquema de muestreo llamado "Centered Binomial Distribution" (CBD), comúnmente utilizado en esquemas de cifrado basados en retículas.

Argumentos de la función:

- **B**: Un puntero que apunta a una secuencia de bytes que contiene la entrada.
- **f**: Un puntero que señala un arreglo de enteros sin signo donde se almacenarán los coeficientes del polinomio resultante.

Función NTT

La función **NTT** opera sobre un polinomio f en el anillo R_q como entrada y produce un elemento f' del anillo T_q , conocido como la representación NTT de f . En términos simples, la NTT convierte un polinomio en R_q en un formato más eficiente para ciertas operaciones, como la multiplicación. Esta transformación preserva la estructura y las propiedades del polinomio original, lo que permite recuperar el polinomio original si es necesario.

La NTT actúa como una traducción especializada que facilita el trabajo con el polinomio para operaciones específicas. Es como cambiar de un idioma a otro para comunicarse de manera más efectiva, asegurando que la información esencial se conserve en el proceso.

Argumentos de la función:

- **f**: Un puntero que apunta a un arreglo de enteros de 16 bits que representa el polinomio de entrada.

Función inverseNTT

La función **inverseNTT** realiza exactamente lo opuesto a la función NTT, el propósito general de este algoritmo es recuperar el polinomio original luego de ser transformado al dominio NTT.

Argumentos de la función:

- **f**: Un puntero que apunta a un arreglo de enteros de 16 bits que representa el polinomio transformado NTT como entrada()

Función BaseCaseMultiplyc0

La función **BaseCaseMultiplyc0** calcula el primer componente c_0 del producto de dos elementos en el anillo T_q utilizando la fórmula $c_0 = (a_0 \cdot b_0 + a_1 \cdot b_1 \cdot \gamma) \text{ mód } q$.

Argumentos de la función:

- **a0**: El primer coeficiente del primer polinomio.
- **a1**: El segundo coeficiente del primer polinomio.
- **b0**: El primer coeficiente del segundo polinomio.
- **b1**: El segundo coeficiente del segundo polinomio.
- **gamma**: El valor de γ , un parámetro.

Función BaseCaseMultiplyc1

La función **BaseCaseMultiplyc1** calcula el segundo componente c_1 del producto de dos elementos en el anillo T_q utilizando la fórmula $c_1 = (a_0 \cdot b_1 + a_1 \cdot b_0) \text{ mód } q$.

Argumentos de la función:

- **a0**: El primer coeficiente del primer polinomio.
- **a1**: El segundo coeficiente del primer polinomio.
- **b0**: El primer coeficiente del segundo polinomio.
- **b1**: El segundo coeficiente del segundo polinomio.

Función MultiplyNTTs

La función **MultiplyNTTs** implementa la multiplicación de dos elementos en el dominio NTT f^\wedge y g^\wedge para obtener el producto h^\wedge . Utiliza las funciones **BaseCaseMultiplyc0** y **BaseCaseMultiplyc1** como subrutinas para realizar la multiplicación en cada una de las 128 coordenadas del dominio NTT.

Argumentos de la función:

- **f_ntt**: Un puntero que señala un arreglo de enteros que representa el primer elemento en el dominio NTT.
- **g_ntt**: Un puntero que apunta a un arreglo de enteros que representa el segundo elemento en el dominio NTT.
- **h_ntt**: Un puntero que indica un arreglo de enteros donde se almacenarán los coeficientes del producto en el dominio NTT.

Estos algoritmos son clave para el funcionamiento de esta implementación ya que forman parte de los algoritmos principales de este estándar. Los algoritmos principales son:

K-PKE KeyGen
 K-PKE Encrypt
 K-PKE Decrypt
 ML-KEM KeyGen
 ML-KEM Encaps
 ML-KEM Decaps

ML-KEM KeyGen

El algoritmo `ML-KEM.KeyGen` es fundamental en la creación de claves para el esquema ML-KEM. A través del Algoritmo 15, este proceso no requiere entrada externa, pero sí aleatoriedad, generando así una clave de encapsulación y otra de desencapsulación. La naturaleza de esta operación es crucial para la seguridad del sistema, ya que mientras la clave de encapsulación puede ser compartida públicamente, la clave de desencapsulación debe mantenerse en privado para garantizar la confidencialidad de la comunicación.

ML-KEM Encaps

El algoritmo `ML-KEM.Encaps` permite la encapsulación de datos dentro del esquema ML-KEM. A través del Algoritmo 16, esta operación requiere una clave de encapsulación como entrada y la generación de aleatoriedad. Como resultado, produce un texto cifrado junto con una clave compartida. Este proceso es esencial para asegurar la confidencialidad de la información transmitida, ya que protege los datos durante su transferencia.

ML-KEM Decaps

El algoritmo `ML-KEM.Decaps` es responsable de la desencapsulación de datos en el esquema ML-KEM. A través del Algoritmo 16, toma una clave de desencapsulación y un texto cifrado como entrada, sin requerir aleatoriedad. Su función es crucial para recuperar el secreto compartido original a partir del texto cifrado, manteniendo la integridad y confidencialidad de la comunicación.

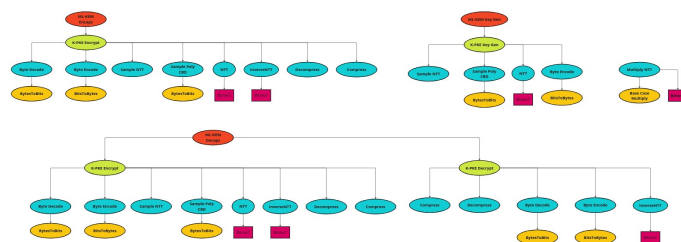
Procedimiento experimental y resultados

Para entender y empezar a aplicar este estándar, es esencial revisar detenidamente cada sección del documento, lo que nos proporcionará un contexto más claro

de lo que se pretende lograr. Después de este paso inicial, nos enfocamos en comprender el Capítulo 2, donde se detalla cómo interpretar el pseudocódigo y los símbolos utilizados en cada algoritmo.

Posteriormente, planteamos una serie de preguntas específicas para cada algoritmo: ¿Qué hace exactamente? ¿Cuál es su propósito? ¿Qué variables requiere como entrada? ¿Qué variables manipula en la salida? Estas preguntas nos ayudan a entender cada algoritmo de manera individual antes de comenzar su implementación en el lenguaje C y realizar pruebas.

En la primera etapa de este proceso, se detalla el procedimiento de comprensión y abstracción necesario para llevar a cabo la implementación. Inicialmente, se elabora un diagrama de flujo que muestra las funciones de cada algoritmo de nivel superior, proporcionando así una visión general de la estructura y la interconexión entre ellos.



Después de completar la implementación del diagrama de flujo, se procede a la redacción del código que corresponde a los algoritmos discutidos en el marco teórico. Dado que cada función dentro del código puede contener múltiples subfunciones, se comienza desde el nivel más básico, teniendo como base el diagrama de flujo con el fin de progresar gradualmente, realizando pruebas específicas a medida que se avanza en el desarrollo.

Para cada función se creó un archivo `.c` para así demostrar el funcionamiento de cada función de manera independiente.

```
void ByteDecoded(const unsigned char *B, unsigned int *F) {
    unsigned int B_len = (N * d + 7) / 8;
    unsigned int b[B_len * 8];
    BytesToBits(B, B_len, b); // Convertir bytes en bits
    // Decodificar los bits en enteros de d bits
    for (int i = 0; i < 256; i++) {
        unsigned int sum = 0;
        for (int j = 0; j < d; j++) {
            sum += b[i * d + j] << j; // Sumar los bits multiplicados por 2^j
        }
        F[i] = sum & k; // Se acaba de agregar
    }
}
```

Cabe destacar que el fragmento de código resaltado se utilizó en varias partes del código debido a que inicialmente no se tenía en cuenta el redondeo y ahora con este +7 en esta expresión se utiliza para realizar un redondeo hacia arriba al dividir el producto $N * d$ por 8. Esto asegura que el resultado de la división sea siempre un número entero mayor o igual que el resultado de la división exacta. Es una técnica común cuando se desea dividir un número entero y asegurarse de que el resultado sea siempre mayor o igual que el resultado real de la división, incluso si hay un residuo.

Durante el desarrollo de la practica, surgió la cuestión sobre la utilidad de implementar operaciones modulares. Se consideró la posibilidad de crear funciones específicas para el manejo de las operaciones módulo q . Sin embargo, al momento de implementar los algoritmos, se tomó la decisión de asegurar que, en los casos donde se requería una operación módulo q , el resultado o la variable estuvieran dentro del rango utilizando el operador $\%$.

Esta decisión se basó en simplificar la implementación y evitar la complejidad adicional que implicaría la creación y gestión de funciones específicas para operaciones modulares. Incluso algunas fueron creadas y metidas en el código pero se encuentran comentadas.

Cada algoritmo implementado compiló correctamente y se pudo analizar su funcionamiento con varios ejemplos utilizados en un archivo .C tipo test para verificar sus salidas.

Discusión de resultados

Al analizar los resultados obtenidos durante las pruebas de cada algoritmo, surge una inquietud respecto a algunos de ellos. Aunque el estándar proporciona una descripción de la salida esperada, no ofrece ejemplos concretos que aclaren cómo debería ser esa salida. Por ejemplo, en los casos de los algoritmos SampleNTT y SamplePolyCBD, no queda claro si las pruebas realizadas se ejecutaron correctamente. Sin embargo, en los demás algoritmos, podemos afirmar con seguridad que cada uno fue sometido a pruebas exhaustivas y confirmado en su funcionamiento. Esto se debe a que para cada uno de estos algoritmos, se dispuso de su correspondiente algoritmo inverso. Esta metodología permitió modificar la entrada original utilizando dichas funciones y luego aplicarles su inversa, restituyendo así la entrada original. Este proceso de verificación nos brindó la certeza de que estos algoritmos operaban correctamente. Con base en la lógica empleada para desarrollar estos algoritmos y

siguiendo las pautas establecidas en el estándar, se pudo verificar que, a pesar de las incertidumbres encontradas en los algoritmos SampleNTT y SamplePolyCBD, estos fueron implementados en lenguaje C de manera fiel al documento proporcionado.

Las siguientes imagenes muestran la salida por consola de la implementacion de las funciones SampleNTT y SamplePolyCBD de las cuales no se tuvo conocimiento de como verificar si dieron correctamente.

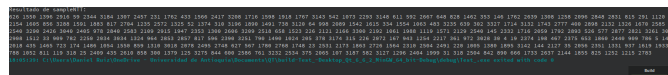


Figura 0-1: Salida por consola SampleNTT

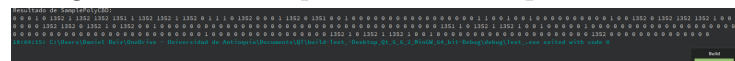


Figura 0-2: Salida por consola SamplePolyCBD

Conclusiones

Teniendo en cuenta el analisis y la implementacion realizada en esta practica, podemos llegar a las siguientes conclusiones:

1. La importancia del analisis previo de cualquier problema, ya que si no entiendes los requerimientos de cualquier proyecto es muy complicado llegar a una solucion eficiente y completa. Lo mas importante es leer y releer [?].
2. Divide y vencerás. En este caso la limitante mas importante fue el tiempo, sin embargo por muy complejo que haya parecido este problema, si empiezas a desglosar cada funcion puedes lograr implementar correctamente cada algoritmo.
3. La importancia del Algebra Lineal a la hora de la abstraccion y comprension de la practica en general. En esta practica aparte de la importancia que tiene la lectura, es indispensable comprender bien los simbolos y la matematica implementada en el estandar ya que se compone en un gran porcentaje por conceptos de Algebra lineal y matematica.
4. Importancia del diagrama de flujo para comprender los problemas. En este caso donde hay tantos algoritmos y funciones es importante saber realizar y tener en cuenta un buen diagrama de flujo que nos indique el

camino que debemos seguir para resolver el problema, esto muchas veces no se tiene en cuenta pero me parece fundamental en nuestro progreso para comprender aun mas la practica y la implementacion realizada.

5. Importancia de las pruebas. Importantisimo realizar pruebas para verificar el correcto funcionamiento de cada algoritmo ya que si se juntan todos en una misma funcion y la pruebas es casi imposible detectar un error puntual lo que implica perdidas de tiempo y de eficiencia en nuestro programa.

6. Tiempo estipulado en la implementacion. En cuanto a los objetivos planteados por la practica y los alcanzados

en esta implementacion, sentimos que nos dio dificultad la implementacion final ya que implica juntar todos los algoritmos realizados por nosotros junto a los realizados por el profe, sin embargo queda la sensacion den que con un poco mas de tiempo se podrian obtener resultados positivos en cuanto a la implementacion completa de la practica.

Bibliografia

[1] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. FIPS 203, August 24, 2023.