UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

Estudio comparativo de diferentes algoritmos de encriptación para comunicaciones industriales

Bogurad Barañski Barañska

Cotutor: Basil Mohammed Al-Hadithi
Departamento: ingeniería eléctrica, electrónica, automática y física aplicada.

Tutor: Roberto Gonzalez Herranz
Departamento: ingeniería eléctrica, electrónica, automática y física aplicada.

Madrid, Septiembre, 2025

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

Título del trabajo

Firma Autor

Firma Tutor

Título: Estudio comparativo de diferentes algoritmos de encriptación para comunicaciones industriales
Autor: Bogurad Barañski Barañska
Tutor: Roberto Gonzalez Herranz

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día ..... de ..... de ... en ....., en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Agradezco a .....

Resumen

Este proyecto se resume en.....

Palabras clave: palabraclave1, palabraclave2, palabraclave3.

Abstract

In this project...

Keywords: keyword1, keyword2, keyword3.

Índice general

Índice de figuras

Índice de tablas

Glosario

Abreviaturas y siglas

DFT	Transformada Discreta de Fourier.
ECC	Criptografía en Curvas Elípticas.
FIPS	Estandar Federal de Procesamiento de la Información.
HQC	Hamming Quasi-Cyclic.
IND-CCA2	Indistinguibilidad bajo ataque de texto cifrado adaptable.
KEM	Mecanismo de intercambio de claves.
LWE	Aprendizaje Con Errores.
LWR	Aprendizaje Con Redondeo.
M-LWE	Aprendizaje Con Errores Modular.
Mod-LWR	Aprendizaje Con Redondeo Modular.
NIST	Instituto Nacional de Seguridad e Información.
NTT	Transformada Teórica de Números.
R-LWE	Aprendizaje Con Errores en Anillos.
RSA	Rivest-Shamir-Adleman.
SHA	Algoritmo Seguro de Hashing.
TFO	Transformadas Fujisaki-Okamoto.

Capítulo 1

Introducción

1.1. Motivación del proyecto

En el ámbito de las comunicaciones industriales, la seguridad en el intercambio de información es un aspecto crítico, especialmente ante el avance de la computación cuántica y su impacto en los algoritmos criptográficos actuales. Este trabajo se enfoca en el análisis, implementación y evaluación de algoritmos de cifrado asimétrico post-cuántico en sistemas embebidos, con el objetivo de garantizar la seguridad de los protocolos de comunicación en un entorno industrial. Se realiza un estudio detallado de los fundamentos matemáticos de los algoritmos asimétricos clásicos y su vulnerabilidad frente al algoritmo de Shor, así como de los nuevos esquemas criptográficos diseñados para resistir ataques cuánticos.

El cifrado asimétrico es esencial para establecer claves seguras en protocolos de intercambio de claves, permitiendo así la utilización de cifrado simétrico en las comunicaciones industriales. Esto es particularmente relevante en sistemas de control en línea, donde el cifrado simétrico debe operar dentro del bucle de control en tiempo real, garantizando tanto seguridad como eficiencia.

Para ello, en este trabajo se comparan diferentes candidatos propuestos en el estándar NIST para evaluar su rendimiento en términos de velocidad, consumo de recursos y nivel de seguridad.

1.2. Objetivos

Para realizar el proyecto, se proponen los siguientes objetivos:

- Analizar el algoritmo de Shor y su impacto en la seguridad del cifrado asimétrico clásico.
- Estudiar los fundamentos matemáticos de los métodos de cifrado asimétrico modernos.
- Implementar un sistema de comunicación entre un PC y un microcontrolador para el intercambio de claves seguras.
- Desarrollar e integrar los siguientes algoritmos de cifrado asimétrico postcuántico en un microcontrolador y PC:
  - Kyber
  - Saber
  - HQC
- Comparar el rendimiento de los algoritmos de cifrado postcuántico evaluando velocidad, consumo de recursos y seguridad.
- Estudiar la necesidad de implementar estos algoritmos en FPGAs en caso de que en el microcontrolador el rendimiento no fuera aceptable.

- Implementar y diseñar un sistema de intercambio de claves que permita la escalabilidad de la solución.

1.3. Herramientas utilizadas

1.3.1. LaTex

Se ha preferido el uso de LaTeX debido a la facilidad que ofrece para el maquetado de textos, superando a otras herramientas de elaboración de documentos. Además, LaTeX permite crear figuras vectorizadas, representar correctamente ecuaciones y ubicar adecuadamente figuras, tablas y bibliografía.

1.3.2. TikzMaker

Esta herramienta [5] permite crear figuras vectorizadas de LaTeX mediante el paquete de circuitikz. Su principal ventaja radica en la interfaz gráfica que proporciona y en la facilidad para elaborar figuras.

1.3.3. C y C++

1.3.4. Microprocesador CY8CPROTO-063-BLE

se usa el [6]

1.4. Estructura del documento

A continuación y para facilitar la lectura del documento, se detalla el contenido de cada capítulo:

- En el capítulo 1 se realiza una introducción.
- En el capítulo 2 se estudia trabajos realizados con relación al tema.
- En el capítulo 3 se desarrollan los fundamentos matemáticos del proyecto.
- En el capítulo 4 se describe la implementación de los algoritmos.
- En el capítulo 5 se presentan y analizan los resultados obtenidos en el capítulo anterior como consecuencia de ejecutar el software realizado.
- En el capítulo 6 se realiza una conclusión.

Capítulo 2

Estado del arte

Los artículos que de momento tengo que son relevantes mencionar, lo haré después de los fundamentos: [7] [8] [9] [10] [11] [12] [13] [**?**] [14] [15] [16] [17] [18]

2.1. Introducción

a

2.1.1. Tipos de cifrado

a

2.1.2. Necesidad del cifrado asimétrico

a

2.2. Sistemas de intercambio de claves

a

2.3. Cambio en el paradigma de cifrado asimétrico. Algoritmo de Shore

a

2.3.1. Algoritmo de Shore

Generico: [19] ECC: [20] RSA: [21]

2.4. Evolución de los algoritmos postcuánticos

a fundamentos y distintas alternativas a codebased [22]

2.5. Cifrado asimétrico postcuántico en sistemas embebidos

a

Capítulo 3

Fundamentos generales

En este capítulo se desarrollan las bases matemáticas de los distintos algoritmos a implementar.

3.1. Introducción

Los algoritmos de cifrado asimétrico analizados en este Trabajo Fin de Grado corresponden a mecanismos de intercambio de claves (KEM), es decir, procedimientos diseñados para establecer un secreto compartido entre dos partes. Este secreto se utiliza posteriormente como clave en algoritmos de cifrado simétrico, con el fin de garantizar una comunicación segura, ya que este tipo de cifrado requiere que ambas partes dispongan previamente de una misma clave secreta.

Siguiendo las recomendaciones del NIST [23], uno de los enfoques consiste en establecer dichas claves a través de un canal público de comunicaciones. Para ello, de forma general y en el contexto de los algoritmos considerados en este trabajo, se emplea un esquema como el ilustrado en la figura 3.1 donde Alice y Bob intercambian las claves mediante tres pasos:

1. **Generación de llaves**: se generan una llave privada, que Alice debe mantener en secreto para no comprometer la seguridad del protocolo, y una llave pública, derivada de la privada, que Alice enviará a Bob.

2. **Encapsulado**: Bob, usando valores la llave pública de Alice, genera un texto cifrado que enviará a Alice y el secreto compartido.

3. **Decapsulado**: Alice, a partir del texto cifrado recibido y su llave privada, deriva el mismo secreto compartido que Bob.

Por último, es importante señalar que existe un paso adicional denominado confirmación de claves, con el objetivo de verificar que ambas partes han obtenido el mismo secreto compartido. El NIST propone realizarlo mediante la generación y verificación de un código de autenticación de mensaje utilizando una clave derivada del secreto compartido. Sin embargo, los mecanismos específicos para llevar a cabo esta verificación quedan fuera del alcance del presente trabajo.

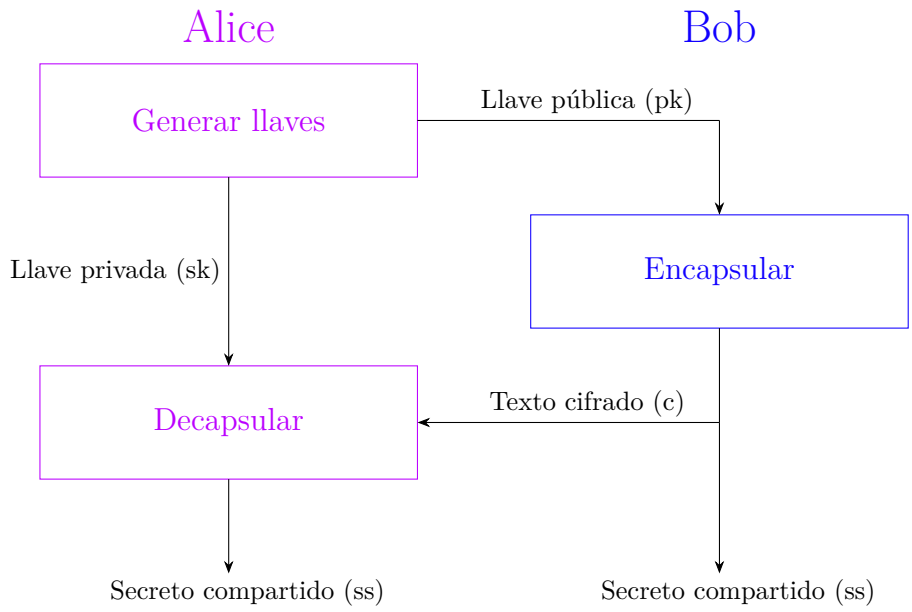

Figura 3.1: Representación del protocolo empleado en los algoritmos asimétricos para establecer una clave común o secreto compartido.

3.2. Algoritmos de Hashing

Las funciones de hashing son funciones utilizadas para obtener una salida determinista a partir de un mensaje de entrada. Se caracterizan porque, dada la salida, no resulta factible reconstruir el mensaje original de manera eficiente.

La familia de funciones de hashing empleada en este trabajo corresponde a SHA3 (Secure Hash Algorithm), definida en el estándar federal FIPS-202 [4]. Dichas funciones se fundamentan en el algoritmo `Keccak-p` [24], del cual se derivan cuatro funciones de hash y dos funciones de salida extendida.

- **Funciones de hash**: transforman un mensaje de entrada en una salida de longitud fija de 224, 256, 384 o 512 bits, según la variante seleccionada. El nombre del algoritmo refleja dicha longitud. $SHA3-256$ genera salidas de 256 bits.
- **Funciones de salida extendida**: generan salidas de longitud arbitraria a partir de un mensaje de entrada. En función de la seguridad deseada se utilizan los algoritmos $SHAKE-128$ o $SHAKE-256$.

3.2.1. Algoritmo `Keccak-p`

Cada permuta dentro del algoritmo `Keccak-p`[b,n] se especifica mediante dos parámetros:

1. **El ancho de la permuta** $b \in \{25, 50, 100, 200, 400, 800, 1600\}$: tamaño de las cadenas a permutar.
2. **El número de rondas** $n_r \in \mathbb{Z}$: el número de iteraciones de una transformación interna.

Además del valor de b se definen dos magnitudes auxiliares $w = b/25$ y $l = \log_2(b/25)$.

3.2.1.1. Notación básica

En esta sección se adopta la siguiente notación

- El símbolo || denota la concatenación de dos cadenas.
- La expresión 0^j representa una cadena de j bits de ceros.
- La función `len`(P) indica la longitud de la cadena P.
- La notación $Z(j)$ corresponde al j-ésimo bit de la cadena Z. Por ejemplo, si $Z = 10010$, entonces $Z(4) = 1$.
- La función `trunc`$_d(Z)$ devuelve los d primeros bits de Z.

3.2.1.2. Vectores de estado

En `Keccak-p` se trabaja mediante vectores de estado de tamaño b, los cuales convierten las cadenas S en una matriz $A(x, y, z)$, donde $x, y \in \{0, 1, 2, 3, 4\}$ y z toma valores en un rango de tamaño w. Esta representación al igual que la nomenclatura se representan en las figuras 3.2 y 3.3.

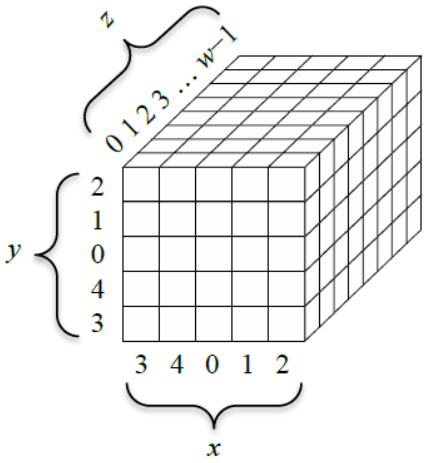

Figura 3.2: Dibujo de un vector de estado [4].

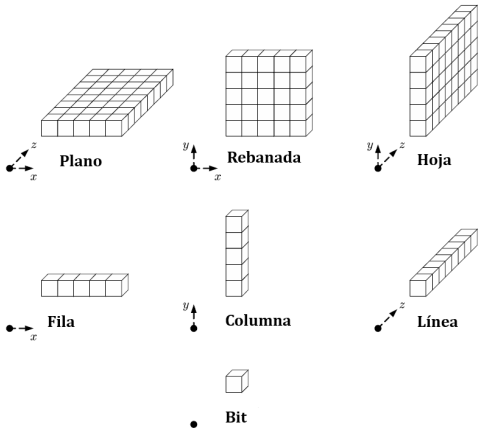

Figura 3.3: Elementos de un vector de estado [4].

Para convertir de la cadena S en la matriz A se usa la siguiente expresión:

$$A(x, y, z) = S(w \cdot (5y + x) + z) \tag{3.1}$$

Para convertir de A a S, se concatenan primero los bits dentro de cada línea, luego las líneas dentro de cada plano y, finalmente, los planos hasta formar la cadena S.

$$\begin{array}{lllll} \text{Linea}(i,j) = & = A(i,j,0) & ||A(i,j,1) & ||\dots & ||A(i,j,w-1) \\ \text{Plano}(j) & = \text{Linea}(0,j) & ||\text{Linea}(1,j) & ||\dots & ||\text{Linea}(4,j) \\ S & = \text{Plano}(0) & ||\text{Plano}(1) & ||\dots & ||\text{Plano}(4) \end{array} \tag{3.2}$$

Una vez definidos los vectores de estado, estos se manipulan mediante cinco transformadas, denotadas θ, ρ, π, χ y ι. Todas las transformadas toman como entrada un vector de estado $A(x, y, z)$ y lo transforman, devolviendo un nuevo vector de estado como salida $A'(x, y, z)$. La transformada ι recibe además como parámetro el índice de ronda i_r.

1. **Transformada** θ: esta transformada realiza una XOR, $\oplus$, del bit $A(x, y, z)$ con la paridad de las columnas C$(x-1 \bmod 5, z)$ y C$(x+1 \bmod 5, z-1 \bmod \text{w})$. Para ello, sigue los siguientes pasos:

Algoritmo 1 Transformada θ en Keccak-p

Entrada: A

Salida: A'

1: Calcular la paridad de cada columna, C:

$$C(x, z) := A(x, 0, z) \oplus A(x, 1, z) \oplus A(x, 2, z) \oplus A(x, 3, z) \oplus A(x, 4, z) \tag{3.3}$$

2: Combinar la paridad de ambas columnas, D:

$$D(x, z) := C(x - 1 \text{ mód } 5, z) \oplus C(x + 1 \text{ mód } 5, z - 1 \text{ mód } w) \tag{3.4}$$

3: Realizar la XOR con el estado:

$$A'(x, y, z) := A(x, y, z) \oplus D(x, z) \tag{3.5}$$

4: **return** A'

2. **Transformada** ρ: esta transformada rota los bits de cada línea un offset módulo la longitud de la línea. Los offsets antes de efectuar el operador módulo se listan en la tabla 3.1.

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Tabla 3.1: Offsets de la transformada ρ.

Para realizar estas rotaciones sigue los siguientes pasos:

Algoritmo 2 Transformada ρ en Keccak-p

Entrada: A

Salida: A'

1: Asignar el caso especial de $(x, y, z) := (0, 0, z)$:

$$A'(0, 0, z) := A(0, 0, z) \tag{3.6}$$

2: **for** $t = 0 : 23$ **do** ▷ Para los 23 valores restantes

3: Asignar el valor de la tabla 3.1 modulo w a cada punto:

$$A'(x, y, z) := A(x, y, [z - (t + 1)(t + 2)/2 \bmod \text{w}]) \tag{3.7}$$

4: Asignar $(x, y) := (y, 2x + 3y \bmod 5)$

5: **end for**

6: **return** A'

3. **Transformada** π: esta transformada rota las coordenadas de cada rebanada (x, y) tal como se ilustra en la tabla 3.2.

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	(4,3)	(0,4)	(1,0)	(2,1)	(3,2)
$y=1$	(1,3)	(2,4)	(3,0)	(4,1)	(0,2)
$y=0$	(3,3)	(4,4)	(0,0)	(1,1)	(2,2)
$y=4$	(0,3)	(1,4)	(2,0)	(3,1)	(4,2)
$y=3$	(2,3)	(3,4)	(4,0)	(0,1)	(1,2)

Tabla 3.2: Tabla de transformación π. Para obtener el valor de $A'(x, y)$, se debe leer el valor de la posición (x', y') indicada en la celda correspondiente de la matriz original A.

Para realizar esta rotación se sigue el siguiente algoritmo:

Algoritmo 3 Transformada π en Keccak-p

Entrada: A

Salida: A'

1: Calcular la rotación:

$$A'(x, y, z) := A(x + 3y \bmod 5, x, z) \tag{3.8}$$

2: **return** A'

4. **Transformada** χ: esta transformada actualiza cada bit como el XOR entre el bit original y una combinación no lineal de sus vecinos en la misma fila mediante una puerta AND como se puede ver en la figura 3.4.

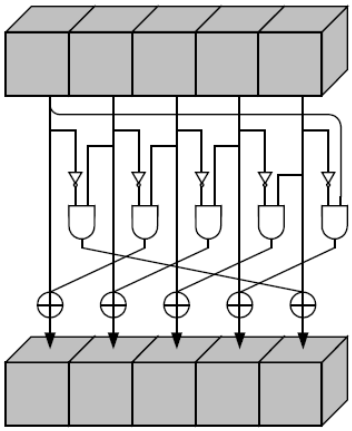

Figura 3.4: Representación de la transformada χ realizada en cada fila [4]. Arriba la matriz $A(x, y, z)$ y abajo la matriz $A'(x, y, z)$.

Para realizar esta rotación se sigue el siguiente algoritmo:

Algoritmo 4 Transformada χ en Keccak-p

Entrada: A

Salida: A'

1: Calcular la rotación:

$$A'(x,y,z) := A(x,y,z) \oplus \{[A(x+1 \bmod 5, y, z) \oplus 1] \& A(x+2 \bmod 5, y, z)\} \tag{3.9}$$

2: **return** A'

5. **Transformada** ι: esta transformada modifica solo algunos bits de la línea $(0,0)$ de tal manera que que dependa del índice de ronda i_r. Para ello, se sigue el siguiente algoritmo:

Algoritmo 5 Transformada ι en Keccak-p

Entrada: A, i_r

Salida: A'

1: Asignar:

$$A'(x,y,z) := A(x,y,z) \tag{3.10}$$

2: Inicializar el vector de ceros RC de longitud w:

3: **for** j=0:l **do** ▷ $l = \log_2(b/25)$

4: Asignar

$$RC(2^j - 1) := \texttt{rc}(j + 7i_r) \tag{3.11}$$

5: **end for**

6: Asignar:

$$A'(0,0,z) := A'(0,0,z) \oplus RC(z) \tag{3.12}$$

7: **return** A'

Este algoritmo depende de la función $\texttt{rc}(t)$la cual, dado un entero, t, genera un bit mediante un procedimiento basado en un registro de desplazamiento con retroalimentación lineal como se describe en el siguiente algoritmo:

Algoritmo 6 rc

Entrada: t

Salida: $rc(t)$

1: **if** $t \bmod 255 = 0$ **then**
2: **return** 1
3: **end if**
4: **for** i=1:t mod 255 **do**
5: $R := R||R$
6: $R[0] := R[0] \oplus R[8]$
7: $R[4] := R[4] \oplus R[8]$
8: $R[5] := R[5] \oplus R[8]$
9: $R[6] := R[6] \oplus R[8]$
10: $R := \texttt{Trunc}_8[R]$
11: **end for**
12: **return** $R[0]$

Una vez descritas estas transformadas la función de cada ronda, $\texttt{Rnd}(A, i_r)$ se define como:

$$\texttt{Rnd}(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r) \tag{3.13}$$

Con esta función ya se puede describir el algoritmo $\texttt{Keccak-p}(b, n_r)$ que consiste en ejecutar n_r iteraciones de `Rnd` para una cadena de longitud b:

Algoritmo 7 Keccak-p

Entrada: S, n_r

Salida: S'

1: Convertir S en un vector de estado A
2: **for** i_r=12+2l-n_r:12+2l-1 **do**
3: A=$\texttt{Rnd}(A, i_r)$
4: **end for**
5: Convertir A en una cadena S' de longitud b
6: **return** S'

3.2.1.3. Función esponja

Este tipo de función, $\texttt{sponge}[f, pad, r](N, d)$, permite convertir una entrada binaria, N, de cualquier longitud a una salida, Z, de longitud d. Para ello, usa tres componentes:

1. Una función sobre cadenas de longitud fija, f.
2. Una regla de relleno, *pad*.
3. Un parámetro llamado velocidad, r, menor que la longitud, b, de las cadenas que procesa f.

La función `sponge`, ilustrada en la figura 3.5, aplica la regla de relleno sobre la entrada y procesa la información mediante los tres componentes descritos. En la fase de absorción, el mensaje se divide en bloques de longitud r, que se incorporan iterativamente al estado interno de longitud b utilizando la permutación f, que en este caso es `keccak-p`. En esta construcción se impone que el parámetro de capacidad sea

$$c = b - r \tag{3.14}$$

Tras finalizar la fase de absorción, se inicia la fase de extracción, en la que se producen bloques de salida de tamaño r de manera iterativa hasta obtener los d bits requeridos.

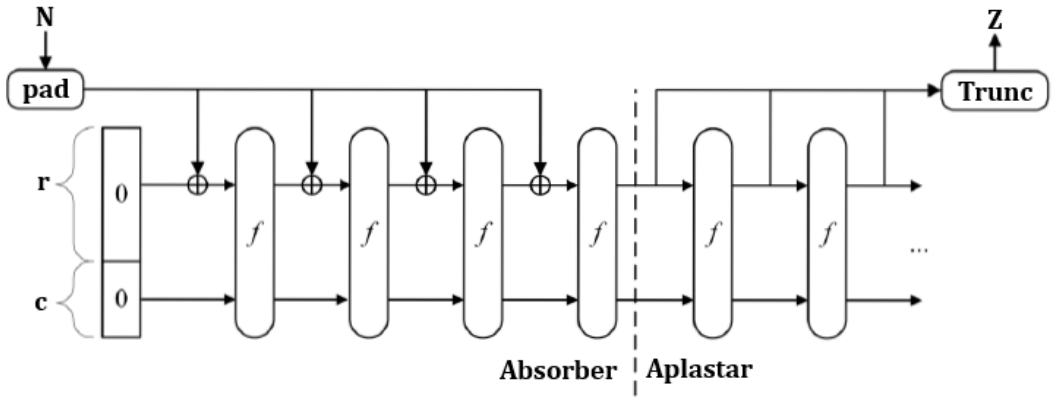

Figura 3.5: Representación visual del funcionamiento del algoritmo `sponge` [4].

Teniendo en cuenta la descripción anterior la función `sponge` se define como:

Algoritmo 8 sponge

Entrada: N, d

Salida: Z

1: Calcular:
$$P = N||\texttt{pad}(r, \texttt{len}(N)) \tag{3.15}$$
2: Calcular:
$$n = \frac{len(P)}{r} \tag{3.16}$$
3: Asignar $c = b - r$ y $S = 0^b$
4: Sea $P_0, \ldots, P_{n-1}$ la secuencia de cadenas de longitud r tal que $P = P_0|| \ldots ||P_{n-1}$.
5: **for** i=0:n-1 **do**
6: Asignar:
$$S = f(S \oplus (P_i||0^c)) \tag{3.17}$$
7: **end for**
8: Inicializar Z como una cadena vacía
9: Asignar
$$Z = Z||\texttt{Trunc}_r(S) \tag{3.18}$$
10: **if** $d \leq len(Z)$ **then**
11: **return** $\texttt{Trunc}_d(S)$
12: **end if**
13: Asignar:
$$S = f(S) \tag{3.19}$$
14: Ir al paso 9

El algoritmo de relleno utilizado en la función `sponge` es el algoritmo `pad10*1` que dado un entero positivo, x y un entero no negativo m se obtiene una cadena, P, de tal manera que:

$$m + \texttt{len}(P) = \lambda x \tag{3.20}$$

El algoritmo de `pad10*1`:

Algoritmo 9 pad10*1

Entrada: x, m

Salida: P

1: Asignar:

$$j = -m - 2 \bmod x \tag{3.21}$$

2: Calcular

$$P = 1||0^j||1 \tag{3.22}$$

3: **return** P

Definida la función de esponja, para el caso de $b = 1600$ el algoritmo de la familia Keccak se denomina `Keccak`$[c](N, d)$. Este algoritmo tiene la siguiente definición:

$$\texttt{Keccak}[c](N, d) = \texttt{sponge}[\texttt{Keccak-p}(1600, 24), \texttt{pad10*1}, 1600 - c](N, d) \tag{3.23}$$

Por tanto, las funciones de hash se definen como:

1. `SHA3-224`(M)=`Keccak`$[448](M||01, 224)$
2. `SHA3-256`(M)=`Keccak`$[512](M||01, 256)$
3. `SHA3-384`(M)=`Keccak`$[768](M||01, 384)$
4. `SHA3-512`(M)=`Keccak`$[1024](M||01, 512)$
5. `SHAKE128`(M, d)=`Keccak`$[256](M||1111, d)$
6. `SHAKE256`(M, d)=`Keccak`$[512](M||1111, d)$

3.2.1.4. Seguridad

Mediante la tabla 4 del artículo [4] se muestra la seguridad en bits que tienen cada uno de los algoritmos SHA3. Estas tres condiciones de seguridad para un algoritmo de hash son las siguientes:

1. **Resistencia a la colisión**: dificultad para encontrar dos entradas diferentes que producen el mismo hash. Esta propiedad previene ataques donde dos mensajes diferentes se pueden sustituir sin que se pueda detectar.
2. **Resistencia a preimagen**: dificultad para que dado un hash, h, se pueda encontrar una entrada, x, tal que `hash`$(x) = h$. Esta propiedad permite proteger datos sensibles de tal manera que si el hash se revela no comprometa las contraseñas.
3. **Resistencia a preimagen secundaria**: dificultad para que dada una entrada, x_1, se pueda encontrar una entrada diferente, x_2, que cumpla `hash`$(x_1) =$ `hash`(x_2). Esta propiedad previene la falsificación de mensajes.

3.3. Funcionamiento básico de los algoritmos postcuánticos

En esta sección se describe el funcionamiento de los algoritmos postcuánticos analizados en este trabajo. Dado que no se desarrollaron implementaciones propias, sino que se utilizó el código proporcionado por el NIST en la tercera [25] y cuarta [26] ronda del proceso de estandarización, resulta apropiado presentar su funcionamiento aquí en lugar de en la sección de desarrollo.

3.3.1. Fundamentos matemáticos de CRYSTALS-Kyber

Se utilizará la misma notación empleada en el artículo [1]. El conjunto de los enteros sin signo de 8 bits se denota por $\mathcal{B} = \{0, \ldots, 255\}$. Para representar vectores de tamaño k, se utiliza la notación $\mathcal{B}^k$, mientras que para vectores de tamaño arbitrario se emplea $\mathcal{B}^*$.

Para trabajar con estos vectores, se utiliza el símbolo $||$ para denotar la concatenación de dos cadenas, y la notación $+k$ para indicar el desplazamiento de k bytes desde el inicio de una cadena. Por ejemplo, si se tiene una cadena a de longitud l y se concatena con una cadena b, se obtiene:

$$c = a||b \tag{3.24}$$

Entonces:

$$b = c + l \tag{3.25}$$

Para denotar vectores, se utiliza la notación $v[i]$, donde v es un vector columna e i indica la posición del elemento (empezando desde 0, si no se indica lo contrario). Para las matrices, se emplea la notación $A[i][j]$, donde i representa la fila y j la columna. La transpuesta de una matriz A se denota como A^T.

Se denota mediante $\lfloor x \rceil$ el redondeo de x al entero más cercano. Por ejemplo: $\lfloor 2{,}3 \rceil = 2$, $\lfloor 2{,}5 \rceil = 3$ y $\lfloor 2{,}8 \rceil = 3$.

Se denota mediante $\|x\|$ al valor absoluto de x.

Para las reducciones modulares se emplean dos tipos: una centrada en cero y otra correspondiente a la reducción modular estándar. Para la reducción modular centrada en cero, sea α un entero par. Esta operación se define como:

$$r' = r \bmod^{\pm} \alpha \quad \Longrightarrow \quad -\frac{\alpha}{2} < r' \leq \frac{\alpha}{2} \tag{3.26}$$

Mientras que la reducción modular estándar se denota como:

$$r' = r \bmod^{+} \alpha \quad \Longrightarrow \quad 0 \leq r' < \alpha \tag{3.27}$$

Finalmente, se denota mediante $s \leftarrow S$ la selección de s de manera uniformemente aleatoria del conjunto S. Si S representa una distribución de probabilidad, entonces s se selecciona de acuerdo con dicha distribución.

3.3.1.1. Transformada Teórica de Números o Number Theoretic Transform (NTT)

Para acelerar las operaciones de multiplicación en el esquema basado en retículas, se utiliza la Transformada Teórica de Números (NTT, por sus siglas en inglés), la cual permite reducir la complejidad temporal de la multiplicación de polinomios desde $\mathcal{O}(n^2)$, correspondiente al método tradicional, hasta $\mathcal{O}(n\log(n))$. Para más detalles, consúltese [27].

Antes de pasar a explicar el funcionamiento de este método es relevante aclarar que se trabaja sobre el siguiente anillo de polinomios para realizar las operaciones, denotado mediante R_q:

$$R_q \coloneqq \frac{\mathbb{Z}_q[X]}{X^n+1} \tag{3.28}$$

En la implementación especificada de Kyber, según el artículo [1], se utiliza un valor de $q = 3329$ y $n = 256$. Esta elección es esencial para permitir el uso de la multiplicación mediante la Transformada Teórica de Números (NTT), la cual requiere que $n|(q-1)$, es decir, que n divida a $(q-1)$. Esta condición garantiza la existencia de n raíces enésimas de la unidad en Z_q, lo cual es necesario para definir la NTT. La validez de esta afirmación se fundamenta en el siguiente teorema[0]:

Teorema 1 *Para n, $q > 1$, el cuerpo Z_q tiene una raíz enésima de la unidad si y solo sí $n|(q-1)$*

Demostración 1 *Si ω es una raíz enésima de la unidad en el conjunto $\mathbb{Z}_q$, entonces el conjunto:*

$$\Omega = \left\{1, \omega, \omega^2, \ldots, \omega^{n-1}\right\} \tag{3.29}$$

forma un subgrupo cíclico H del grupo multiplicativo G_{q-1}. Por el Teorema de Lagrange, se concluye que el orden de H divide al orden de G_{q-1}, es decir, $n \mid (q-1)$.

Dado que G_{q-1} es también un grupo cíclico, existe un generador α tal que, por el pequeño teorema de Fermat, se cumple:

$$\alpha^{q-1} = 1 \tag{3.30}$$

Por lo tanto, el grupo G_{q-1} puede escribirse como:

$$G_{q-1} = \left\{1, \alpha, \alpha^2, \ldots, \alpha^{q-2}\right\} \tag{3.31}$$

Si se define ω como:

$$\omega = \alpha^{\frac{q-1}{n}}, \tag{3.32}$$

entonces:

$$\omega^n = \alpha^{q-1} = 1, \tag{3.33}$$

y además, para todo $0 < k < n$, se cumple:

$$k \cdot \frac{q-1}{n} < q-1 \quad \Rightarrow \quad \omega^k \neq 1. \tag{3.34}$$

Por lo tanto, ω es una raíz enésima de la unidad en $\mathbb{Z}_q$.

[0]Extraído de la siguiente página web: `https://www.csd.uwo.ca/~mmorenom/CS874/Lectures/Newton2Hensel.html/node9.html`

Por tanto, el polinomio $X^{256}+1$ se puede factorizar sobre el cuerpo $\mathbb{Z}_q$. En la implementación concreta del esquema Kyber, este polinomio se descompone en 128 factores cuadráticos.

A este polinomio se le aplica la transformada NTT a sus coeficientes, la cual no es más que una variación de la DFT aplicada a cuerpos finitos $\mathbb{Z}_q$ y aplicada a polinomios de grado n. Para ello, se definen dos operaciones fundamentales:

1. La transformada directa:
$$\hat{a}_j = \sum_{i=0}^{n-1} \phi^{i(2j+1)} a_j \bmod q \tag{3.35}$$

2. La transformada inversa:
$$a_i = n^{-1} \sum_{j=0}^{n-1} \phi^{-i(2j+1)} \hat{a}_j \bmod q \tag{3.36}$$

Donde ϕ es un valor tal que $\phi^2 = \omega$, con ω una raíz enésima de la unidad en $\mathbb{Z}_q$ y n^{-1} es la inversa multiplicativa de n en $\mathbb{Z}_q$.

A continuación, se presenta un ejemplo extraído de [27] para ilustrar su funcionamiento. Sea el polinomio $G(x) = 5 + 6x + 7x^2 + 8x^3$, cuyo vector de coeficientes es $g = [5, 6, 7, 8]$. Trabajando en el anillo $\mathbb{Z}_{7681}$, y tomando $\phi = 1925$, se puede calcular la transformada NTT $\hat{g}$. Aplicando luego la transformada inversa, es posible recuperar el vector original g.

$$\hat{g} = \begin{bmatrix} \phi^0 & \phi^1 & \phi^2 & \phi^3 \\ \phi^0 & \phi^3 & \phi^6 & \phi^1 \\ \phi^0 & \phi^5 & \phi^2 & \phi^7 \\ \phi^0 & \phi^7 & \phi^6 & \phi^5 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 & 1925 & 3383 & 6468 \\ 1 & 6468 & 4298 & 1925 \\ 1 & 5756 & 3383 & 1213 \\ 1 & 1213 & 4298 & 5756 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix} \tag{3.37}$$

Aplicando la transformada inversa, donde la inversa de $\phi = 1925$ en $\mathbb{Z}_{7681}$ es $\phi^{-1} = 1213$ y el inverso del orden del polinomio $n = 4$ es $n^{-1} = 5761$:

$$g = n^{-1} \begin{bmatrix} \phi^0 & \phi^0 & \phi^0 & \phi^0 \\ \phi^{-1} & \phi^{-3} & \phi^{-5} & \phi^{-7} \\ \phi^{-2} & \phi^{-6} & \phi^{-2} & \phi^{-6} \\ \phi^{-3} & \phi^{-1} & \phi^{-7} & \phi^{-5} \end{bmatrix} \cdot \hat{g} = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1213 & 5756 & 6468 & 1925 \\ 4298 & 3383 & 4298 & 3383 \\ 5756 & 1213 & 1925 & 6468 \end{bmatrix} \cdot \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix} \tag{3.38}$$

Con estas transformadas definidas se define el producto o convolución negativa en el anillo $R_q \coloneqq \dfrac{\mathbb{Z}_q[X]}{X^n+1}$ entre dos polinomios g y h como:

$$g \cdot h = \mathrm{NTT}^{-1}(\mathrm{NTT}(g) \circ \mathrm{NTT}(h)) \tag{3.39}$$

Donde la operación $\circ$ denota la multiplicación elemento a elemento (o punto a punto) entre los vectores en $Z_q[X]$.

Ahora, queda demostrar que el tiempo de ejecución de la NTT es de $\mathcal{O}(n\log(n))$. Para lograr este cometido, se aprovechan dos propiedades que cumplen estas raíces calculadas:

1. Peridicidad:
$$\phi^{k+2n} = \phi^k \tag{3.40}$$
2. Simetría:
$$\phi^{k+n} = \phi^{-k} \tag{3.41}$$

Con estas propiedades, se puede implementar el algoritmo de Cooley-Tukey [28] que consiste en ir descomponiendo el problema en mitades de manera recursiva para reducir al máximo la cantidad de cálculos realizados. Partiendo de la transformada directa y desarrollando:

$$\hat{a}_j = \sum_{i=0}^{n-1} \phi^{i(2j+1)} a_j \bmod q = \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i} + \phi^{2j+1} \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i+1} \bmod q \tag{3.42}$$

Si se sustituye $A_j = \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i}$ y $B_j = \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i+1}$. Ahora aplicando simetría en ϕ:

$$\hat{a}_j = A_j + \phi^{4ij+2i} B_j \bmod q \tag{3.43}$$

$$\hat{a}_{j+n/2} = A_j - \phi^{4ij+2i} B_j \bmod q \tag{3.44}$$

Donde las matrices A_j y $B_j$pueden obtenerse como el resultado de aplicar la NTT sobre la mitad de los puntos, gracias a la estructura recursiva del algoritmo. Esto implica que, si n es una potencia de 2, el proceso puede repetirse recursivamente sobre subproblemas de tamaño cada vez menor, hasta alcanzar el caso base.

De manera similar, se puede mostrar esta propiedad para la transformada inversa. Por tanto, se demuestra que este algoritmo tiene complejidad $\mathcal{O}(n\log(n))$.

En el caso concreto de Kyber [1], la Transformada (NTT) de un polinomio en el anillo R_q se representa como un vector de 128 polinomios de grado 1.

Sean las 256 raíces enésimas de la unidad $\left\{\xi, \xi^3, \ldots, \xi^{255}\right\}$, con $\xi = 17$ como primera raíz primitiva. Entonces, se cumple que:

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \xi^{2i+1}\right) \tag{3.45}$$

Aplicando la NTT propuesta en [1] a un polinomio $f \in R_q$ se obtiene:

$$NTT(f) = \hat{f} = \left(\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, ..., \hat{f}_{254} + \hat{f}_{255} X\right) \tag{3.46}$$

Con

$$\begin{aligned} \hat{f}_{2i} &= \sum_{j=0}^{127} f_{2j} \xi^{(2i+1)j} \\ \hat{f}_{2i+1} &= \sum_{j=0}^{127} f_{2j+1} \xi^{(2i+1)j} \end{aligned} \tag{3.47}$$

Donde mediante la transformada directa NTT e inversa NTT^{-1} se puede realizar el producto de f,$g \in R_q$ de manera eficiente de la siguiente manera:

$$h = f \cdot g = \mathrm{NTT}^{-1}\left[\mathrm{NTT}(f) \circ \mathrm{NTT}(g)\right] \tag{3.48}$$

Siendo $\hat{h} = \hat{f} \circ \hat{g} = \mathrm{NTT}(f) \circ \mathrm{NTT}(g)$ la multiplicación base definida como:

$$\hat{h}_{2i} + \hat{h}_{2i+1} X = \left(\hat{f}_{2i} + \hat{f}_{2i+1}\right)\left(\hat{g}_{2i} + \hat{g}_{2i+1}\right) \bmod \left(X^2 - \xi^{2i+1}\right) \tag{3.49}$$

En la figura 3.6 se puede ver una representación gráfica de como funciona este proceso.

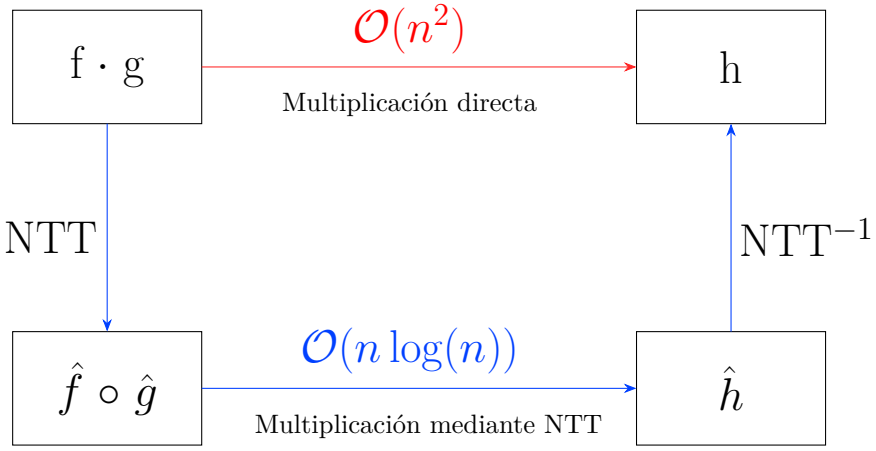

Figura 3.6: Representación del cálculo de un producto de polinomios mediante NTT en comparación con los algoritmos clásicos.

3.3.1.2. Aprendizaje Con Errores o Learning With Errors (LWE)

Para la elaboración de esta sección, se ha utilizado la información contenida en el artículo [29], el cual expone los fundamentos matemáticos del problema de Aprendizaje con Errores (Learning With Errors, LWE). Este problema constituye la base teórica sobre la que se sustenta el esquema criptográfico Kyber.

Para ello, los esquemas basados en LWE trabajan con un objeto matemático conocido como retícula. Una retícula es un conjunto discreto de puntos en el espacio, generado por todas las combinaciones lineales con coeficientes enteros de un conjunto de n vectores linealmente independientes que conforman una base $\mathbb{B} = \{b_1, ..., b_n\}$:

$$\mathcal{A} = \mathcal{L}(\mathbb{B}) = \left\{ \sum_{i \in n} z_i b_i : z \in Z^n \right\} \tag{3.50}$$

Esta definición será útil en la demostración de la seguridad de los esquemas basados en LWE. Aun así, antes de pasar a la explicación del algoritmo de intercambio de claves públicas, es necesario definir el problema de Aprendizaje con Errores.

El Aprendizaje con Errores consiste en la tarea de distinguir entre parejas de la forma $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, donde $b_i \approx \langle a_i, s \rangle$, y parejas generados de manera completamente aleatoria. Donde, $s \in \mathbb{Z}_q^n$ es el secreto que se mantiene fijo para todas las muestras, $a_i \in \mathbb{Z}_q^n$ es un vector elegido uniformemente al azar, y $\langle , \rangle$ denota el producto escalar Euclídeo.

La principal utilidad criptográfica del problema LWE radica en que, para quien no conoce el secreto, resulta computacionalmente difícil distinguir entre pares estructurados y pares aleatorios. En cambio, quien sí conoce el secreto puede identificar fácilmente qué parejas son consistentes con este, lo que permite construir esquemas seguros de cifrado e intercambio de claves.

LWE en anillos

En Kyber no se parte del problema LWE "puro", ya que los esquemas basados directamente en LWE suelen presentar claves de tamaño y tiempos de cálculo con un orden de magnitud $\mathcal{O}(n^2)$ [30]. Esto se debe a que, al trabajar con matrices genéricas, se carece de la estructura de anillo necesaria para aprovechar multiplicaciones mediante convolución rápida, es decir, la NTT. En cambio, Kyber se fundamenta en una variante conocida como Modulus-LWE (M-LWE), donde las operaciones de

multiplicación matricial se pueden implementar como multiplicaciones de polinomios en $\frac{\mathbb{Z}_q[x]}{X^n+1}$, lo cual permite usar la NTT.

Antes de describir en detalle el funcionamiento de la M-LWE, conviene introducir primero el problema Ring-LWE (R-LWE). La M-LWE puede entenderse como una extensión modular (un vector) de R-LWE. Este enfoque permite romper la simetría inherente a un anillo y aporta mayor flexibilidad en la selección de parámetros para lograr una mejor relación entre eficiencia y resistencia criptográfica [31].

En cuanto a la eficiencia en la reducción del tamaño de las claves, esta puede observarse claramente a través del siguiente ejemplo ilustrativo:

Esquema	LWE	R-LWE
Tamaño clave pública	$\mathcal{O}(n^2)$ bits	$\mathcal{O}(n)$
Operación para generar la pareja	$b_i = \langle a_i, s\rangle + e_i$	$b[x] = a[x] \cdot s[x] + e[x]$
Matriz A	$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	3+11X
Secreto s	$\begin{pmatrix} 5 \\ 6 \end{pmatrix}$	5+6X
Error e	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$	1+X
Resultado b	$\begin{pmatrix} 1 \\ 6 \end{pmatrix}$	1+6X

Tabla 3.3: Tabla con un ejemplo numérico del calculo del valor b necesario para el problema LWE como en R-LWE para ver como efectivamente en la clave pública (a, b) el tamaño es menor. En este ejemplo se trabaja en $\mathbb{Z}_{17}$ y $X^2 + 1$. En M-LWE las claves son de tamaño similar que en R-LWE pues la matriz A se puede reconstruir a través de una semilla y la matriz b solo ve reescalada en función de la seguridad empleada.

Aplicación del R-LWE para el intercambio de claves públicas

A continuación, se presenta el algoritmo principal empleado para el intercambio de claves basado en el problema R-LWE que fundamenta matemáticamente el funcionamiento de Kyber. El uso de la M-LWE es similar y se puede consultar en la sección 3.3.1.4.

El algoritmo de generación de claves emplea los valores q y n , que definen la estructura algebraica, para calcular la llave privada sk y la llave pública pk.

Algoritmo 10 Generación claves R-LWE en $\mathbb{Z}_q[x]/(X^n+1)$ [0]

Entrada: q, n

Salida: sk, pk

1: Generar $a \in R_q$ al azar según una distribución uniforme.
2: Generar $sk, e \in R_q$ como elementos pequeños según la distribución del error.
▷ Binomial en Kyber, discreta Gaussiana en otros esquemas
3: Calcular

$$b := a \cdot sk + e \tag{3.51}$$

4: **return** $(sk, pk := a||b)$

En el algoritmo de cifrado a partir de la llave pública, pk y un mensaje $z \in \{0,1\}^n$ se obtienen los textos cifrados u y v que serán enviados al poseedor de la clave privada para descifrar el mensaje.

Algoritmo 11 Cifrado R-LWE [0]

Entrada: pk, z, q, n

Salida: u, v

1: Generar $r, e_1, e_2 \in R_q$ como elementos pequeños según la distribución del error.
2: Calcular el primer texto cifrado u:

$$u := a \cdot r + e_1 \bmod^{+} \mathrm{q} \tag{3.52}$$

3: Calcular el segundo texto cifrado v:

$$v := b \cdot r + e_2 + \left\lfloor \frac{q}{2} \right\rceil \cdot z \bmod^{+} \mathrm{q} \tag{3.53}$$

4: **return** (u, v)

En el algoritmo de descifrado, a partir de los textos cifrados u y $v$se recupera el mensaje original z. La seguridad del esquema radica en el término de ruido ($\varepsilon = r \cdot e - s \cdot e_1 + e_2$) que ofusca el mensaje, el cual solo puede recuperarse correctamente usando la clave secreta siempre que $\|\varepsilon\| < q/4$.

Para mantener este error dentro de niveles aceptables, es fundamental ajustar adecuadamente los parámetros de las distribuciones de ruido en el algoritmo Kyber. En la versión empleada en este trabajo (Kyber1024), esto se traduce en una tasa de error de 2^{-174} [32].

[0] En Kyber se usa la variante M-LWE, donde $s \in R_q^d$, $A \in R_q^{d \times d}$ y $b = A \cdot s + e \in R_q^d$; el pseudocódigo completo en M-LWE aparece en la sección 3.3.1.4.

Algoritmo 12 Descifrado R-LWE [0]

Entrada: sk, u, v, q, n

Salida: z

1: Calcular el polinomio z' a partir del cual se calcula el mensaje.

$$z' := v - u \cdot s = (r \cdot e - s \cdot e_1 + e_2) + \left\lfloor \frac{q}{2} \right\rceil \cdot z \bmod^{+} \mathrm{q} \tag{3.54}$$

2: Calcular la distancia de cada coeficiente (z'_i) de z':

1.1: Distancia a 0:

$$\mathrm{d}_i(0) := \left\| z'_i \bmod^{\pm} \left\lfloor \frac{q}{2} \right\rceil \right\| \tag{3.55}$$

1.2: Distancia a $\left\lfloor \frac{q}{2} \right\rceil$:

$$\mathrm{d}_i\left(\frac{q}{2}\right) := \left\| \left\lfloor \frac{q}{2} \right\rceil - \mathrm{d}_i(0) \right\| \tag{3.56}$$

3: **if** $\mathrm{d}_i(0) < \mathrm{d}_i\left(\frac{q}{2}\right)$ **then**

4: Descifrar el bit i del mensaje z como un 0.

5: **else**

6: Descifrar el bit i del mensaje z como un 1.

7: **end if**

8: **return** z

En el anexo A se puede ver un breve ejemplo de aplicación numérica de funcionamiento de los algoritmos anteriores.

Uso de M-LWE en Kyber

A partir del siguiente articulo se define la M-LWE que es el esquema de fondo en Kyber [31].

Sean R_q el anillo a trabajar y d la dimensión de la retícula. Se define:

$s \in R_q^d$	(el vector secreto)
$A \in R_q^{d \times d}$	(la matriz pública, muestreada $R_q^{d \times d}$ a partir de una semilla ρ)
$e \in R_q^d$	(el vector de error, con coeficientes "pequeños")
$b \in R_q^d$	(la pareja resultante)
$b = A \cdot s + e \quad \in R_q^d$	

Entonces, el problema M-LWE (como extensión modular de R-LWE) se define como el problema de distinguir muestras (A, b) de parejas uniformes en $R_q^{d \times d} \times R_q^d$.

Como se observa, esta definición resulta relativamente sencilla de entender a partir del ejemplo anterior. Sin embargo, las razones que avalan su seguridad son diferentes: al presentar una estructura menos uniforme, la M-LWE ofrece mejores garantías frente a ataques que explotan la regularidad de los anillos.

Por tanto, la M-LWE es un punto intermedio entre ambos esquemas que permite:

- Escalabilidad en seguridad mediante el parámetro d.
- Mejor eficiencia que en LWE, al poder emplear la NTT sobre anillos.
- Reducción del tamaño de claves, pues la matriz A puede reconstruirse a partir de una semilla.

[0] En Kyber se usa la variante M-LWE, donde $s \in R_q^d$, $A \in R_q^{d \times d}$ y $b = A \cdot s + e \in R_q^d$; el pseudocódigo completo en M-LWE aparece en la sección 3.3.1.4.

- Mayor robustez, dado que su estructura "menos simétrica" que un anillo puro dificulta algunos ataques específicos a R-LWE.

3.3.1.3. Parámetros empleados en Kyber

A continuación, se presenta una tabla con los parámetros del esquema Kyber1024 empleado en este trabajo fin de grado.

	n	k	q	η_1	η_2	d_u	d_v	δ	$pk(bytes)$	$sk(bytes)$	$c(bytes)$
Kyber1024	256	4	3329	2	2	11	5	2^{-174}	3168 (32)	1568	1568

Tabla 3.4: Tabla con los parámetros utilizados por Kyber1024. El valor de $n = 256$ se elige para permitir una escalabilidad sencilla y permitir distintos niveles de seguridad sin perder capacidad de tener un nivel de ruido aceptable. El valor de $k = 4$ fija el tamaño de la retícula e implica una seguridad de 256 bits. El valor de $q = 3329$ es un primo que satisface $n|(q-1)$ para permitir la NTT, los primos anterior y siguiente que también satisfacen esta propiedad conllevan probabilidades de fallo demasiado altas. Los valores η_1 y η_2 definen el ruido y junto a d_u y d_v se usan para equilibrar la seguridad y la tasa de fallos δ. El valor (32) en la llave pública es el tamaño de la semilla necesaria para reconstruirla.

3.3.1.4. Algoritmos principales de Kyber [1]

El algoritmo de generación de llaves genera las llaves pública (pk) y privada (sk) a partir de los parámetros de la tabla 3.4.

- La función Parse(x) se encarga de convertir cadenas de bits a su representación NTT garantizando que los coefientes a_i sean del tamaño adecuado $\log_2(q) \approx 11{,}7$ y no permitiendo desbordamientos $a_i < q$.
- La función $\text{CBD}_\eta(x)$ muestrea el ruido a partir mediante una distribución binomial. Convierte un vector de bits $B \in \mathcal{B}^{64\eta}$ a un polinomio $f \in R_q$.
- La función $\text{Encode}_k(x)$ convierte de un vector de bits $B \in \mathcal{B}^{32l}$ a un polinomio $f \in R_q$.

Algoritmo 13 Generación llaves en Kyber

Salida: $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$, $sk \in \mathcal{B}^{24\cdot k\cdot n/8+96}$

1: Obtener $d, z \in \mathcal{B}^{32}$ de manera aleatoria usando una distribución uniforme.
2: Obtener dos nuevos parámetros ρ, σ expandiendo el valor inicial d:

$$(\rho, \sigma) := \text{SHA3-512}(d) \tag{3.57}$$

Se crean las matrices para realizar las operaciones de los algoritmos de la R-LWE.
3: **for** i=0:k-1 **do**
4: **for** j=0:k-1 **do**
5: $\hat{A}[i][j] := \text{Parse}[\text{SHAKE-128}(\rho, j, i)]$ ▷ Muestreo matriz en el dominio NTT.
6: **end for**
7: **end for**
8: $N := 0$
9: **for** i=0:k-1 **do**
10: $s[i] := \text{CBD}_{\eta_1}[\text{SHAKE-256}(\sigma, N)]$ ▷ Muestreo secreto.
11: $N := N + 1$
12: **end for**
13: **for** i=0:k-1 **do**
14: $e[i] := \text{CBD}_{\eta_1}[\text{SHAKE-256}(\sigma, N)]$ ▷ Muestreo error.
15: $N := N + 1$
16: **end for**
17: Se convierten las magnitudes mediante la NTT para agilizar los cálculos:

$$\begin{aligned} \hat{s} &:= \text{NTT}(s) \\ \hat{e} &:= \text{NTT}(e) \\ \hat{t} &:= \hat{A} \circ \hat{s} + \hat{e} \end{aligned} \tag{3.58}$$

18: Se calcula la llave pública:

$$pk := \text{Encode}_{12}(\hat{t} \bmod^{+} \text{q})||\rho \tag{3.59}$$

▷ Se envía $\hat{b}$ junto con la semilla ρ para el calculo de $\hat{A}$.
▷ Así se reduce el tamaño de la clave enviada.
19: Se calcula la llave secreta:

$$sk := \text{Encode}_{12}(\hat{s} \bmod^{+} \text{q})||pk||\text{SHA3-256}(pk)||z \tag{3.60}$$

▷ Se realiza esta concatenación para cumplir la seguridad IND-CCA2 mediante la TFO.
20: **return** (pk, sk)

En el algoritmo de cifrado Kyber se obtiene el texto cifrado c a partir de la llave pública pk, un mensaje m y una semilla aleatoria γ mediante el uso de la NTT.

- Las funciones $\text{Compress}_q(x, y)$ y $\text{Decompress}_q(x, y)$ se usan para reducir el tamaño de los textos cifrados basándose en el fundamento descrito para los mecanismos basados en la R-LWE.
- La función $\text{Decode}_k(x)$ convierte de un polinomio $f \in R_q$ a un vector de bits $B \in \mathcal{B}^{32l}$.

Algoritmo 14 Cifrado Kyber

Entrada: $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$, $m \in \mathcal{B}^{32}$, $\gamma \in \mathcal{B}^{32}$

Salida: $c \in \mathcal{B}^{d_u\cdot k\cdot n/8+d_v\cdot n/8}$

1: Calcular los parámetros necesarios:
2: $N := 0$
3: **for** i=0:k-1 **do**
4: $\quad r[i] := \text{CBD}_{\eta_1}[\text{SHAKE-256}(\gamma, N)]$ ▷ Muestreo elemento r.
5: $\quad N := N + 1$
6: **end for**
7: **for** i=0:k-1 **do**
8: $\quad e_1[i] := \text{CBD}_{\eta_2}[\text{SHAKE-256}(\gamma, N)]$ ▷ Muestreo del primer error.
9: $\quad N := N + 1$
10: **end for**
11: $e_2[i] := \text{CBD}_{\eta_2}[\text{SHAKE-256}(\gamma, N)]$ ▷ Muestreo del segundo error.
12: Calcular los valores de los textos cifrados:

$$\begin{aligned} \hat{r} &:= \text{NTT}(r) \\ u &:= \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1 \\ v &:= \text{NTT}^{-1}(\hat{t}^T \circ \hat{r}) + e_2 + \text{Decompress}_q[\text{Decode}_1(m), 1] \\ c_1 &:= \text{Encode}_{d_u}[\text{Compress}_q(u, d_u)] \\ c_2 &:= \text{Encode}_{d_v}[\text{Compress}_q(v, d_v)] \end{aligned} \tag{3.61}$$

13: **return** $(c := c1||c2)$

En el algoritmo de encapsulado Kyber a partir de la llave pública pk se obtiene el texto cifrado c y el secreto compartido k.

Algoritmo 15 Encapsulado Kyber

Entrada: $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$

Salida: $c \in \mathcal{B}^{d_u\cdot k\cdot n/8+d_v\cdot n/8}$, $k \in \mathcal{B}^*$

1: Obtener los valores necesarios a partir de la llave pública:

$$\begin{aligned} \hat{t} &:= \text{Decode}_{12}(pk) \\ p &:= pk + 12 \cdot k \cdot n/8 \end{aligned} \tag{3.62}$$

2: Calcular la matriz $\hat{A}^T$ a partir de ρ codificado en la llave pública.
3: Obtener m' de manera aleatoria usando una distribución uniforme.
4: Obtener los parámetros m, κ, γ a partir de m' y la llave pública:

$$\begin{aligned} m &:= \text{SHA3-256}(m') \\ (\kappa, \gamma) &:= \text{SHA3-512}[m||\text{SHA3-256}(pk)] \end{aligned} \tag{3.63}$$

5: Obtener el texto cifrado:

$$c \leftarrow \text{Cifrado Kyber}(pk, m, \gamma) \tag{3.64}$$

6: Calcular el secreto compartido:

$$k := \text{SHAKE-256}[\kappa||\text{SHA3-256}(c)] \tag{3.65}$$

7: **return** (c, k)

En el algoritmo de decapsulado Kyber a partir del texto cifrado c y la llave secreta sk se puede obtener el secreto compartido k.

Algoritmo 16 Decapsulado Kyber

Entrada: $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$, $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$

Salida: $k \in \mathcal{B}^*$

1: Obtener los valores descomprimidos u, v y el valor de la llave secreta s en el dominio NTT:

$$\begin{aligned} u &:= \text{Decompress}_q[\text{Decode}_{d_u}(c, d_u)] \\ v &:= \text{Decompress}_q[\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8, d_v)] \\ \hat{s} &:= \text{Decode}_{12}(sk) \end{aligned} \tag{3.66}$$

2: Obtener el mensaje m' cifrado anteriormente:

$$m' := \text{Encode}_1[\text{Compress}_q\left(v - \text{NTT}^{-1}(\hat{s}^T \circ \text{NTT}(u)), 1\right)] \tag{3.67}$$

Para Garantizar la seguridad ante ataques de canal lateral se vuelve a calcular el texto cifrado.

3: Cifrar m' con la llave pública pk y el parámetro γ para obtener el texto cifrado c':

$$\begin{aligned} pk &:= sk + 12 \cdot k \cdot n/8 \\ h &:= sk + 24 \cdot k \cdot n/8 + 32 \\ (\kappa, \gamma) &:= \text{SHA3-512}[m'||h] \\ c' &\leftarrow \text{Cifrado Kyber}(pk, m', \gamma) \end{aligned} \tag{3.68}$$

4: Comparar los textos cifrados obtenidos, añadiendo un nuevo parámetro z para textos cifrados no válidos.

$$z := sk + 24 \cdot k \cdot n/8 + 64 \tag{3.69}$$

5: **if** $c == c'$ **then**

6: **return** $K := \text{SHAKE-256}[\kappa||\text{SHA3-256}(c)]$ ▷ Mismo secreto compartido.

7: **else**

8: **return** $K := \text{SHAKE-256}[z||\text{SHA3-256}(c)]$ ▷ No distinguible de llaves válidas.

9: **end if**

3.3.2. Fundamentos matemáticos de Saber

Para elaborar esta sección se parte del artículo de Saber adjuntado con la sumisión al NIST [2]. En Saber al igual que en Kyber se trabaja en un anillo de la forma:

$$R_q = \frac{\mathbb{Z}_q}{X^n + 1} \tag{3.70}$$

Con $n = 256$, pero con la diferencia de que como $q = 2^{13}$, no es posible utilizar la NTT, ya que esta requiere un módulo primo adecuado. No obstante, el uso de un módulo en base 2 ofrece ciertas ventajas [33] frente a esquemas basados en M-LWE:

- Uso de LWR: a diferencia de los esquemas basados en LWE, donde es necesario muestrear errores desde una distribución aleatoria, en LWR el error se introduce mediante redondeo determinista.

- Uso de potencias de 2: al trabajar con módulos del tipo 2^k, las reducciones modulares se pueden implementar de forma eficiente mediante operaciones de desplazamiento de bits (bitshifts), eliminando la necesidad de reducciones modulares explícitas.

3.3.2.1. Algoritmos de Toom-Cook y Karatsuba

Dado que en Saber no se cumple la condición $n|(q-1)$ para poder utilizar la NTT, se recurre a los algoritmos de Karatsuba [34] y Toom-Cook-4 [35] para acelerar las multiplicaciones polinómicas. Para ello se sigue la estructura del diagrama de la figura 3.7.

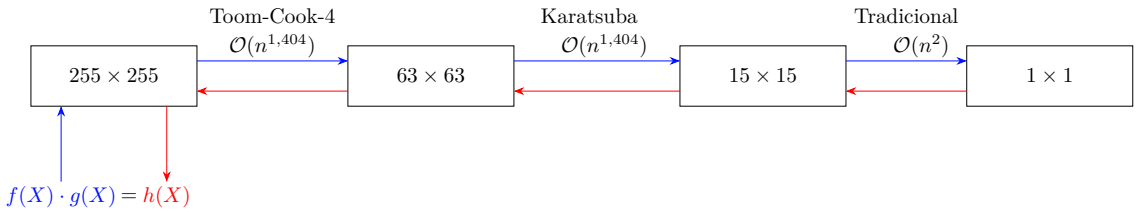

Figura 3.7: Representación del cálculo de un producto de polinomios en Saber. Inicialmente, para polinomios de grado 255, se aplica el algoritmo Toom-Cook-4 que reduce los productos a polinomios de grado 63, ya que ofrece el mejor rendimiento asintótico. Sin embargo, conforme disminuye el tamaño de los polinomios, este rendimiento asintótico se vuelve menos eficiente, por lo que se utiliza el algoritmo de Karatsuba para reducir a polinomios de grado 15, y finalmente se realiza el producto mediante multiplicación tradicional de polinomios.

El algoritmo de Toom-Cook-4 usado en Saber [36] para hacer más eficiente la multiplicación de dos polinomios $c = f \cdot g$ con una complejidad de $\mathcal{O}(n^{\log_4(7)}) \approx \mathcal{O}(n^{1,404})$, sigue los siguientes pasos:

- **Partición**: en este paso se representan los operandos f, g mediante 4 bloques iguales teniendo en cuenta que $n-1$ es el grado del polinomio:

$$\begin{aligned} h(T) &= h(X,T) = h_0(X) + h_1(X) \cdot T + h_2(X) \cdot T^2 + h_3(X) \cdot T^3 \\ T &= X^{n/4} \end{aligned} \tag{3.71}$$

  De esta manera, con la introducción de una nueva variable T se consigue reducir el tamaño efectivo de los polinomios para el paso de la evaluación. La variable X se utiliza como un parámetro. Aplicado a los polinomios f y g con $n = 256$ en Saber:

$$\begin{aligned} f(T) &= f_0(X) + f_1(X) \cdot T + f_2(X) \cdot T^2 + f_3(X) \cdot T^3 \\ g(T) &= g_0(X) + g_1(X) \cdot T + g_2(X) \cdot T^2 + g_3(X) \cdot T^3 \end{aligned} \tag{3.72}$$

  Donde cada f_i y g_i son polinomios de grado 63.

- **Evaluación**: se eligen 7 puntos v_i en los que se evalúan ambos polinomios en T. En [36] se desarrolla un algoritmo eficiente para luego recuperar adecuadamente los coeficientes en la interpolación y para ello, se deben utilizar los siguientes puntos:

$$v_i = \left\{\infty, 2, 1, -1, \frac{1}{2}, -\frac{1}{2}, 0\right\} \tag{3.73}$$

  De esta manera el producto se convierte en un polinomio de grado 6 en T de la forma:

$$c(T) = c_0 + c_1 \cdot T + c_2 \cdot T^2 + c_3 \cdot T^3 + c_4 \cdot T^4 + c_5 \cdot T^5 + c_6 \cdot T^6 \tag{3.74}$$

  Con cada c_i siendo un polinomio de grado 126 de la forma:

$$c_i(X) = c_{i,0} + c_{i,1} \cdot X + c_{i,2} \cdot X^2 + ... + c_{i,126} \cdot X^{126} \tag{3.75}$$

  Analizando este polinomio en todos los puntos se obtiene el producto $c = A_n \cdot \omega$:

$$\begin{pmatrix} c_6(X) \\ c_5(X) \\ c_4(X) \\ c_3(X) \\ c_2(X) \\ c_1(X) \\ c_0(X) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1/64 & 1/32 & 1/16 & 1/8 & 1/4 & 1/2 & 1 \\ 1/64 & -1/32 & 1/16 & -1/8 & 1/4 & -1/2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \omega_6(X) \\ \omega_5(X) \\ \omega_4(X) \\ \omega_3(X) \\ \omega_2(X) \\ \omega_1(X) \\ \omega_0(X) \end{pmatrix} \tag{3.76}$$

  Para implementar las divisiones por potencias de 2 en la práctica se utilizan bitshifts. Mientras que para implementar las multiplicaciones de los polinomios de grado 63 se utiliza el algoritmo de Karatsuba.

- **Interpolación**: se resulve el problema de interpolación $w(v_i) = \omega_i$ invirtiendo la matriz de Vandermonde A_n generada mediante v_i y computando el vector de coeficientes $w = A_n^{-1}c$.

  En este problema de interpolación la matriz se tiene precomputada y una vez obtenidas las inversas se reconstruye el polinomio teniendo en cuenta que este producto se repite para cada uno de los 127 coeficientes del polinomio de salida.

  Por último, se deshace la división en bloques sustituyendo $T = X^{64}$, que equivale a desplazar los coeficientes, y se aplica la reducción modular correspondiente a $X^{256} + 1$.

El algoritmo de Karatsuba aplica el paradigma de "divide y vencerás" para multiplicar los polinomios dividiéndolos en dos partes más pequeñas. No obstante, en la implementación de Saber no se usa Karatsuba "tradicional" pues se divide en cuatro bloques [2]. Para ello, utilizando los polinomios de grado 63 anteriormente divididos en Karatsuba se vuelve a introducir una variable $T = X^{16}$ para particionar:

$$\begin{aligned} f &= f_0 + f_1 \cdot T + f_2 \cdot T^2 + f_3 \cdot T^3 \\ g &= g_0 + g_1 \cdot T + g_2 \cdot T^2 + g_3 \cdot T^3 \end{aligned} \tag{3.77}$$

Entonces su producto queda como:

$$f \cdot g = \begin{cases} T^6 \cdot (f_3 \cdot g_3) + \\ T^5 \cdot (f_3 \cdot g_2 + f_2 \cdot g_3) + \\ T^4 \cdot (f_3 \cdot g_1 + f_2 \cdot g_2 + f_1 \cdot g_3) + \\ T^3 \cdot (f_3 \cdot g_0 + f_2 \cdot g_1 + f_1 \cdot g_2 + f_0 \cdot g_3) + \\ T^2 \cdot (f_2 \cdot g_0 + f_1 \cdot g_1 + f_0 \cdot g_2) + \\ T^1 \cdot (f_1 \cdot g_0 + f_0 \cdot g_1) + \\ T^0 \cdot (f_0 \cdot g_0) \end{cases} \tag{3.78}$$

Descomponiendo adecuadamente en 9 productos de polinomios más pequeños z_i de grado 15 que se realizan mediante el método tradicional:

$$\begin{aligned} z_0 &= f_0 \cdot g_0 \\ z_1 &= f_1 \cdot g_1 \\ z_2 &= f_2 \cdot g_2 \\ z_3 &= f_3 \cdot g_3 \\ z_4 &= (f_0 + f_1)\,(g_0 + g_1) \\ z_5 &= (f_0 + f_2)\,(g_0 + g_2) \\ z_6 &= (f_1 + f_3)\,(g_1 + g_3) \\ z_7 &= (f_2 + f_3)\,(g_2 + g_3) \\ z_8 &= (f_0 + f_1 + f_2 + f_3)\,(g_0 + g_1 + g_2 + g_3) \end{aligned} \tag{3.79}$$

Con estos términos se puede ver como la ecuación 3.79 se puede escribir como una combinación lineal de los términos anteriores:

$$f \cdot g = C_0 + C_1 T + C_2 T^2 + C_3 T^3 + C_4 T^4 + C_5 T^5 + C_6 T^6 \tag{3.80}$$

Donde C_i es:

$$\begin{aligned} C_0 &= z_0 \\ C_1 &= z_4 - z_1 - z_0 \\ C_2 &= z_5 - z_2 - z_0 + z_1 \\ C_3 &= z_8 - (z_7 + z_6 + z_5 + z_4) + z_3 + z_2 + z_1 + z_0 \\ C_4 &= z_6 - z_3 - z_1 + z_2 \\ C_5 &= z_7 - z_3 - z_2 \\ C_6 &= z_3 \end{aligned} \tag{3.81}$$

Es relevante señalar que el algoritmo Karatsuba empleado en Saber tiene la misma complejidad asintótica que el tradicional $\mathcal{O}(n^{\log_2(3)}) \approx \mathcal{O}(n^{1,585})$, demostrable de forma directa mediante el teorema maestro [37].

Una vez hechas estas descomposiciones recursivas se van deshaciendo las particiones hasta llegar al producto deseado. De esta manera, se consigue reducir considerablemente el tiempo de cálculo.

3.3.2.2. Aprendizaje Con Redondeo Modular o Modular Learning With Rounding (Mod-LWR)

+ Los esquemas basados en la LWR [33] emplean operadores de redondeo y, a diferencia de los esquemas de LWE, no requieren muestrear ruido de forma explícita, ya que éste se obtiene a partir de escalar y redondear los coeficientes. Por esta razón, en ocasiones se hace referencia a estos esquemas como una variante "desaleatorizada" de LWE.

Asimismo, al igual que en el caso de Kyber, con el fin de incrementar la seguridad del esquema y mitigar posibles ataques contra la estructura del anillo, se recurre a la versión modular de la LWR, la cual consiste en considerar dicho anillo en un mayor número de dimensiones.

Por tanto, el problema de la Mod-LWRse formula como la dificultad de distinguir entre parejas de muestras uniformes (a, u) y parejas (a, b) generadas de la siguiente manera:

$$\begin{aligned} &a \leftarrow \mathcal{U}(R_q^{l\times l}) \\ &s \leftarrow \beta_\mu(R_q^{l\times 1})) \\ &b \in R_p^{l\times 1} \\ &b = \left\lfloor \frac{p}{q}(A \cdot s) \right\rceil \end{aligned} \tag{3.82}$$

Donde $\mathcal{U}$ representa una distribución uniforme, β_μ representa una distribución binomial centrada en μ y con desviación típica $\sigma = \sqrt{\mu/2}$, y l es el tamaño de la retícula. Antes de pasar a describir los algoritmos de generación de claves es relevante comentar la definición de la función $\texttt{bits}(x, i, j)$:

$$\texttt{bits}(x, i, j) = [x(i-j)]\&(2^j - 1) = \begin{cases} \dfrac{x}{2^{i-j}} \bmod^+ 2^j \text{ si } j \neq i \\ x \bmod^+ 2^j \text{ si } j = i \end{cases} \tag{3.83}$$

En Saber los protocolos de generación de claves se definen de la siguiente manera:

Algoritmo 17 Generación claves M-LWE

Entrada: q, p, n, l, μ

Salida: sk, b, A

1: Muestrear la matriz $A \in R_q^{l\times l}$ a partir de una distribución uniforme y el secreto $sk \in R_q^{l\times 1}$ a partir de la distribución β_μ.
2: Calcular la llave pública $b \in R_p^{l\times 1}$:

$$b := \texttt{bits}(A \cdot s + h, \varepsilon_q, \varepsilon_p) \tag{3.84}$$

3: **return** (sk, b, A)

En este algoritmo los valores ε_i representan el valor $\varepsilon_i = \log_2(i)$ mientras que $h \in R_q$ es un polinomio con todos sus coeficientes con iguales a $2^{\varepsilon_q - \varepsilon_p - 1}$ para emular el comportamiento del redondeo.

Algoritmo 18 Cifrado Mod-LWR

Entrada: pk, A, q, p, t, n, l, μ

Salida: ss', b', c

1: Muestrear otro secreto $s' \in R_q^{l\times 1}$ a partir de la distribución β_μ.

2: Calcular el texto cifrado $b' \in R_p^{l\times 1}$ y la variable intermedia $v' \in R_p$ a partir de la matriz A y la llave pública b:

$$\begin{aligned} b' &:= \texttt{bits}(A^T \cdot s' + h, \varepsilon_q, \varepsilon_p) \\ v' &:= b \cdot \texttt{bits}(s', \varepsilon_p, \varepsilon_p) + h_1 \end{aligned} \tag{3.85}$$

3: Calcular el otro texto cifrado $c \in R_t$ a partir de la variable v':

$$c := \texttt{bits}(v', \varepsilon_p - 1, \varepsilon_t) \tag{3.86}$$

4: Calcular el secreto compartido ss

$$ss' := \texttt{bits}(v', \varepsilon_p, 1) \tag{3.87}$$

5: **return** (ss', b', c)

En este algoritmo se generan los texto cifrados para que en el descifrado se obtenga el mismo secreto compartido. El valor $h_1 \in R_q$ es un polinomio con todos sus coeficientes con iguales a $2^{\varepsilon_q - \varepsilon_p - 1}$ para emular el comportamiento del redondeo.

Algoritmo 19 Descifrado Mod-LWR

Entrada: sk, b', c, p, t

Salida: ss

1: Calcular la variable intermedia $v \in R_p$

$$v := b'^T \cdot \texttt{bits}(s, \varepsilon_p, \varepsilon_p) + h_1 \tag{3.88}$$

2: Calcular el valor del secreto compartido ss

$$ss := \texttt{bits}(v - 2^{\varepsilon_p - \varepsilon_t - 1} \cdot c + h_2), \varepsilon_p, 1 \tag{3.89}$$

3: **return** ss

El valor $h_2 \in R_q$ es un polinomio con todos sus coeficientes con iguales a $2^{\varepsilon_p - 2} - 2^{\varepsilon_p - \varepsilon_t - 2}$ para emular el comportamiento del redondeo. Es relevante destacar que para FireSaber [2], el valor de las constantes corresponde a los de la tabla 3.5, con los cuales se logra una baja probabilidad de fallo.

Una vez definidos los algoritmos basados en Mod-LWR para el intercambio de claves en Saber, resulta fundamental demostrar que ambas partes obtienen efectivamente el mismo secreto compartido. Siguiendo el análisis de [38], la probabilidad de coincidencia del secreto depende de la distancia entre las variables v y v'. En particular, se cumple que

$$d = \|v - v'\| \quad \Rightarrow \quad \begin{cases} d > \dfrac{p}{4}\left(1 + \dfrac{1}{t}\right) = c_1, & \text{el secreto difiere.} \\ d < \dfrac{p}{4}\left(1 - \dfrac{1}{t}\right) = c_2, & \text{el secreto coincide.} \\ c_2 \leq d \leq c_1, & \text{el secreto coincide con probabilidad intermedia.} \end{cases} \tag{3.90}$$

Si se le añade a $\Delta v = v - v'$ una distribución de error $e_r \in R_p$ en el rango $[-p/4t, p/4t)$, se puede calcular la probabilidad $1 - \delta$ de que ambas partes lleguen al mismo secreto compartido como:

$$1 - \delta = \Pr\left[\|\Delta v + e_r\| < \frac{p}{4}\right] \tag{3.91}$$

En la Mod-LWR esta probabilidad se generaliza para que se exprese unicamente en función de las distribuciones de error como se puede ver en el siguiente teorema.

Teorema 2 *Sea una A una matriz en $R_q^{l\times l}$, s, s' vectores en $R_q^{l\times 1}$ muestreados como se describe en los algoritmos anteriores. Se definen e y e' como los errores de redondeo introducidos al reescalar y redondear $A\cdot s$ y $A^T\cdot s'$, es decir:*

$$\begin{aligned} \texttt{bits}(A\cdot s+h,\varepsilon_q,\varepsilon_p) &= \frac{p}{q}\cdot A\cdot s+e \\ \texttt{bits}(A^T\cdot s'+h,\varepsilon_q,\varepsilon_p) &= \frac{p}{q}\cdot A^T\cdot s'+e' \end{aligned} \tag{3.92}$$

Sea $e_r\in R_q$ in polinomio con los coeficientes distribuidos uniformemente en el rango $[-p/4t, p/4t]$. Si se define:

$$\delta = Pr\left[\left\|\left(s'^T\cdot e-e'^T\cdot s+e_r\right)\ \ mod^+ p\right\|_\infty > \frac{p}{4}\right] \tag{3.93}$$

entonces tras ejecutar el protocolo de intercambio de claves en Saber, ambas partes acuerdan un secreto compartido de n-bits con probabilidad $1-\delta$.

Demostración 2 *En la demostración de este teorema se utilizan dos pasos, el primero [33] para demostrar que las condiciones propuestas son equivalentes a las de la ecuación 3.91 y el segundo paso de elaboración propia para efectivamente mostrar que ambos secretos coinciden:*

Paso 1: Equivalencia de condiciones *Se despejan los valores de b y b' mediante la ecuación 3.92:*

$$\begin{aligned} b &= \frac{p}{q}\cdot A\cdot s+e \\ b' &= \frac{p}{q}\cdot A^T\cdot s'+e' \end{aligned} \tag{3.94}$$

A continuación se despejan v y v':

$$\begin{aligned} v &= \frac{p}{q}s'^T\cdot A\cdot s+h_1+s'^T\cdot e\ \ mod^+ p \\ v' &= \frac{p}{q}s'^T\cdot A\cdot s+h_1+e'^T\cdot s\ \ mod^+ p \end{aligned} \tag{3.95}$$

Simplemente restando se obtiene Δv y se claramente se ve la equivalencia a la ecuación 3.93.

Paso 2: Igualdad de secretos *En este paso se hace la demostración empleando los parámetros de FireSaber de la tabla 3.5. Sea $Q_8\in R_{128}$:*

$$Q_8 = \left\lfloor\frac{v'}{8}\right\rfloor \tag{3.96}$$

Como se puede definir v' a través del operador redondeo hacia abajo si se tiene en cuenta un resto $r\in R_8$:

$$v' = 8\left\lfloor\frac{v'}{8}\right\rfloor + r \tag{3.97}$$

Teniendo en cuenta que c se encuentra en R_{64} existe un bit de información, ω, de Q_8 que se esta perdiendo y por tanto, se cumple que:

$$c = \left\lfloor\frac{v'}{8}\right\rfloor\ \ mod^+ 64 = 64\cdot\omega+\left\lfloor\frac{v'}{8}\right\rfloor\ \ mod^+ 64 \tag{3.98}$$

Por tanto, el primer secreto compartido ss' se puede expresar como:

$$ss' = \left\lfloor\frac{8\cdot(64\cdot\omega+c)+r}{512}\right\rfloor\ \ mod^+ 2 = \left\lfloor\omega+\frac{8\cdot c+r}{512}\right\rfloor\ \ mod^+ 2 \tag{3.99}$$

Donde como $c < 64$ y $r < 8$ se obtiene que este secreto solo depende de ω:

$$ss' = \left\lfloor \omega + \frac{511}{512} \right\rfloor \ mod^+ 2 = \omega \ mod^+ 2 \tag{3.100}$$

Ahora solo queda comprobar que efectivamente el secreto compartido ss también acaba teniendo este valor. Partiendo de la definición de ss:

$$ss = \left\lfloor \frac{v - 8 \cdot c + h_2}{512} \right\rfloor \ mod^+ 2 \tag{3.101}$$

Sustituyendo v mediante la definición $\Delta v = v - v'$ y teniendo en cuenta el desarrollo realizado para v' realizado en las ecuaciones anteriores se obtiene:

$$ss = \left\lfloor \frac{[8 \cdot (64 \cdot \omega + c) + r + \Delta] - 8 \cdot c + h_2}{512} \right\rfloor \ mod^+ 2 \tag{3.102}$$

Despejando:

$$ss = \left\lfloor \omega + \frac{r + \Delta + h_2}{512} \right\rfloor \ mod^+ 2 \tag{3.103}$$

Donde justamente si se hace el cambio de notación $e_r := r$ se obtiene la condición de la ecuación 3.93, el método falla si:

$$e_r + \Delta > (512 - h_2) = 260 > \frac{p}{4} = 256 \tag{3.104}$$

3.3.2.3. Parámetros empleados en Saber

A continuación, se presenta una tabla con los parámetros del esquema FireSaber empleado en este trabajo fin de grado.

	n	l	q	p	t	μ	δ	$pk(bytes)$	$sk(bytes)$	$c(bytes)$
FireSaber	256	4	2^{13}	2^{10}	2^6	6	2^{-165}	1312	3040 (1760)	1472

Tabla 3.5: Tabla con los parámetros utilizados por FireSaber. Se elige el valor $n = 256$ para permitir la escalabilidad de Saber mediante el parámetro $l = 4$, que marca el tamaño de las matrices sobre las que se trabaja $R_q^{k \times k}$. Los parámetros q, p y t tienen esos valores para que la probabilidad de fallo δ sea baja. El valor μ representa el tamaño de la binomial a partir de la cual se muestrea la llave secreta sk.

3.3.2.4. Algoritmos principales de Saber [2]

El algoritmo de generación de llaves en Saber genera las llaves pública (pk) y privada (sk) a partir de los parámetros de la tabla 3.5. Teniendo en cuenta que $\varepsilon_i = \log_2(i)$.

- La función `POLVEC`$_x$`2BS`(y) convierte un vector $y \in R_x^{l \times 1}$ en una cadena de bytes de longitud $l \cdot k \cdot 256/8$ con $x = 2^k$.
- El operador $\circ$ denota la multiplicación estándar de una matriz $\in R_x^{l \times l}$ por un vector $\in R_x^l$. Donde uno de los polinomios se multiplica utilizando el algoritmo de la figura 3.7.
- La función `HammingWeight`(x) devuelve la distancia de Hamming del vector x, es decir, el número de simbolos distintos de 0 en x.
- Se define el símbolo ε_i como $\varepsilon_i = \log_2(i)$.
- Los valores de las constantes h, h_1 y h_2 se pueden consultar en la sección 3.3.2.2.

Algoritmo 20 Generación llaves en Saber

Salida: $pk \in \mathcal{B}^{n \cdot l \cdot \varepsilon_p/8+32}$, $sk \in \mathcal{B}^{n \cdot l \cdot \varepsilon_q/8+n \cdot l \cdot \varepsilon_p/8+96}$

1: Muestrear las semillas de la matriz A, secreto y un parámetro auxiliar z, $seed_A$, $seed_s$ y $z \in B^{32}$ de manera aleatoria usando una distribución uniforme
2: Modificar la semilla $seed_A$ mediante hashing:

$$seed_A := \texttt{SHAKE-128}(seed_A, 32) \tag{3.105}$$

3: Se genera la matriz A a partir de la semilla:

$$buf := \texttt{SHAKE-128}(seed_A, l^2 \cdot n \cdot \varepsilon_q/8) \tag{3.106}$$

▷ Se genera buf un vector de $l^2 \cdot n$ cadenas de longitud ε_q.

4: $k := 0$
5: **for** i=0:l-1 **do**
6: **for** j=0:l-1 **do**
7: **for** k=0:n-1 **do**
8: $A[i, j][k] := buf[k]$
9: $k := k + 1$
10: **end for**
11: **end for**
12: **end for**
13: Se genera el secreto s a partir de la distribución binomial:

$$buf := \texttt{SHAKE-128}(seed_s, l \cdot n \cdot \mu/8) \tag{3.107}$$

▷ Se genera buf un vector de $2 \cdot l \cdot n$ cadenas de $\mu/2$ bits.

14: $k := 0$
15: **for** i=0:l-1 **do**
16: **for** j=0:n-1 **do**
17: $s[i, j] := \texttt{HammingWeight}(buf[k]) - \texttt{HammingWeight}(buf[k + 1]) \bmod^{+} \mathrm{q}$
18: $k := k + 2$
19: **end for**
20: **end for**
21: Se calcula el parámetro b:

$$b := \left(A^T \circ s + h \bmod^{+} \mathrm{q}\right) / 2^{\varepsilon_q - \varepsilon_p} \tag{3.108}$$

22: se calcula la llave pública

$$pk := seed_A || \texttt{POLVEC}_q\texttt{2BS}(b) \tag{3.109}$$

23: Se calcula la llave secreta

$$sk := z || \texttt{SHA3-256}(pk) || pk || \texttt{POLVEC}_q\texttt{2BS}(s) \tag{3.110}$$

24: **return** (pk, sk)

En el algoritmo de cifrado de Saber se obtiene el texto cifrado c a partir de la llave pública pk, un mensaje m y una semilla aleatoria γ.

- La función $\texttt{BS2POLVEC}_x(y)$ convierte una cadena de bytes y de longitud $l \cdot k \cdot 256/8$ en un vector en $R_x^{l\times 1}$.
- La función $\texttt{POL}_x\texttt{2BS}(y)$ convierte un polinomio $y \in R_x$ en una cadena de bytes de longitud $k \cdot 256/8$.

Algoritmo 21 Cifrado Saber

Entrada: $pk \in \mathcal{B}^{n\cdot l\cdot \varepsilon_p/8+32}$, $m \in \mathcal{B}^{32}$, $\gamma \in \mathcal{B}^{32}$

Salida: $c \in \mathcal{B}^{n\cdot l\cdot \varepsilon_p/8+n\cdot \varepsilon_t/8}$

1: Se extrae la matriz A y la llave pública pk' a partir de pk.
2: Se genera un nuevo secreto s' de manera similar que en el algoritmo de generación de llaves.
3: Se calcula el parámetro b':

$$b' := \left(A \circ s' + h \bmod^{+} \mathrm{q}\right) /2^{\varepsilon_q - \varepsilon_p} \tag{3.111}$$

4: Se obtiene el parámetro b de la llave pública:

$$b := \texttt{BS2POLVEC}_p(pk') \tag{3.112}$$

5: se calcula el parámetro auxiliar v':

$$v' := b^T \circ (s' \bmod^{+} \mathrm{p}) \bmod^{+} \mathrm{p} \tag{3.113}$$

6: Se calcula el texto cifrado c

$$c := (v' - m \cdot 2^{\varepsilon_p - 1} + h_1 \bmod^{+} \mathrm{p})/2^{\varepsilon_p - \varepsilon_t} \tag{3.114}$$

7: **return** $c := \texttt{POL}_T\texttt{2BS}(c)||\texttt{POLVEC}_p\texttt{2BS}(b')$

En el algoritmo de encapsulado Saber a partir de la llave pública pk se obtiene el texto cifrado c y el secreto compartido k.

Algoritmo 22 Encapsulado Saber

Entrada: $pk \in \mathcal{B}^{n\cdot l\cdot \varepsilon_p/8+32}$

Salida: $c \in \mathcal{B}^{n\cdot l\cdot \varepsilon_p/8+n\cdot \varepsilon_t/8}$, $k \in \mathcal{B}^*$

1: Se muestrea el mensaje $m \in \mathcal{B}^{32}$ de manera aleatoria usando una distribución uniforme.
2: Se hashea el mensaje m y la llave pública pk para crear una variable buf:

$$\begin{aligned} m &:= \texttt{SHA3-256}(m) \\ hash_{pk} &:= \texttt{SHA3-256}(pk) \\ buf &:= hash_{pk}||m \end{aligned} \tag{3.115}$$

3: Se hashea el buffer para obtener dos cadenas del mismo tamaño y así inicializar la semilla aleatoria γ:

$$(\gamma||r) := \texttt{SHA3-512}(buf) \tag{3.116}$$

4: Se ejecuta el cifrado para obtener el texto cifrado c:

$$c := \texttt{Cifrado Saber}(pk, m, \gamma) \tag{3.117}$$

5: Se calcula el secreto compartido k:

$$k := \texttt{SHA3-256}(r||\texttt{SHA3-256}(c)) \tag{3.118}$$

6: **return** (c, k)

En el algoritmo de decapsulado Saber a partir del texto cifrado c y la llave secreta sk se puede obtener el secreto compartido k.

Algoritmo 23 Decapsulado Saber

Entrada: $c \in \mathcal{B}^{n\cdot l\cdot \varepsilon_p/8+n\cdot\varepsilon_t/8}$, $sk \in \mathcal{B}^{n\cdot l\cdot\varepsilon_q/8+n\cdot l\cdot\varepsilon_p/8+96}$

Salida: $k \in \mathcal{B}^*$

1: Obtener el valor de la llave secreta sk' y el valor z de sk

2: Se obtiene el polinomio del secreto s

$$s := \texttt{BS2POLVEC}_q(sk') \tag{3.119}$$

3: se descompone el texto cifrado en sus partes

$$(c_m||ct) := c \tag{3.120}$$

4: A partir de estos valores se calcula el mensaje cifrado anteriormente m':

$$\begin{aligned} c_m &:= c_m \cdot 2^{\varepsilon_p-\varepsilon_t} \\ b' &:= \texttt{BS2POLVEC}_p(ct) \\ m' &:= (b'^T \circ (s \bmod^+ \mathrm{p}) - c_m + h_2 \bmod^+ \mathrm{p})/2^{\varepsilon_p-1} \end{aligned} \tag{3.121}$$

5: Una vez obtenido el mensaje se transforma en una cadena m :

$$m := \texttt{POL}_2\texttt{2BS}(m') \tag{3.122}$$

6: Se calculan las variables γ y r:

$$(\gamma||r) := \texttt{SHA3-512}(buf) \tag{3.123}$$

7: Se calcula nuevamente el texto cifrado c' para comprobar que es igual que c y cumplir la TFO:

$$c' := \texttt{Cifrado Saber}(pk, m, \gamma) \tag{3.124}$$

8: **if** c==c' **then**

9: Asignar k:

$$k := \texttt{SHA3-256}(r||\texttt{SHA3-256}(c')) \tag{3.125}$$

10: **else**

11: Asignar k:

$$k := \texttt{SHA3-256}(z||\texttt{SHA3-256}(c')) \tag{3.126}$$

12: **end if**

13: **return** k

3.3.3. Fundamentos matemáticos de Hamming Quasi-Cyclic (HQC)

A diferencia de Kyber y Saber, que se basan en problemas matemáticos relacionados con retículas, el esquema Hamming Quasi-Cyclic (HQC) [39] se fundamenta en la criptografía basada en códigos.

El proceso de la criptografía basada en código se asemeja a las técnicas de corrección de errores empleadas en comunicaciones digitales, pues implica recuperar un mensaje a partir de un código corrupto [22]. No obstante, como el atacante no conoce la estructura del código no es factible recuperar el mensaje sin los secretos.

En HQC, se trabaja en el espacio vectorial ν de dimensión n sobre el cuerpo binario $\mathbb{F}_2$, es decir, $\mathbb{F}_2^n$. Un elemento de ν también puede considerarse como un vector fila en el anillo de polinomios $\mathcal{R} = \mathbb{F}_2[x]/(C^n - 1)$, al cual es isomórfico.

3.3.3.1. Códigos cuasi-cíclicos y problema de decodificación por síndrome

Para elaborar esta sección se ha utilizado el artículo sobre HQC, así como la sumisión al NIST [39]. Los códigos cíclicos y el problema de decodificación por síndrome constituyen el fundamento matemático que sostiene el funcionamiento de HQC, de manera análoga a cómo la M-LWE y la Mod-LWR son los fundamentos de Kyber y Saber, respectivamente.

No obstante, antes de pasar a definirlos es necesario introducir algunas definiciones preliminares:

- Se habla de peso de Hamming o Hamming Weight, $\omega(x)$, de un vector x para referirse al número de cordenadas que tiene este distintas de 0.
- Se habla de que un entero primo n es primitivo si se cumple que la factorización del polinomio $(X^n - 1)/(X - 1)$ es irreducible en $\mathcal{R}$.
- Para dos elementos $u, v \in \nu$, se define su producto de manera similar que en $\mathcal{R}$. Donde $w = u \cdot v$

$$w_k = \sum_{i+j=k \bmod^+ n} u_i \cdot v_j \ \forall\ k \in 0, \ldots, n-1 \tag{3.127}$$

- Se define la matriz circulante, $\texttt{rot}(h)$, para $h \in \nu$ como la matriz donde cada columna i denota $h \cdot x^i$:

$$v = (v_0, \ldots, v_{n-1}) \in \mathbb{F}_2^n$$

$$\texttt{rot}(v) = \begin{pmatrix} v_0 & v_{n-1} & \ldots & v_1 \\ v_1 & v_0 & \ldots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} & v_{n-2} & \ldots & v_0 \end{pmatrix} \tag{3.128}$$

A partir de esta matriz también se puede definir el producto de dos elementos $u, v \in \nu$ como:

$$u \cdot v = u \cdot \texttt{rot}(v)^T = (\texttt{rot}(u) \cdot v^T)^T = v \cdot u \tag{3.129}$$

Con estas definiciones en cuenta se puede definir un código lineal C de longitud n y de dimensión k, denotado mediante $[n, k]$, como un subespacio vectorial de R de dimensión k. Para cada código lineal se definen dos matrices:

1. Matriz generadora $G \in \mathbb{F}_2^{k \times n}$:

$$C = \{m \cdot G,\ m \in \mathbb{F}_2^k\} \tag{3.130}$$

2. Matriz de paridad $H \in \mathbb{F}_2^{(n-k) \times n}$, también definida como la matriz generadora del código dual $C^\perp$, u ortogonal de C:

$$\begin{aligned} C &= \{v \in \mathbb{F}_2^n \mid H \cdot v^T = 0\} \\ C^T &= \{u \cdot H,\ u \in \mathbb{F}_2^k\} \end{aligned} \tag{3.131}$$

A partir de la matriz de paridad se define el síndrome de una palabra $v \in \mathbb{F}_2^n$ como:

$$H \cdot v^T \rightarrow \text{ si } v \in C \rightarrow H \cdot v^T = 0 \tag{3.132}$$

Para un código $C[n,k] \in \mathcal{R}$ se define la distancia mínima d dentro del subespacio como:

$$d = \min_{\substack{u,v \in C \\ u \neq v}} \omega(u-v) \tag{3.133}$$

Para esta distancia mínima, un código lineal puede corregir hasta δ errores. Esto se debe a que, al recibir un vector $r = u + e$ que corresponde a un código u con ruido e, si el número de errores es demasiado grande, la decodificación podría producir otro vector $v \neq u$. El número máximo de errores se deduce fácilmente como:

$$\delta = \left\lfloor \frac{d-1}{2} \right\rfloor. \tag{3.134}$$

que es equivalente a decir que los errores no produzcan vectores más alejados de u que la distancia mínima. Por esta razón, cuando se habla de un código es relevante tener en cuenta el parámetro d.

Con estas definiciones se introduce el concepto de códigos cuasi-cíclicos (QC), los cuales surgen como una solución al problema de los tamaños de clave excesivamente grandes en esquemas clásicos basados en códigos, como el de McEliece. La introducción de los códigos cuasi-cíclicos permite, por tanto, reducir significativamente el tamaño de las claves [40].

Un código cuasi-cíclico se puede representar como un vector

$$c = (c_0, \ldots, c_{s-1}) \in \mathbb{F}_2^{sn}, \quad c_i \in \mathcal{R}. \tag{3.135}$$

Además, un código lineal $C[sn,k,d]$ es QC si, para cualquier $c = (c_0, \ldots, c_{s-1}) \in C$, se cumple que una rotación circular de dicho vector permanece dentro del código:

$$c \cdot X = (c_0 \cdot X, \ldots, c_{s-1} \cdot X) \in C. \tag{3.136}$$

Un código QC es sistemático de índice s y razón $1/s$ sí su matriz de paridad H es de la forma:

$$H = \begin{pmatrix} I_n & 0 & \ldots & 0 & A_0 \\ 0 & I_n & \ldots & 0 & A_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & I_n & A_{s-2} \end{pmatrix} \in \mathcal{R}^{(s-1)n \times sn} \tag{3.137}$$

donde I_n es la matriz identidad y A_i son matrices circulantes de tamaño $n \times n$.

En relación con el problema de seguridad para un código lineal QC y los enteros n, w y s, se define la distribución cuasi-lineal del síndrome $\texttt{s-QCSD}(n,w)$ como la función que selecciona aleatoriamente una matriz de paridad $H \in \mathbb{F}_2^{(sn-n)\times sn}$, correspondiente a un código sistemático QC de índice s y razón $1/s$, junto con un vector $x \in \mathbb{F}_2^{sn}$ tal que $\omega(x_i) = w$ para todo $i = 0, \ldots, s-1$, y devuelve la pareja (H, Hx^T).

En base a esta distribución y de manera similar que en Kyber o Saber el problema fundamental consiste en discernir si esta pareja (H, Hx^T) proviene de una distribución uniforme o no, lo cual es equivalente a encontrar el vector x cuyo peso de Hamming es w [39].

3.3.3.2. Códigos Reed-Solomon y Reed-Muller

Para implementar los algoritmos de codificación y decodificación de un código C se usan códigos concatenados de Reed-Muller y Reed-Solomon [3]. Estos códigos concatenados están compuestos de un código externo $[n_e, k_e, d_e] \in \mathbb{F}_q$ y código interno $[n_i, k_i, d_i] \in \mathbb{F}_2$ de tal manera que exista la biyeción:

$$\mathbb{F}_q^{n_e} \cong \mathbb{F}_2^N \tag{3.138}$$

para lo cual debe cumplirse que $q = 2^{k_i}$ y $N = n_e \cdot n_i$. De esta forma el código externo se convierte en un código binario de parámetros $[N = n_e \cdot n_i, K = k_e \cdot k_i, D \geq d_e \cdot d_i]$.

La combinación de estos métodos se utiliza de tal manera que Reed-Solomon maneja la estructura algebraica, lo que permite codificar una mayor cantidad de información gracias a su buen escalado y a su eficacia en bloques grandes, mientras que Reed-Muller trabaja de forma eficiente con cadenas binarias y ofrece mejores propiedades de codificación, como una mayor distancia mínima, lo que lo hace más tolerante a errores aleatorios [41]. Esta estructura se puede ver en la figura 3.8.

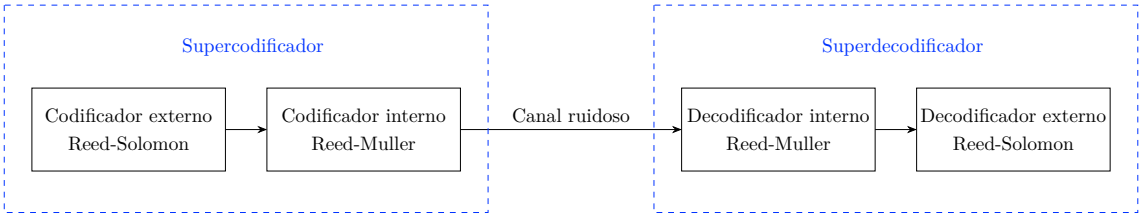

Figura 3.8: Representación del funcionamiento mediante códigos concatenados en HQC.

Para el código externo se utiliza un código de Reed-Solomon de dimensión $k_e = 32$ sobre $\mathbb{F}_{256}$ mientras que para el código interno se utiliza un código de Reed-Muller $[128, 8, 64]$ duplicando cada bit de 5 veces para aumentar el número máximo de errores admitido:

$$C[128, 8, 64] \rightarrow C[640, 8, 320] \tag{3.139}$$

Un código de Reed-Solomon, denotado mediante $RS[n, k, d_{min}]$ con sus elementos en $\mathbb{F}_q$, tiene los siguientes parámetros:

- Longitud de bloque: $n = q - 1$, donde q es la potencia de un número primo p.
- Número de dígitos de paridad: $n - k = 2\delta$, donde δ es la capacidad de corrección del código y k el número de bits de información.
- La distancia mínima definida a partir de los parámetros anteriores como $d_{min} = 2\delta + 1$
- El polinomio generador $g(x)$ del código para un elemento primitivo $\alpha \in \mathbb{F}_{2^m}$ se define como:

$$g(x) = (x + \alpha)\left(x + \alpha^2\right) \cdots \left(x + \alpha^{2\delta}\right) \tag{3.140}$$

No obstante, como usar estos códigos directamente implica trabajar en espacios con dimensiones elevadas, se utilizan los códigos acortados de Reed–Solomon al fijar algunas partes de los mensajes a 0 y aprovechando este hecho para reducir el tamaño al comprimir la información. Para el nivel de seguridad empleado en HQC-5 se usa el código `RS-S3`$[90, 32, 49] \in \mathbb{F}_{2^8}$ con el polinomio primitivo:

$$1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8 \tag{3.141}$$

y el polinomio generador $g_3(x)$ que se encuentra en el anexo B.

- **Codificación mediante códigos Reed–Solomon**: Sea un mensaje $u = (u_0, u_1, \ldots, u_{k-1})$, cuyo polinomio asociado es $u(x) = u_0 + u_1 x + \cdots + u_{k-1} x^{k-1}$. En la forma sistemática de codificación, los k símbolos más a la derecha corresponden al mensaje original, mientras que los $n - k$ primeros símbolos constituyen los bits de paridad.

  Por tanto, la codificación se lleva a cabo siguiendo los pasos que se describen a continuación:

  - Obtener el producto:
  $$y(x) = u(x) \cdot x^{n-k} \tag{3.142}$$
  - Calcular el resto $b(x)$ de $y(x)$ dividido por el polinomio generador $g(x)$.

- Obtener el polinomio codificado $c(x)$:

$$c(x) = b(x) + y(x) \tag{3.143}$$

- **Decodificación mediante códigos Reed-Solomon**: se usa el mismo procedimiento que para códigos completos, puesto que los códigos acortados no son más que versiones comprimidas. Para ello, considerando el código $\texttt{RS}[n, k, d_{min}]$, con $n = 2^m - 1$.

Suponiendo que una palabra, $v(x) = v_0 + v_1 + \cdots v_{n-1}x^{n-1}$, es transmitida por un canal "ruidoso" se obtiene una palabra, $r(x) = r_0 + r_1x \cdots r_{n-1}x^{n-1}$,que es la consecuencia de superponer ruido $e(x) = e_0 + e_1x \cdots e_{n-1}x^{n-1}$, a $v(x)$:

$$r(x) = v(x) + e(x) \tag{3.144}$$

Definiendo el conjunto de síndromes $S_1, \ldots, S_{2\delta}$ como $S_i = r(\alpha^i)$, con α un elemento primitivo en $\mathbb{F}_{2^m}$ se obtiene:

$$r(\alpha^i) = e(\alpha^i) + v(\alpha^i) = e(\alpha^i),\ v(\alpha^i) = 0 \text{ porque pertenece al código} \tag{3.145}$$

Si $e(x)$ tiene t errores en posiciones $j_1, \ldots, j_t$ se obtiene el siguiente sistema de ecuaciones donde la incógnita es α:

$$\begin{cases} S_1 = e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + \cdots + e_{j_t}\alpha^{j_t} \\ S_2 = e_{j_1}(\alpha^{j_1})^2 + e_{j_2}(\alpha^{j_2})^2 + \cdots + e_{j_t}(\alpha^{j_t})^2 \\ S_3 = e_{j_1}(\alpha^{j_1})^3 + e_{j_2}(\alpha^{j_2})^3 + \cdots + e_{j_t}(\alpha^{j_t})^3 \\ \vdots \\ S_{2\delta} = e_{j_1}(\alpha^{j_1})^{2\delta} + e_{j_2}(\alpha^{j_2})^{2\delta} + \cdots + e_{j_t}(\alpha^{j_t})^{2\delta} \end{cases} \tag{3.146}$$

Sustituyendo $\beta_i = \alpha^{j_i}$:

$$\begin{cases} S_1 = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \cdots + e_{j_t}\beta_t \\ S_2 = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \cdots + e_{j_t}\beta_t^2 \\ S_3 = e_{j_1}\beta_1^3 + e_{j_2}\beta_2^3 + \cdots + e_{j_t}\beta_t^3 \\ \vdots \\ S_{2\delta} = e_{j_1}\beta_1^{2\delta} + e_{j_2}\beta_2^{2\delta} + \cdots + e_{j_t}\beta_t^{2\delta} \end{cases} \tag{3.147}$$

Si se define el polinomio de localización de errores $\sigma(x)$ como la función donde sus raíces son $\{\beta_1^{-1}, \beta_2^{-1}, \ldots, \beta_t^{-1}\}$:

$$\sigma(x) = (1 + \beta_1 x)(1 + \beta_2 x) \cdots (1 + \beta_t x) = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t \tag{3.148}$$

este polinomio se calcula utilizando el algoritmo de Berlekamp [42]:

$$\begin{pmatrix} S_1 & S_2 & \cdots & S_\delta \\ S_2 & S_3 & \cdots & S_{\delta+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_\delta & S_{\delta+1} & \cdots & S_{2\delta-1} \end{pmatrix} \begin{pmatrix} \sigma_\delta \\ \sigma_{\delta-1} \\ \vdots \\ \sigma_1 \end{pmatrix} = \begin{pmatrix} -S_{\delta+1} \\ -S_{\delta+2} \\ \vdots \\ -S_{\delta+\delta} \end{pmatrix}. \tag{3.149}$$

Por tanto, una vez obtenido el polinomio se pueden calcular sus raíces y así conocer los puntos donde están localizados los errores.

A continuación se define el polinomio $Z(x)$ que sirve para obtener los valores de los errores e_{jl} al ya conocerse las raíces:

$$\begin{cases} \sigma_0 = S_0 = 1 \\ Z(x) = \sum_{i=0}^{t} \sum_{j=0}^{i} S_j \cdot \sigma_{i-j} x^i \\ \\ e_{jl} = \dfrac{Z(\beta_l^{-1})}{\prod_{\substack{i=1 \\ i \neq l}}^{t} \left(1 + \beta_i \beta_l^{-1}\right)} \end{cases} \tag{3.150}$$

Por último, se descifra el mensaje v:

$$v(x) = r(x) - e(x) \tag{3.151}$$

Un código de Reed-Muller, denotado mediante $RM(r, m)$, con m y r enteros con $0 \leq r \leq m$ con los siguientes parámetros

- Longitud del código $n = 2^m$
- Dimensión $k = \sum_{i=0}^{r} \binom{m}{i}$
- Distancia mínima $d_{min} = 2^{m-r}$

En HQC se utiliza $RM(1, 7)$ correspondiente con el código $[128, 8, 64]$.

- **Codificación mediante códigos Reed-Muller**: la codificación se realiza para dar estructura algebraica al mensaje m. Para ello, se siguen los siguientes pasos:
  - Multiplicar por la matriz generadora G cada símbolo obtenido anteriormente por Reed-Solomon:
    $$c = m \cdot G \tag{3.152}$$
  - Duplicar el mensaje para aumentar la resistencia a los errores para mejorar las propiedades de distancia y así obtener un código de Reed-Muller duplicado $[640, 8, 320]$. Aunque esto podría comprometer la seguridad del esquema, en la sección de seguridad se discutirá como en la práctica el esquema HQC sigue siendo seguro.
- **Decodificación mediante códigos Reed-Muller**: para descifrar el mensaje se siguen los siguientes pasos:
  - Computar la función $F : \mathbb{F}_2^3 \to \{5, 3, 1, -1, -3, -5\}$ que sirve para transformar los códigos duplicados aprovechando esta duplicidad para eliminar los errores. Véase el ejemplo de un bloque de 5 repeticiones x_1, x_2, x_3, x_4 y x_5.
    $$(-1)^{x_1} + (-1)^{x_2} + (-1)^{x_3} + (-1)^{x_4} + (-1)^{x_5} \tag{3.153}$$
  - Realizar la transformada de Hadamard sobre el vector F creado, donde se define la matriz de Hadamart H_n como:
    $$H_{2^k} = \begin{pmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{pmatrix} \text{ con } H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.154}$$
    Al multiplicar esta matriz por F, $\hat{F}$ se obtiene el peso de cada fila:
    $$\hat{F} = H_n \cdot F \tag{3.155}$$
- Con los pesos obtenidos de cada fila se busca aquel con mayor valor, denominado pico, y se escoge esa fila como código. Si hay varios picos con el mismo valor se escoge aquel con menor índice módulo 128.

- Descifrar el mensaje como la fila de la matriz de Hadamant escogida si el valor del pico es negativo sumar el vector de unos a la palabra y si es positivo no realizar modificaciones. El convenio para descifrar el mensaje (realizar antes de sumar el vector de 1):

$$\begin{cases} +1 \to 0 \\ -1 \to 1 \end{cases} \tag{3.156}$$

En el anexo C se encuentra un ejemplo numérico de codificación mediante estos códigos para mostrar la función de este algoritmo.

Por último, es importante señalar que, al igual que en *Saber*, en HQC se emplean los algoritmos de Karatsuba y Toom-Cook para acelerar la multiplicación de polinomios, aunque con ciertas variaciones en el número de divisiones. En particular, se utiliza el algoritmo de Karatsuba tradicional, dividiendo los polinomios en dos mitades, y Toom-Cook-3, dividiéndolos en tres partes.

3.3.3.3. Parámetros empleados en HQC

A continuación, se presenta una tabla con los parámetros del esquema HQC-5 empleado en este trabajo fin de grado.

	n_1	n_2	n	k	ω	$\omega_r = \omega_e$	δ	$pk(bytes)$	$sk(bytes)$	$c(bytes)$
FireSaber	90	640	57637	256	131	159	2^{-256}	7237	7333 (32)	14421

Tabla 3.6: Tabla con los parámetros utilizados por HQC-5. Mediante n_1 se denota la longitud del código de Reed-Solomon, mediante n_2 se denota la longitud del código Reed-Muller y n es el menor número primitivo que sea mayor que $n_1 \cdot n_2$. El párametro k es la dimensión de los códigos. Mediante ω, ω_r, y ω_e se denotan los pesos de hamming de los vectores de la llave privada, de la aleatoriedad $r1, r2$ y del error respectivamente. δ denota la probabilidad de fallo para obtener el mismo secreto compartido.

3.3.3.4. Algoritmos principales de HQC [3]

El algoritmo de generación de llaves en HQC genera las llaves pública (pk) y privada (sk) a partir de los parámetros de la tabla 3.6.

- La función `SHAKE256.absorb` implementa el mecanismo de absorción o inicialización de la función como se puede ver en la figura 3.5. Recibe como argumento el valor inicial de la función.
- La función `SHAKE256.squeeze` implementa el mecanismo de aplastado o obtención de la salida como se puede ver en la figura 3.5. Recibe como argumento el valor obtenida de la fase de absorción y la longitud de salida deseada.
- La función $\texttt{SampleFixedWeightVector}_{\$}(x, y)$ genera vectores en base al estado x y peso fijo y con distribución uniforme usando rechazo aleatorio, es decir, si un vector no cumple la condición de peso de Hamming se rechaza y se genera uno nuevo.
- La función `SampleVect` muestrea un vector al azar a partir de `SHAKE256.squeeze`.
- Los separadores dentro de las funciones de hashing se usan para evitar colisiones entre cadenas.

Algoritmo 24 Generación llaves en HQC

Salida: $pk \in \mathcal{B}^{\lceil n/8 \rceil + 32}$, $sk \in \mathcal{B}^{\lceil n/8 \rceil + \lceil k/8 \rceil + 96}$

1: Muestrear la semilla $seed \in \mathcal{B}^{32}$ a partir de una distribución uniforme.
2: Inicializar el mecanismo de absorción

$$ctx := \texttt{SHAKE256.absorb}(seed) \tag{3.157}$$

3: Absorber los primeros 32 bytes generados mediante la función `keccak` en $seed$ y los siguientes 256 bytes en σ.

$$(seed||\sigma) := \texttt{SHAKE256.absorb}(ctx, (32, 256)) \tag{3.158}$$

4: Computar los valores de las semillas de la llave pública y de la llave privada a partir de la semilla:

$$(seed_{sk}||seed_{pk}) := \texttt{SHA3-512}(seed||\text{I\_SEPARATOR}) \tag{3.159}$$

5: Se crea la salida de la absorción para generar las llaves pública y privada:

$$\begin{aligned} ctx_{pk} &:= \texttt{SHAKE256.absorb}(seed_{pk}) \\ ctx_{sk} &:= \texttt{SHAKE256.absorb}(seed_{sk}) \end{aligned} \tag{3.160}$$

6: Se muestrean vectores x, y cada uno de peso de Hamming ω en base al estado ctx_{pk}:

$$(x||y) := \texttt{SampleFixedWeightVector}_{\$}(ctx_{sk}, \omega) \tag{3.161}$$

7: Se muestrea el vector h para calcular la llave pública:

$$h := \texttt{SampleVect}(ctx_{pk}) \tag{3.162}$$

8: Se calcula el valor s:

$$s := x + h \cdot y \tag{3.163}$$

9: Se calcula la llave pública:

$$pk := seed_{pk}||s \tag{3.164}$$

10: Se calcula la llave privada:

$$sk := pk||seed_{sk}||\sigma||seed \tag{3.165}$$

11: **return** (pk, sk)

En el algoritmo de cifrado de HQC se obtiene el texto cifrado c a partir de la llave pública pk, un mensaje m y una semilla aleatoria γ.

- La función `SampleFixedWeightVector` genera vectores de peso fijo mediante una función de selección por lo cual se introduce un pequeño sesgo, pero en el artículo de HQC se describe como esto no supone un problema [3].
- La función $\texttt{Truncate}(x, y)$ mantiene los y bits menos significativos de x.
- La función `Encode` aplica la codificación mediante Reed-Solomon y Reed-Muller descrita anteriormente.

Algoritmo 25 Cifrado HQC

Entrada: $pk \in \mathcal{B}^{\lceil n/8 \rceil + 32}$, $m \in \mathcal{B}^{256}$, $\gamma \in \mathcal{B}^{16}$

Salida: $c \in \mathcal{B}^{n \cdot l \cdot \varepsilon_p / 8 + n \cdot \varepsilon_t / 8}$

1: Se obtiene el valor de la semilla de la llave pública $seed_{pk}$ y de s a partir de la llave pública pk.

2: Se calcula el estado a partir de la semilla

$$ctx_{pk} := \texttt{SHAKE256.absorb}(seed_{pk}) \tag{3.166}$$

3: Se obtiene el parámetro h para posteriormente calcular el texto cifrado:

$$h := \texttt{SampleVect}(ctx) \tag{3.167}$$

4: Se inicializa el estado a partir de la aleatoriedad:

$$ctx_\theta := \texttt{SHAKE256.absorb}(\gamma) \tag{3.168}$$

5: Obtener los valores para el cálculo del texto cifrado:

$$(r_2||e||r_1) := \texttt{SampleFixedWeightVector}(ctx_\theta) \tag{3.169}$$

6: Se calcula el valor de u:

$$u := r_1 + h \cdot r_2 \tag{3.170}$$

7: Se calcula el valor de v: ▷ l es el valor de bits con los que se trabaja $n - n_1 \cdot n_2$

$$v := \texttt{Encode}(m) + \texttt{Truncate}(s \cdot r_2 + e, l) \tag{3.171}$$

8: **return** $c := u||v$

En el algoritmo de encapsulado HQC a partir de la llave pública pk se obtiene el texto cifrado c y el secreto compartido k.

Algoritmo 26 Encapsulado HQC

Entrada: $pk \in \mathcal{B}^{\lceil n/8 \rceil + 32}$

Salida: $c \in \mathcal{B}^{\lceil n/8 \rceil + \lceil n_1 \cdot n_2 \rceil + 16}$, $k \in \mathcal{B}^*$

1: Obtener el mensaje m y la aleatoriedad $salt$ a partir de sus distribuciones uniformes aleatorias.

2: Calcular el secreto compartido y la aleatoriedad γ:

$$(k||\gamma) := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{3.172}$$

3: Calcular el texto cifrado:

$$c := \texttt{Cifrado Saber}(pk, m, \gamma)||salt \tag{3.173}$$

4: **return** (c, k)

En el algoritmo de decapsulado HQC a partir del texto cifrado c y la llave secreta sk se puede obtener el secreto compartido k.

- La función `Decode` aplica la decodificación mediante Reed-Solomon y Reed-Muller descrita anteriormente.

Algoritmo 27 Decapsulado HQC

Entrada: $c \in \mathcal{B}^{\lceil n/8 \rceil + \lceil n_1 \cdot n_2 \rceil + 16}$, $sk \in \mathcal{B}^{\lceil n/8 \rceil + \lceil k/8 \rceil + 96}$

Salida: $k \in \mathcal{B}^*$

1: Desempaquetar la llave privada sk y el texto cifrado c.
2: Se inicializa el vector para decodificar el mensaje:

$$ctx := \texttt{SHAKE256.absorb}(seed_{sk}) \tag{3.174}$$

3: Se muestrea el vector y:

$$y := \texttt{SampleFixedWeightVector}_{\$}(ctx_{sk}, \omega) \tag{3.175}$$

4: Se computa el mensaje:

$$m' := \texttt{Decode}(v - \texttt{Truncate}(u \cdot y, l)) \tag{3.176}$$

5: Se calcula el secreto compartido:

$$(k'||\gamma') := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{3.177}$$

6: Calcular el texto cifrado:

$$c' := \texttt{Cifrado Saber}(pk, m', \gamma')||salt \tag{3.178}$$

▷ A continuación se describe la implementación correcta para cumplir seguridad IND-CCA2 debido a que la implementación actual cuando no consigue el mismo texto cifrado simplemente falla la función.
7: **if** c==c' **then**
8: Asignar k:

$$(k||-) := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{3.179}$$

9: **else**
10: Asignar k:

$$k := \texttt{SHA3-256}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||\sigma||c||\text{J\_SEPARATOR}) \tag{3.180}$$

11: **end if**
12: **return** k

3.4. Fundamentos de seguridad de los algoritmos asimétricos

Al trabajar con algoritmos de cifrado, resulta esencial garantizar su seguridad. Para ello, es necesario establecer con precisión las condiciones que la sustentan. Por ello, a continuación se presentan las definiciones de seguridad relevantes, así como los criterios para la generación de números aleatorios criptográficamente seguros.

3.4.1. Indistinguibilidad bajo ataque de texto cifrado adaptable IND-CCA2

Para redactar esta sección se usa el artículo [43] donde se describen las diferentes nociones de seguridad para esquemas de clave pública o cifrado asimétrico. Es importante recalcar que el texto cifrado de desafío es aquel que se genera normalmente en el protocola sin que intervenga el atacante.

En el cifrado se buscan dos objetivos:

1. **Indistinguibilidad del cifrado** (IND): incapacidad del adversario para poder aprender información sobre el texto plano x subyacente a un texto cifrado de desafío y.
2. **No maleabilidad del cifrado** (NM): incapacidad del adversario para dado un texto cifrado de desafío y, para obtener otro texto cifrado y' de tal manera que los textos planos x y x' estén fuertemente relacionados.

Un atacante puede realizar tres tipos de ataques teóricos, es decir, estos ataques no son ataques físicos al sistema si no :

1. **Ataque de texto plano** (CPA): el adversario puede obtener textos cifrados de textos planos a su elección. Un esquema es seguro frente a CPA si el adversario no puede distinguir cuál de dos mensajes elegidos ha sido cifrado en el texto cifrado de desafío. Es decir, puede realizar la fase 1 de la figura 3.9.
2. **Ataque de texto cifrado no adaptable** (CCA1): el adversario tiene acceso a un oráculo de descifrado, es decir, puede enviar textos cifrados y obtener sus correspondientes textos planos. Sin embargo, este acceso es limitado: el oráculo solo puede consultarse antes de recibir el texto cifrado de desafío. Es decir, puede realizar la fase 1 y la fase 2 del ataque mientras no se dé la fase de desafío como se puede ver en la figura 3.9.
3. **Ataque de texto cifrado adaptable** (CCA2): es similar al ataque CCA1, pero el acceso al oráculo de descifrado no se pierde. La única limitación es que no se puede utilizar esta función con el propio texto cifrado de desafío. Se denomina adaptable debido a que los textos cifrados utilizados pueden depender del propio texto cifrado de desafío. Es decir, puede realizar todas las fases de la figura 3.9.

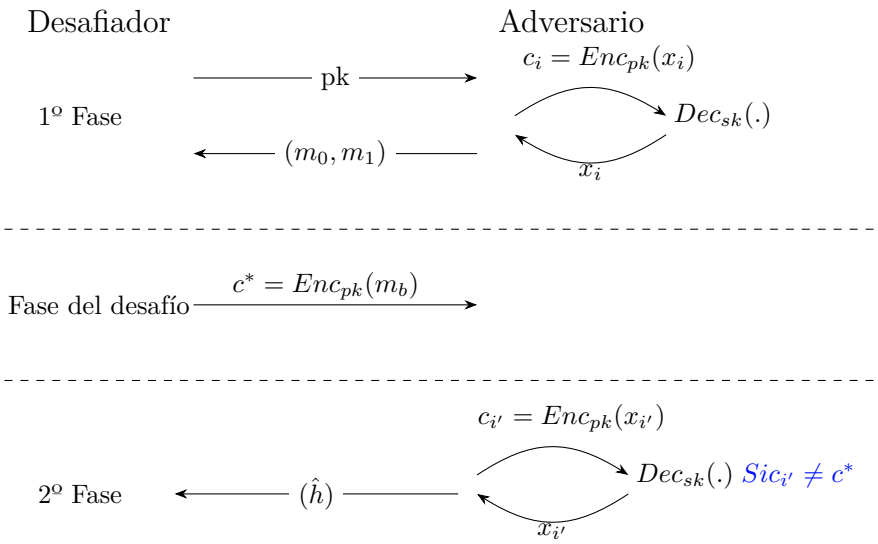

Figura 3.9: Representación de las fases posibles de ataque mediante los oráculos de cifrado y descifrado.

3.4.1.1. Transformadas Fujisaki-Okamoto TFO

[44]

3.4.2. Generación de números aleatorios criptográficamente seguros

En esta sección se analiza el proceso de generación de números aleatorios, con el fin de evitar vulnerabilidades que puedan comprometer la seguridad de los algoritmos criptográficos. Un generador predecible permitiría a un atacante inferir los valores producidos, exponiendo así la información de los secretos.

3.4.2.1. Generación de números aleatorios en Windows

[45]

3.4.2.2. Generación de números aleatorios en el PSOC

3.5. Garantías de seguridad de los algoritmos postcuánticos

En esta sección se hace un breve resumen de las garantías de seguridad de los diferentes algoritmos sin entrar en el desarrollo matemático que justifica el mismo.

3.5.1. Garantías de seguridad de Kyber

3.5.1.1. Seguridad esperada en Kyber

3.5.1.2. Análisis respecto a ataques conocidos en el esquema Kyber

3.5.2. Garantías de seguridad de Saber

3.5.2.1. Seguridad esperada en Saber

3.5.2.2. Análisis respecto a ataques conocidos en el esquema Saber

3.5.3. Garantías de seguridad de HQC

3.5.3.1. Seguridad esperada en HQC

3.5.3.2. Análisis respecto a ataques conocidos en el esquema HQC

Capítulo 4

Desarrollo

4.1. Implementación comunicación serie

4.1.1. Parámetros generales y formato mensajes

4.1.2. Implementación en el ordenador

4.1.3. Implementación en el microprocesador

4.2. Interfaz para los algoritmos de cifrado asimétrico

4.2.1. Interfaz Kyber

4.2.2. Interfaz Saber

4.2.3. Interfaz HQC

4.3. Implementación algoritmos en el PC

4.3.1. Compilación en librerías

4.3.2. Diagrama de uso

4.3.3. Diagrama funcional

4.3.4. Diagrama de clases

4.4. Implementación de algoritmos en el microcontrolador

4.4.1. Diagrama de uso

4.4.2. Diagrama funcional

4.5. Implementación del intercambio de claves. Creación del secreto compartido

Hablar de los modelos de comunicaciones a implementar .... (msg teams)

4.5.1. Modelo 1

4.5.2. Modelo 2

4.5.3. Modelo 3

4.6. Tests de rendimiento realizados

Capítulo 5

Resultados y discusión

En este capítulo se muestran los resultados obtenidos de aplicar las rutinas desarrolladas con anterioridad.

5.1. Resultados

5.1.1. (No sé si lo metere) Resultado de ejecución de los algoritmos clásicos

5.1.1.1. Rivest-Shamir-Adleman (RSA)

5.1.1.2. Criptografía en Curvas Elípticas (ECC)

5.1.2. Resultados de ejecución de los algoritmos post-cuánticos

5.1.2.1. Kyber

5.1.2.2. Saber

5.1.2.3. HQC

5.1.3. Resultados de los distintos modelos de comunicación

5.1.3.1. Modelo 1

5.1.3.2. Modelo 2

5.1.3.3. Modelo 3

5.2. Discusión

5.2.1. Comparativa entre los distintos algoritmos post-cuánticos analizados

5.2.2. Comparativa entre los distintos modelos de comunicación

Capítulo 6

Conclusiones

Se presentan a continuación las conclusiones del proyecto y desarrollos futuros para mejorar la implementación.

6.1. Conclusión

Una vez finalizado el proyecto...

6.2. Desarrollos futuros

Un posible desarrollo...

Apéndice A

Ejemplo Aprendizaje Con Errores en Anillos (R-LWE)

Sea el espacio de trabajo en $R_{17} = \mathbb{Z}_{17}[X]/\left(X^2+1\right)$ con un mensaje $z \in \{0,1\}^2$ y la distribución del error $e \in \{-1,0,1\}$.

En el primer paso se generan a, s y e:

$$\begin{array}{l} a[X] = 3 + 4X \\ s[X] = 1 + 0X \\ e[X] = -1 + 1X \end{array} \tag{A.1}$$

Una vez inicializados los parámetros, se procede al cálculo de b. La reducción módulo X^2+1 equivale a sustituir X^2 por -1 cada vez que aparezca.

$$b[X] = a[X] \cdot s[X] + e[X] = 2 + 5X \tag{A.2}$$

Con la clave pública calculada $a||b$ se puede cifrar un mensaje z, pero antes se generan los valores de z, r, e_1 y e_2:

$$\begin{array}{l} z[X] = 1 + 0X \\ r[X] = 1 + 1X \\ e_1[X] = 0 + 1X \\ e_2[X] = -1 + 0X \end{array} \tag{A.3}$$

Con estos valores se calculan los textos cifrados:

$$\begin{array}{l} u[X] = a[X] \cdot r[X] + e_1[X] = 16 + 8X \\ v[X] = b[X] \cdot r[X] + e_2[X] + \left\lfloor \dfrac{q}{2} \right\rceil \cdot z[X] = 5 + 7X \end{array} \tag{A.4}$$

Por último, se comprueba que el mensaje se descifra correctamente.

$$z'[X] = v[X] - u[X] \cdot s[X] = 6 + 16X \rightarrow \begin{cases} z'_0: & \begin{array}{l} \mathrm{d}_0(0) = 6 \\ \mathrm{d}_0(9) = 3 \end{array} \\ z'_1: & \begin{array}{l} \mathrm{d}_1(0) = 1 \\ \mathrm{d}_1(9) = 8 \end{array} \end{cases} \tag{A.5}$$

Con estas distancia se obtiene que $z = (1,0)$. No obstante, aunque este descifrado se comprueba que se cumple que el error no supera la magnitud límite $q/4 = 4{,}25$.

$$\varepsilon[X] = r[X] \cdot e[X] - s[X] \cdot e_1[X] + e_2[X] = 14 + 16X \tag{A.6}$$

Para cumplirse la distancia a 0 debe ser menor a $q/4$ para cada coeficiente:

$$\begin{array}{l} \mathrm{d}_0(0) = 3 \\ \mathrm{d}_1(0) = 1 \end{array} \tag{A.7}$$

Apéndice B

Polinomios generadores acortados de Reed-Solomon en HQC

A continuación se exponen los polinomios generadores de Reed-Solomon [3]:

$$\begin{aligned} RS-S1:\ g_1(x) = {} & 89 + 69x + 153x^2 + 116x^3 + 176x^4 + 117x^5 + 111x^6 + 75x^7 \\ & + 73x^8 + 233x^9 + 242x^{10} + 233x^{11} + 65x^{12} + 210x^{13} \\ & + 21x^{14} + 139x^{15} + 103x^{16} + 173x^{17} + 67x^{18} + 118x^{19} \\ & + 105x^{20} + 210x^{21} + 174x^{22} + 110x^{23} + 74x^{24} + 69x^{25} \\ & + 228x^{26} + 82x^{27} + 255x^{28} + 181x^{29} + x^{30} \end{aligned}$$

$$\begin{aligned} RS-S2:\ g_2(x) = {} & 45 + 216x + 239x^2 + 24x^3 + 253x^4 + 104x^5 + 27x^6 + 40x^7 \\ & + 107x^8 + 50x^9 + 163x^{10} + 210x^{11} + 227x^{12} + 134x^{13} \\ & + 224x^{14} + 158x^{15} + 119x^{16} + 13x^{17} + 158x^{18} + x^{19} \\ & + 238x^{20} + 164x^{21} + 82x^{22} + 43x^{23} + 15x^{24} + 232x^{25} \\ & + 246x^{26} + 142x^{27} + 50x^{28} + 189x^{29} + 29x^{30} \\ & + 232x^{31} + x^{32} \end{aligned}$$

$$\begin{aligned} RS-S3:\ g_3(x) = {} & 49 + 167x + 49x^2 + 39x^3 + 200x^4 + 121x^5 + 124x^6 + 91x^7 \\ & + 240x^8 + 63x^9 + 148x^{10} + 71x^{11} + 150x^{12} + 123x^{13} \\ & + 87x^{14} + 101x^{15} + 32x^{16} + 215x^{17} + 159x^{18} + 71x^{19} \\ & + 201x^{20} + 115x^{21} + 97x^{22} + 210x^{23} + 186x^{24} + 183x^{25} \\ & + 141x^{26} + 217x^{27} + 123x^{28} + 12x^{29} + 31x^{30} + 243x^{31} \\ & + 180x^{32} + 219x^{33} + 152x^{34} + 239x^{35} + 99x^{36} + 141x^{37} \\ & + 4x^{38} + 246x^{39} + 191x^{40} + 144x^{41} + 8x^{42} + 232x^{43} \\ & + 47x^{44} + 27x^{45} + 141x^{46} + 178x^{47} + 130x^{48} + 64x^{49} \\ & + 124x^{50} + 47x^{51} + 39x^{52} + 188x^{53} + 216x^{54} + 48x^{55} \\ & + 199x^{56} + 187x^{57} + x^{58} \end{aligned}$$

Apéndice C

Ejemplo de codificación mediante polinomios de Reed-Solomon y Reed-Muller

Se trabaja con códigos de menor dimensión para hacer viable mostrar el ejemplo numérico.

Sea el código de Reed-Solomon $RS[7, 3, 5]$:

- Cuerpo $\mathbb{F}_{2^3}$ con el polinomio $x^3 + x + 1$
- $n = 7$
- $k = 3$
- $d_{min} = 5$

Sea el código de Reed-Muller $RM(1, 2) = [4, 3, 2]$:

- $n = 4$
- $k = 3$
- $d_{min} = 2$
- Se duplica el código 2 veces.

Como se trabaja en Reed-Solomon se trabaja en campos de Galois, en este caso $GF(8)$ con el polinomio primitivo $x^3 + x + 1$ se obtiene el polinomio generador:

$$g(x) = \prod_{i=1}^{4} \left(x + \alpha^i\right) = (\alpha+1) + (\alpha^2+\alpha)x + \alpha x^2 + (\alpha+1)x^3 + x^4 = 3 + 6x + 2x^2 + 3x^3 + x^4 \quad \text{(C.1)}$$

La equivalencia de α a binario y decimal en $GF(8)$ se puede obtener en la tabla C.1:

α^i	α^i (polinomial)	α^i (binario)	α^i (decimal)
α^0	1	001	1
α^1	α	010	2
α^2	α^2	100	4
α^3	$\alpha + 1$	011	3
α^4	$\alpha^2 + \alpha$	110	6
α^5	$\alpha^2 + \alpha + 1$	111	7
α^6	$\alpha^2 + 1$	101	5
α^7	1	001	1

Tabla C.1: Potencias de α en GF(2^3) con representación en forma de potencia, binaria y decimal.

Sea el mensaje u el mensaje a codificar:

$$u = (u_0, u_1, u_2) = (\alpha^1, \alpha^2, 1) \rightarrow u(x) = 2 + 4x + x^2 \tag{C.2}$$

Codificando mediante Reed-Solomon:

1. Obtener $y(x)$

$$y(x) = x^{n-k} = x^4 \cdot u(x) = 2x^4 + 4x^5 + x^6 \tag{C.3}$$

2. Obtener $b(x)$ como el resto de $y(x)/(g(x)$ mediante división larga, como $GF(8)$ es de característica 2 la inversa aditiva de α es $-\alpha = \alpha$. Por tanto $b(x)$:

$$\begin{array}{rccccccc|l}
y(x) = & x^6 & +\alpha^2x^5 & +\alpha x^4 & +0 & +0 & +0 & +0 & g(x) = x^4 + \alpha^3x^3 + \alpha x^2 + \alpha^4 x + \alpha^3 \\
+ & x^6 & +\alpha^3x^5 & +\alpha x^4 & +\alpha^4x^3 & +\alpha^2x^2 & +0 & +0 & x^2 + \alpha^5 x + \alpha \\
\hline
 & & \alpha^5x^5 & +0 & +\alpha^4x^3 & +\alpha^2x^2 & +0 & +0 & \\
+ & & \alpha^5x^5 & +\alpha x^4 & +\alpha^6x^3 & +\alpha^2x^2 & +\alpha x & +0 & \\
\hline
 & & & \alpha x^4 & +\alpha^3x^3 & +0 & +\alpha x & +0 & \\
 & & + & \alpha x^4 & +\alpha^4x^3 & +\alpha^2x^2 & +\alpha^5 x & +\alpha^4 & \\
\hline
 & & & b(x) = & \alpha^6x^3 & +\alpha^2x^2 & +\alpha^6 x & +\alpha^4 &
\end{array} \tag{C.4}$$

3. El mensaje codificado c quedaría como:

$$c(x) = \alpha^4 + \alpha^6 x + \alpha^2x^2 + \alpha^6x^3 + \alpha x^4 + \alpha^2x^5 + x^6 \rightarrow c = (6, 5, 4, 5, 2, 4, 1) \tag{C.5}$$

Ahora mediante Reed-Muller se codifica el mensaje c:

- Calcular la matriz generadora G para el código[4,3,2]:

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \tag{C.6}$$

- Codificar cada una de las palabras en c para obtener el código u. Para ello se trabaja con los símbolos o números de c en su codificación binaria como se puede ver en la tabla C.1.

$$\begin{array}{lcccccl}
c_0 = 6 & : & 110 & \rightarrow & c_0 \cdot G & = & 1100 \\
c_1 = 5 & : & 101 & \rightarrow & c_1 \cdot G & = & 1010 \\
c_2 = 4 & : & 100 & \rightarrow & c_2 \cdot G & = & 1001 \\
c_3 = 5 & : & 101 & \rightarrow & c_3 \cdot G & = & 1010 \\
c_4 = 2 & : & 010 & \rightarrow & c_4 \cdot G & = & 0101 \\
c_5 = 4 & : & 100 & \rightarrow & c_5 \cdot G & = & 1001 \\
c_6 = 1 & : & 001 & \rightarrow & c_6 \cdot G & = & 0011
\end{array} \tag{C.7}$$

- Se repite cada bloque 2 veces para producir el código final v:

$$v = 110011001010101010011001101010100101010110011001001100011 \tag{C.8}$$

Ahora al mensaje se le añadiría ruido aleatorio r para que el atacante no conozca el mensaje enviado:

$$v' = v + r = 110010001010101001100100101110100101110110011011000010011 \tag{C.9}$$

Para decodificar con Reed-Muller se siguen los siguientes pasos:

- Dividir el mensaje v' en bloques

$$\begin{array}{rcl} \text{Bloque}_0 & = & 11001000 \\ \text{Bloque}_1 & = & 10101010 \\ \text{Bloque}_2 & = & 01100100 \\ \text{Bloque}_3 & = & 10111010 \\ \text{Bloque}_4 & = & 01011101 \\ \text{Bloque}_5 & = & 10011011 \\ \text{Bloque}_6 & = & 00010011 \end{array} \tag{C.10}$$

- Para cada bloque se aplica la función F, en la tabla C.2 se ve el ejemplo del Bloque_0 y se obtienen los siguientes resultados para cada bloque:

$$\begin{array}{rcl} F(\text{Bloque}_0) & = & (-2,0,2,2) \\ F(\text{Bloque}_1) & = & (-2,2,-2,2) \\ F(\text{Bloque}_2) & = & (2,-2,0,2) \\ F(\text{Bloque}_3) & = & (-2,2,-2,0) \\ F(\text{Bloque}_4) & = & (0,-2,2,-2) \\ F(\text{Bloque}_5) & = & (-2,2,0,-2) \\ F(\text{Bloque}_6) & = & (2,2,0,-2) \end{array} \tag{C.11}$$

Bit	1	2	3	4
Mitad 1	1	1	0	0
Mitad 2	1	0	0	0
F	-2	0	2	2

Tabla C.2: Ejemplo de aplicación de F sobre el Bloque_0.

- Se construye la matriz de Hadamant H_4 y se multiplica cada fila:

$$H_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{C.12}$$

$$\begin{array}{rcl} H_4 \cdot F(\text{Bloque}_0) & = & (2,-2,-6,-2) \\ H_4 \cdot F(\text{Bloque}_1) & = & (0,-8,0,0) \\ H_4 \cdot F(\text{Bloque}_2) & = & (2,2,-2,6) \\ H_4 \cdot F(\text{Bloque}_3) & = & (-2,-6,2,-2) \\ H_4 \cdot F(\text{Bloque}_4) & = & (-2,6,-2,-2) \\ H_4 \cdot F(\text{Bloque}_5) & = & (-2,-2,2,-6) \\ H_4 \cdot F(\text{Bloque}_6) & = & (2,2,6,-2) \end{array} \tag{C.13}$$

- Viendo que los picos se descifra para cada bloque su mensaje como la fila de H_4 en la cual se da el mayor valor en valor absoluto. Donde si este valor es negativo se cambia el signo de la fila. Por tanto, el código descifrado por Reed-Muller es r:

$$\begin{array}{rcccccl} r_0 = 6 & : & 110 & \rightarrow & c_0 \cdot G & = & 1100 \\ r_1 = 5 & : & 101 & \rightarrow & c_1 \cdot G & = & 1010 \\ r_2 = 3 & : & 011 & \rightarrow & c_2 \cdot G & = & 0110 \\ r_3 = 5 & : & 101 & \rightarrow & c_3 \cdot G & = & 1010 \\ r_4 = 2 & : & 010 & \rightarrow & c_4 \cdot G & = & 0101 \\ r_5 = 4 & : & 100 & \rightarrow & c_5 \cdot G & = & 1001 \\ r_6 = 1 & : & 001 & \rightarrow & c_6 \cdot G & = & 0011 \end{array} \tag{C.14}$$

A continuación se decodifica con Reed-Solomon:

- Se calcula el número de síndromes posibles 2δ:

$$\delta = \frac{d_{min} - 1}{2} = 2 \tag{C.15}$$

- Se calculan los 2δ síndromes para $r(x)$:

$$\begin{array}{l} r(x) = \alpha^4 + \alpha^6 x + \alpha^3 x^2 + \alpha^6 x^3 + \alpha x^4 + \alpha^2 x^5 + x^6 \leftarrow r = (6,5,3,5,2,4,1) \\ S_1 = r(\alpha^1) = 1 \\ S_2 = r(\alpha^2) = \alpha^4 \\ S_3 = r(\alpha^3) = \alpha^2 \\ S_4 = r(\alpha^4) = 1 \end{array} \tag{C.16}$$

- Se construye el polinomio σ que como máximo puede tener δ errores:

$$\sigma = \prod_{i=1}^{2} (1 + \beta_i x) = 1 + \sigma_1 x + \sigma_2 x^2 \rightarrow \begin{pmatrix} S_1 & S_2 \\ S_2 & S_3 \end{pmatrix} \begin{pmatrix} \sigma_2 \\ \sigma_1 \end{pmatrix} = \begin{pmatrix} -S_3 \\ -S_4 \end{pmatrix} \tag{C.17}$$

$$\begin{pmatrix} 1 & \alpha^4 \\ \alpha^4 & \alpha^2 \end{pmatrix} \begin{pmatrix} \sigma_2 \\ \sigma_1 \end{pmatrix} = \begin{pmatrix} \alpha^2 \\ 1 \end{pmatrix} \rightarrow 1 + \alpha^2 x \tag{C.18}$$

- Se obtienen las raíz β^{-1} de σ

$$\beta^{-1} = \alpha^2 \tag{C.19}$$

- Se calcula el polinomio $Z(x)$:

$$Z(x) = \sum_{i=0}^{2\delta-1} \sum_{j=0}^{i} S_j \cdot \sigma_{i-j} x^i = 1 + (S_1 + \sigma_1)x + (S_2 + S_1\sigma_1)x^2 + (S_3 + S_2\sigma_1)x^3 \tag{C.20}$$

$$Z(x) = 1 + \alpha^6 x + \alpha x^2 + x^3 \tag{C.21}$$

- La magnitud del error:

$$e_{jl} = \frac{Z(\beta_l^{-1})}{\prod\limits_{\substack{i=1 \\ i \neq l}}^{t} \left(1 + \beta_i \beta_l^{-1}\right)} \tag{C.22}$$

$$e = \frac{\alpha}{1} = \alpha \tag{C.23}$$

- Se corrigen los errores en la posición $\log_\alpha \beta^{-1}$ y se obtiene el mensaje c:

$$c = r + e = (6,5,4,5,2,4,1) \tag{C.24}$$

Como se puede ver el decodificador ha sido capaz de descifrar el código.

Bibliografía

[1] L. Ducas et al. R. Avanzi, J. Bos. CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation (version 3.01). Technical report, CRYSTALS Project, January 2021. NIST PQC Round 3 submission.

[2] A. Basso, J.M. Bermudo Mera, J.P. Anvers, et al. SABER: Mod-LWR based KEM (Round 3 Submission), 2020. NIST PQC Round 3 submission.

[3] N. Aragon et al. P. Gaborit, C. Aguilar-Melchor. Hamming quasi-cyclic (hqc) specification. `https://pqc-hqc.org/resources.html`, August 2025.

[4] M.J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 00:08:00 2015.

[5] Tikzmaker. `https://tikzmaker.com/editor`, 2024.

[6] Infineon Technologies AG. CY8CPROTO-063-BLE PSoC 6 BLE Prototyping Kit. `https://www.infineon.com/cms/en/product/evaluation-boards/cy8cproto-063-ble/`, 2020. Consultado: 2025-05-05.

[7] D. Cooper et al. G. Alagic, D. Apon. Status report on the third round of the nist post-quantum cryptography standardization process. NIST Interagency Report NIST IR 8413 (Updated), National Institute of Standards and Technology (NIST), July 2022.

[8] P. Ciadoux et al. G. Alagic, M. Bros. Status report on the fourth round of the nist post-quantum cryptography standardization process. NIST Internal Report NIST IR 8545, National Institute of Standards and Technology (NIST), Gaithersburg, MD, March 2025.

[9] Z. Li et al. Y. Zhang, Y. Bian. Continuous-variable quantum key distribution system: Past, present, and future. *Applied Physics Reviews*, 11(1):011318, 03 2024.

[10] M. Kumar and B. Mondal. A brief review on quantum key distribution protocols. *Multimedia Tools and Applications*, January 2025.

[11] B. Samson et al. A. Atadoga, O. Ajoke. A comparative review of data encryption of data methosds in the usa and europe. 5:447–460, Feb. 2024.

[12] Q. Zhang. An overview and analysis of hybrid encryption: The combination of symmetric encryption and asymmetric encryption. In *2021 2nd International Conference on Computing and Data Science (CDS)*, pages 616–622, 2021.

[13] Y. Yilmaz B. Halak and D. Shiu. Comparative analysis of energy costs of asymmetric vs symmetric encryption-based security applications. *IEEE Access*, 10:76707–76719, 2022.

[14] P. Kampanakis M. Anastasova and J. Massimo. PQ-HPKE: Post-quantum hybrid public key encryption. Cryptology ePrint Archive, Paper 2022/414, 2022.

[15] C.I. Ugwu et al.a C.H. Ugwuishiwu, U.E. Orji. An overview of quantum cryptography and shor?s algorithm. *International Journal of Advanced Trends in Computer Science and Engineering*, 9(5):7487–7495, September 2020.

[16] Z. Alam et al. D. Aasen, M. Aghaee. Roadmap to fault tolerant quantum computation using topological qubit arrays, 2025.

[17] S. Akleylek et al. F. Samiullah, M.L. Gan. Quantum resistance saber-based group key exchange protocol for iot. *IEEE Open Journal of the Communications Society*, 6:378–398, 2025.

[18] FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS 203, National Institute of Standards and Technology, Gaithersburg, MD, USA, August 2024.

[19] P.W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[20] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves, 2004.

[21] S. Singh and E. Sakk. Implementation and analysis of shor?s algorithm to break rsa cryptosystem security. January 2024.

[22] N. Gassner V. Weger and J. Rosenthal. A survey on code-based cryptography. *CoRR*, abs/2201.07119, 2022.

[23] L. Chen et al. G. Alagic, E. Barker. Recommendations for key-encapsulation mechanisms: Initial public draft. NIST Special Publication NIST SP 800-227 ipd, National Institute of Standards and Technology (NIST), January 2025.

[24] J. Daemen G. Bertoni and M. Peeters. The keccak reference. Technical report, NIST SHA-3 Competition, 2011. Round 3 submission.

[25] Post-quantum cryptography - round 3 submissions. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`, 2020. Consultado: 2025-05-05.

[26] Post-quantum cryptography - round 4 submissions. `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions`, 2022. Consultado: 2025-05-05.

[27] R. Mareta A. Satriawan and H. Lee. A complete beginner guide to the number theoretic transform (NTT). Cryptology ePrint Archive, Paper 2024/585, 2024.

[28] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[29] C. Peikert V. Lyubashevsky and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), November 2013.

[30] D. Micciancio and O. Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[31] Y. Wang and M. Wang. Module-LWE versus ring-LWE, revisited. Cryptology ePrint Archive, Paper 2019/930, 2019.

[32] L. Ducas and J. Schanck. Security estimation scripts for kyber and dilithium. `https://github.com/pq-crystals/security-estimates`, 2023. Consultado: 2025-05-31.

[33] S. Roy et al. J.P. Anvers, A. Karmakar. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. Cryptology ePrint Archive, Paper 2018/230, 2018.

[34] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 12 1962.

[35] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akademii Nauk SSSR*, 3, 01 1963.

[36] M. Bodrato and A. Zanoni. Integer and polynomial multiplication: towards optimal toom-cook matrices. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, page 17?24, New York, NY, USA, 2007. Association for Computing Machinery.

[37] D. Haken J.L. Bentley and J.B. James. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36?44, September 1980.

[38] L. Ducas et al. J. Bos, C. Costello. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. Cryptology ePrint Archive, Paper 2016/659, 2016.

[39] N. Aragon C. Aguilar Melchor and S. Bettaieb et al. HQC: Hamming Quasi-Cyclic (Fourth Round Submission). Technical report, HQC Team, October 2022. NIST PQC Round 4 submission.

[40] P. Gaborit. Shorter keys for code-based cryptography. 2005.

[41] G. David Forney. *Concatenated Codes*. MIT Press, Cambridge, MA, 1966.

[42] Shu Lin and Daniel Costello. *Error Control Coding*. 01 2004.

[43] A. Desai M. Bellare and D. Pointcheval. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 26–45, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[44] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554, Berlin, Heidelberg, 1999. Springer.

[45] Z. Gutterman L. Dorrendorf and B. Pinkas. Cryptanalysis of the random number generator of the windows operating system. Cryptology ePrint Archive, Paper 2007/419, 2007.