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, Febrero, 2026

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

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

Abreviaturas y siglas

CCA1 Ataque de texto cifrado no adaptable.

CCA2 Ataque de texto cifrado adaptable.

CPA Ataque de texto plano.

DFT Transformada Discreta de Fourier.

FIFO First In First Out.

FIPS Estandar Federal de Procesamiento de la Información.

HQC Hamming Quasi-Cyclic.

IND Indistinguibilidad del cifrado.

IND-CCA2 Indistinguibilidad bajo ataque de texto cifrado adaptable.

KAT Known Answer Test o Test de Respuesta Conocida.

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.

NM No maleabilidad del cifrado.

NTT Transformada Teórica de Números.

PA Consciencia de texto plano.

QC Cuasi-cíclico.

QROM Modelo de oráculo cuántico aleatorio.

R-LWE Aprendizaje Con Errores en Anillos.

ROM Modelo de oráculo aleatorio.

SHA Algoritmo Seguro de Hashing.

TFO Transformadas Fujisaki-Okamoto.

TRNG True Random Number Generator o Generador de Números Aleatorios Verdadero.

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 criptografía poscuántica en sistemas embebidos, con el objetivo de garantizar la seguridad de los protocolos de comunicación en un entorno industrial. Para ello, se realiza un estudio de la problemática de los algoritmos asimétricos clásicos y su vulnerabilidad frente al algoritmo de Shor, así como los fundamentos matemáticos de los nuevos esquemas criptográficos diseñados para resistir ataques cuánticos.

El cifrado asimétrico resulta esencial para establecer canales seguros mediante protocolos de intercambio de claves, permitiendo la posterior aplicación de cifrado simétrico. 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.

Como se ilustra en la Figura 1.1, el modelo propuesto plantea una arquitectura de control centralizado con alta capacidad de cómputo, donde los microprocesadores actúan como interfaces de entrada/salida encargadas de la comunicación con sensores y actuadores. Bajo este esquema, el cifrado se integra directamente en el bucle de control. No obstante, la viabilidad de este proceso depende estrictamente del establecimiento previo de claves seguras, siendo este último el objetivo de estudio en este trabajo.

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

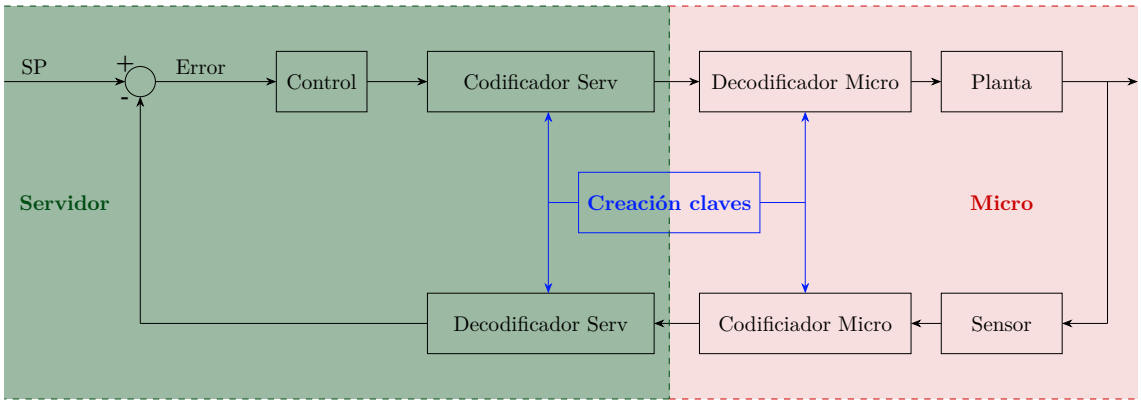

Figura 1.1: Representación del esquema de control en línea entre un micro y un servidor.

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.
- 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. C y C++

Se ha optado por el lenguaje de programación C debido a su predominio en el desarrollo para microcontroladores. Además, este lenguaje garantiza la compatibilidad directa con los algoritmos de cifrado postcuántico, cuyas implementaciones de referencia se encuentran desarrolladas en C.

1.3.3. Microprocesador CY8CPROTO-063-BLE

Para las pruebas experimentales referidas al entorno embebido, se ha seleccionado el kit de desarrollo [6]. Se ha preferido este hardware frente a las plataformas Arduino por sus superiores prestaciones, y frente a la familia STM32 debido a que la configuración y programación de esta última resulta significativamente más compleja.

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 y las pruebas a realizar.
- 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] [19]

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: [20] ECC: [21] RSA: [22]

2.4. Evolución de los algoritmos postcuánticos

a fundamentos y distintas alternativas a codebased [23]

2.5. Cifrado asimétrico postcuántico en sistemas embebidos

a

2.6. ALgoritmos postcuanticos

Kyber has a very efficient key-generation procedure (see also Section 2) and is therefore particularly well suited for applications that use frequent key generations to achieve forward secrecy. [1]

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 [24], 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.

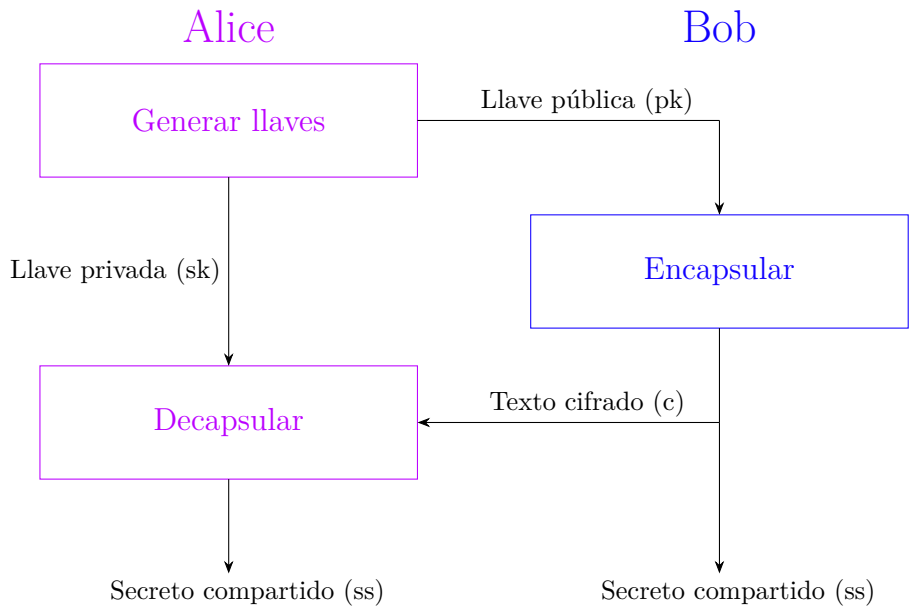

Figura 3.1: Representación del protocolo empleado en los algoritmos asimétricos para establecer una clave común o secreto compartido.

3.2. Notación básica

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:

$$c = a||b \tag{3.1}$$

Entonces:

$$b = c + l \tag{3.2}$$

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:

$$r' = r \bmod^{\pm} \alpha \quad \Longrightarrow \quad -\frac{\alpha}{2} < r' \leq \frac{\alpha}{2} \tag{3.3}$$

Mientras que la reducción modular estándar se denota como:

$$r' = r \bmod^{+} \alpha \quad \Longrightarrow \quad 0 \leq r' < \alpha \tag{3.4}$$

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. 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` [25], 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 $SHAKE-256$.

3.3.1. 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 $l = \log_2(b/25)$ para trabajar con los datos en formato matricial.

En esta sección se utiliza la siguiente notación:

- La expresión 0^j representa una cadena de j bits de ceros.
- La función $\texttt{len}(P)$ indica la longitud de la cadena P.
- La función $\texttt{trunc}_d(Z)$ devuelve los d primeros bits de Z.

3.3.1.1. 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.

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

$$A(x, y, z) = S(w \cdot (5y + x) + z) \tag{3.5}$$

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.

$$\begin{array}{lllll} \text{Linea}(i,j) = & = A(i,j,0) & ||A(i,j,1) & ||\ldots & ||A(i,j,w-1) \\ \text{Plano}(j) & = \text{Linea}(0,j) & ||\text{Linea}(1,j) & ||\ldots & ||\text{Linea}(4,j) \\ S & = \text{Plano}(0) & ||\text{Plano}(1) & ||\ldots & ||\text{Plano}(4) \end{array} \tag{3.6}$$

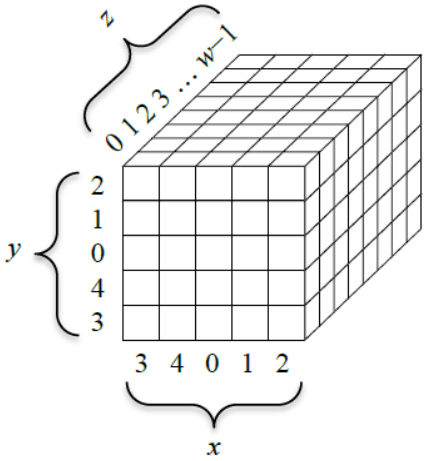

Figura 3.2: Dibujo de un vector de estado [4].

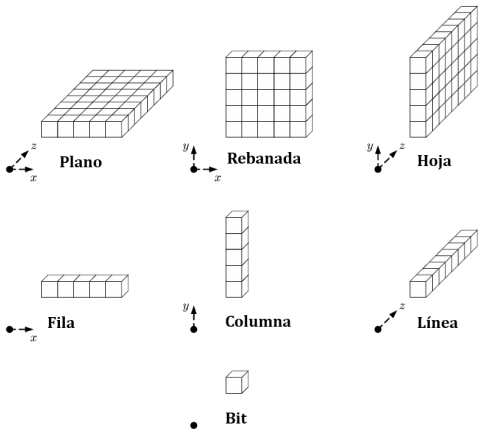

Figura 3.3: Elementos de un vector de estado [4].

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. La descripción detallada de las transformadas se encuentra en el anexo A.1.

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

$$\texttt{Rnd}(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r) \tag{3.7}$$

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 1 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'

3.3.1.2. 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 3.4, 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

$$c = b - r \tag{3.8}$$

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.

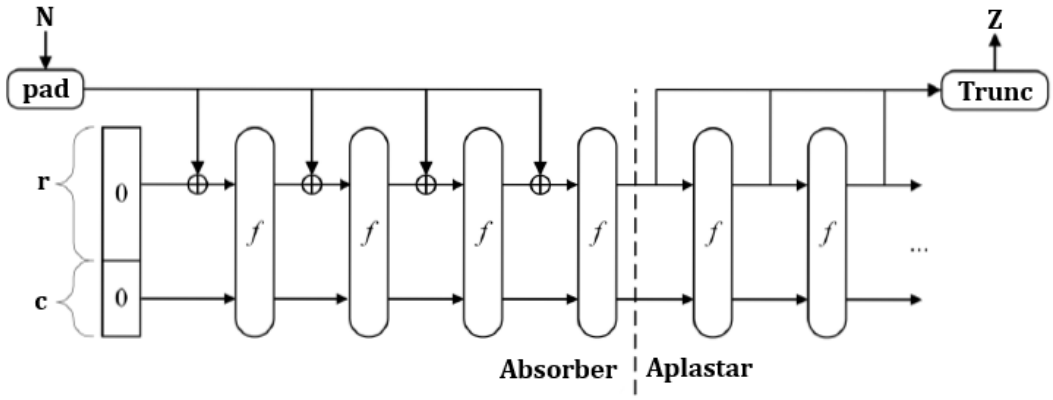

Figura 3.4: Representación visual del funcionamiento del algoritmo `sponge` [4].

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

Algoritmo 2 sponge

Entrada: N, d

Salida: Z

1: Calcular:
$$P = N||\texttt{pad}(r, \texttt{len}(N)) \tag{3.9}$$
2: Calcular:
$$n = \frac{len(P)}{r} \tag{3.10}$$
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:
$$S = f(S \oplus (P_i||0^c)) \tag{3.11}$$
7: **end for**
8: Inicializar Z como una cadena vacía
9: Asignar
$$Z = Z||\texttt{Trunc}_r(S) \tag{3.12}$$
10: **if** $d \leq len(Z)$ **then**
11: **return** $\texttt{Trunc}_d(S)$
12: **end if**
13: Asignar:
$$S = f(S) \tag{3.13}$$
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:

$$m + \texttt{len}(P) = \lambda x \tag{3.14}$$

El algoritmo de `pad10*1`:

Algoritmo 3 pad10*1

Entrada: x, m

Salida: P

1: Asignar:

$$j = -m - 2 \bmod x \tag{3.15}$$

2: Calcular

$$P = 1||0^j||1 \tag{3.16}$$

3: **return** P

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

$$\texttt{Keccak}[c](N, d) = \texttt{sponge}[\texttt{Keccak-p}(1600, 24), \texttt{pad10*1}, 1600 - c](N, d) \tag{3.17}$$

Por tanto, las funciones de hash se definen como:

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

3.3.1.3. 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 $\texttt{hash}(x) = h$. 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 $\texttt{hash}(x_1) = \texttt{hash}(x_2)$. Esta propiedad previene la falsificación de mensajes.

3.4. 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 [26] y cuarta [27] ronda del proceso de estandarización, resulta apropiado presentar su funcionamiento aquí en lugar de en la sección de desarrollo.

3.4.1. Fundamentos matemáticos de CRYSTALS-Kyber

Se elabora esta sección siguiendo el artículo [1]. CRYSTALS-Kyber es un algoritmo postcuántico basado en anillos algebraicos sobre polinomios, denotados como R_q:

$$R_q \coloneqq \frac{\mathbb{Z}_q[X]}{X^n + 1} \tag{3.18}$$

Según la especificación [1], la implementación de referencia utiliza $q = 3329$ y $n = 256$. Esta elección de parámetros es crítica, pues habilita el uso de la Transformada Teórica de Números (NTT) para la multiplicación polinómica. El uso de la NTT reduce la complejidad temporal de la operación desde $\mathcal{O}(n^2)$, correspondiente al método clásico de multiplicación, a $\mathcal{O}(n \log n)$. Para un análisis detallado del véase [28].

Para la correcta definición de la NTT, es requisito indispensable que $n|(q-1)$, es decir, que n divida a $(q-1)$. Esta condición asegura la existencia de raíces enésimas de la unidad en el cuerpo finito $\mathbb{Z}_q$ (dado que q es primo), consecuencia directa del teorema 2.8 de [29]:

Proposición 1 Para n, $q > 1$, el cuerpo $\mathbb{Z}_q$ tiene una raíz enésima de la unidad si y solo sí $n|(q-1)$

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 en lugar de los 256 factores lineales.

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

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:
$$\hat{a}_j = \sum_{i=0}^{n-1} \phi^{i(2j+1)} a_j \bmod q \tag{3.19}$$
2. La transformada inversa:
$$a_i = n^{-1} \sum_{j=0}^{n-1} \phi^{-i(2j+1)} \hat{a}_j \bmod q \tag{3.20}$$

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 [28] 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.

$$\hat{g} = \begin{bmatrix} \phi^0 & \phi^1 & \phi^2 & \phi^3 \\ \phi^0 & \phi^3 & \phi^6 & \phi^1 \\ \phi^0 & \phi^5 & \phi^2 & \phi^7 \\ \phi^0 & \phi^7 & \phi^6 & \phi^5 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 & 1925 & 3383 & 6468 \\ 1 & 6468 & 4298 & 1925 \\ 1 & 5756 & 3383 & 1213 \\ 1 & 1213 & 4298 & 5756 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix} \tag{3.21}$$

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 $n^{-1} = 5761$:

$$g = n^{-1} \begin{bmatrix} \phi^0 & \phi^0 & \phi^0 & \phi^0 \\ \phi^{-1} & \phi^{-3} & \phi^{-5} & \phi^{-7} \\ \phi^{-2} & \phi^{-6} & \phi^{-2} & \phi^{-6} \\ \phi^{-3} & \phi^{-1} & \phi^{-7} & \phi^{-5} \end{bmatrix} \cdot \hat{g} = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1213 & 5756 & 6468 & 1925 \\ 4298 & 3383 & 4298 & 3383 \\ 5756 & 1213 & 1925 & 6468 \end{bmatrix} \cdot \begin{bmatrix} 2489 \\ 7489 \\ 6478 \\ 6607 \end{bmatrix} \tag{3.22}$$

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:

$$g \cdot h = \mathrm{NTT}^{-1}(\mathrm{NTT}(g) \circ \mathrm{NTT}(h)) \tag{3.23}$$

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:
$$\phi^{k+2n} = \phi^k \tag{3.24}$$
2. Simetría:
$$\phi^{k+n} = \phi^{-k} \tag{3.25}$$

Con estas propiedades, se puede implementar el algoritmo de Cooley-Tukey [30] 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:

$$\hat{a}_j = \sum_{i=0}^{n-1} \phi^{i(2j+1)} a_j \bmod q = \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i} + \phi^{2j+1} \sum_{i=0}^{n/2-1} \phi^{4ij+2i} a_{2i+1} \bmod q \tag{3.26}$$

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:

$$\hat{a}_j = A_j + \phi^{4ij+2i} B_j \bmod q \tag{3.27}$$

$$\hat{a}_{j+n/2} = A_j - \phi^{4ij+2i} B_j \bmod q \tag{3.28}$$

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:

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \xi^{2i+1}\right) \tag{3.29}$$

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

$$NTT(f) = \hat{f} = \left(\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, ..., \hat{f}_{254} + \hat{f}_{255} X\right) \tag{3.30}$$

Con

$$\begin{aligned}\hat{f}_{2i} &= \sum_{j=0}^{127} f_{2j}\xi^{(2i+1)j} \\ \hat{f}_{2i+1} &= \sum_{j=0}^{127} f_{2j+1}\xi^{(2i+1)j}\end{aligned} \tag{3.31}$$

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:

$$h = f \cdot g = \text{NTT}^{-1}\left[\text{NTT}(f) \circ \text{NTT}(g)\right] \tag{3.32}$$

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

$$\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1})\,(\hat{g}_{2i} + \hat{g}_{2i+1}) \mod \left(X^2 - \xi^{2i+1}\right) \tag{3.33}$$

En la figura 3.5 se puede ver una representación gráfica de como funciona este proceso.

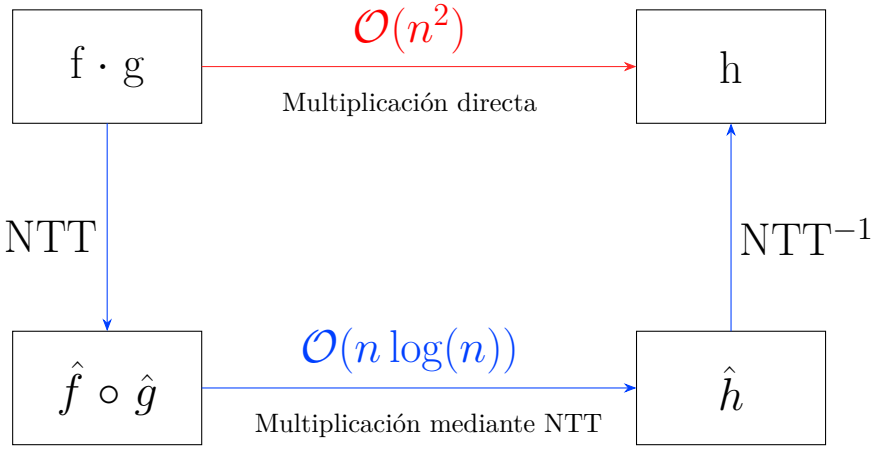

Figura 3.5: Representación del cálculo de un producto de polinomios mediante NTT en comparación con los algoritmos clásicos.

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

Para la elaboración de esta sección, se ha utilizado la información contenida en el artículo [31], 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 $\mathbb{B} = \{b_1, ..., b_n\}$:

$$\mathcal{A} = \mathcal{L}(\mathbb{B}) = \left\{\sum_{i\in n} z_i b_i : z \in Z^n\right\} \tag{3.34}$$

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.

3.4.1.3. 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 $\mathcal{O}(n^2)$ [32]. 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 [33].

En cuanto a la eficiencia en la reducción del tamaño de las claves, esta puede observarse claramente a través del ejemplo de la tabla 3.1.

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 3.1: 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 A.2.

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 4 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

$$b := a \cdot sk + e \tag{3.35}$$

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 5 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:

$$u := a \cdot r + e_1 \bmod^{+} \mathrm{q} \tag{3.36}$$

3: Calcular el segundo texto cifrado v:

$$v := b \cdot r + e_2 + \left\lfloor \frac{q}{2} \right\rceil \cdot z \bmod^{+} \mathrm{q} \tag{3.37}$$

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} [34].

[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 A.2.

Algoritmo 6 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.

$$z' := v - u \cdot s = (r \cdot e - s \cdot e_1 + e_2) + \left\lfloor \frac{q}{2} \right\rceil \cdot z \bmod^{+} \mathrm{q} \tag{3.38}$$

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

1.1: Distancia a 0:

$$\mathrm{d}_i(0) := \left\| z'_i \bmod^{\pm} \left\lfloor \frac{q}{2} \right\rceil \right\| \tag{3.39}$$

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

$$\mathrm{d}_i\left(\frac{q}{2}\right) := \left\| \left\lfloor \frac{q}{2} \right\rceil - \mathrm{d}_i(0) \right\| \tag{3.40}$$

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 B 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 [33].

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 A.2.

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

3.4.1.4. Parámetros empleados en Kyber

A continuación, se presenta la tabla 3.2 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 3.2: 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.

3.4.2. Fundamentos matemáticos de Saber

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

$$R_q = \frac{\mathbb{Z}_q}{X^n + 1} \tag{3.41}$$

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 [35] 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.

3.4.2.1. 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 [36] y Toom-Cook-4 [37] para acelerar las multiplicaciones polinómicas. Para ello se sigue la estructura del diagrama de la figura 3.6.

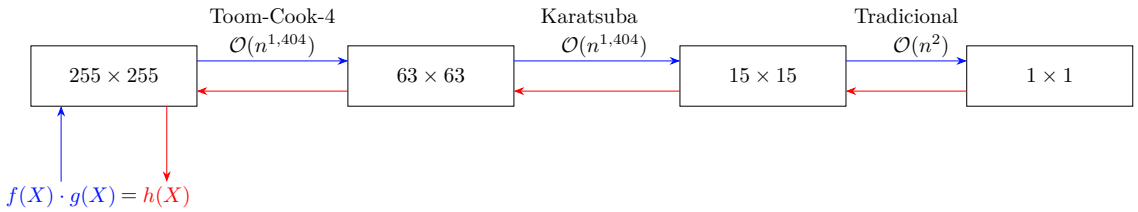

Figura 3.6: Representación del cálculo de un producto de polinomios en Saber. Inicialmente, para polinomios de grado 255, se aplica el algoritmo Toom-Cook-4 que reduce los productos a polinomios de grado 63, ya que ofrece el mejor rendimiento asintótico. Sin embargo, conforme disminuye el tamaño de los 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 [38] 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:

$$\begin{aligned} h(T) &= h(X,T) = h_0(X) + h_1(X) \cdot T + h_2(X) \cdot T^2 + h_3(X) \cdot T^3 \\ T &= X^{n/4} \end{aligned} \tag{3.42}$$

  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:

$$\begin{aligned} f(T) &= f_0(X) + f_1(X) \cdot T + f_2(X) \cdot T^2 + f_3(X) \cdot T^3 \\ g(T) &= g_0(X) + g_1(X) \cdot T + g_2(X) \cdot T^2 + g_3(X) \cdot T^3 \end{aligned} \tag{3.43}$$

  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 [38] se desarrolla un algoritmo eficiente para luego recuperar adecuadamente los coeficientes en la interpolación y para ello, se deben utilizar los siguientes puntos:

$$v_i = \left\{\infty, 2, 1, -1, \frac{1}{2}, -\frac{1}{2}, 0\right\} \tag{3.44}$$

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

$$c(T) = c_0 + c_1 \cdot T + c_2 \cdot T^2 + c_3 \cdot T^3 + c_4 \cdot T^4 + c_5 \cdot T^5 + c_6 \cdot T^6 \tag{3.45}$$

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

$$c_i(X) = c_{i,0} + c_{i,1} \cdot X + c_{i,2} \cdot X^2 + ... + c_{i,126} \cdot X^{126} \tag{3.46}$$

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

$$\begin{pmatrix} c_6(X) \\ c_5(X) \\ c_4(X) \\ c_3(X) \\ c_2(X) \\ c_1(X) \\ c_0(X) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1/64 & 1/32 & 1/16 & 1/8 & 1/4 & 1/2 & 1 \\ 1/64 & -1/32 & 1/16 & -1/8 & 1/4 & -1/2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \omega_6(X) \\ \omega_5(X) \\ \omega_4(X) \\ \omega_3(X) \\ \omega_2(X) \\ \omega_1(X) \\ \omega_0(X) \end{pmatrix} \tag{3.47}$$

  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:

$$\begin{aligned} f &= f_0 + f_1 \cdot T + f_2 \cdot T^2 + f_3 \cdot T^3 \\ g &= g_0 + g_1 \cdot T + g_2 \cdot T^2 + g_3 \cdot T^3 \end{aligned} \tag{3.48}$$

Entonces su producto queda como:

$$f \cdot g = \begin{cases} T^6 \cdot (f_3 \cdot g_3) + \\ T^5 \cdot (f_3 \cdot g_2 + f_2 \cdot g_3) + \\ T^4 \cdot (f_3 \cdot g_1 + f_2 \cdot g_2 + f_1 \cdot g_3) + \\ T^3 \cdot (f_3 \cdot g_0 + f_2 \cdot g_1 + f_1 \cdot g_2 + f_0 \cdot g_3) + \\ T^2 \cdot (f_2 \cdot g_0 + f_1 \cdot g_1 + f_0 \cdot g_2) + \\ T^1 \cdot (f_1 \cdot g_0 + f_0 \cdot g_1) + \\ T^0 \cdot (f_0 \cdot g_0) \end{cases} \tag{3.49}$$

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

$$\begin{aligned} z_0 &= f_0 \cdot g_0 \\ z_1 &= f_1 \cdot g_1 \\ z_2 &= f_2 \cdot g_2 \\ z_3 &= f_3 \cdot g_3 \\ z_4 &= (f_0 + f_1)(g_0 + g_1) \\ z_5 &= (f_0 + f_2)(g_0 + g_2) \\ z_6 &= (f_1 + f_3)(g_1 + g_3) \\ z_7 &= (f_2 + f_3)(g_2 + g_3) \\ z_8 &= (f_0 + f_1 + f_2 + f_3)(g_0 + g_1 + g_2 + g_3) \end{aligned} \tag{3.50}$$

Con estos términos se puede ver como la ecuación 3.50 se puede escribir como una combinación lineal de los términos anteriores:

$$f \cdot g = C_0 + C_1 T + C_2 T^2 + C_3 T^3 + C_4 T^4 + C_5 T^5 + C_6 T^6 \tag{3.51}$$

Donde C_i es:

$$\begin{aligned} C_0 &= z_0 \\ C_1 &= z_4 - z_1 - z_0 \\ C_2 &= z_5 - z_2 - z_0 + z_1 \\ C_3 &= z_8 - (z_7 + z_6 + z_5 + z_4) + z_3 + z_2 + z_1 + z_0 \\ C_4 &= z_6 - z_3 - z_1 + z_2 \\ C_5 &= z_7 - z_3 - z_2 \\ C_6 &= z_3 \end{aligned} \tag{3.52}$$

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 [39].

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.

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

Los esquemas basados en la LWR [35] 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 reescalar 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-LWR se formula como la dificultad de distinguir entre parejas de muestras uniformes (a, u) y parejas (a, b) generadas de la siguiente manera:

$$\begin{aligned} &a \leftarrow \mathcal{U}(R_q^{l\times l}) \\ &s \leftarrow \beta_\mu(R_q^{l\times 1})) \\ &b \in R_p^{l\times 1} \\ &b = \left\lfloor \frac{p}{q}(A \cdot s) \right\rceil \end{aligned} \tag{3.53}$$

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 $\texttt{bits}(x, i, j)$:

$$\texttt{bits}(x, i, j) = [x(i-j)]\&(2^j - 1) = \begin{cases} \dfrac{x}{2^{i-j}} \bmod^+ 2^j \text{ si } j \neq i \\ x \bmod^+ 2^j \text{ si } j = i \end{cases} \tag{3.54}$$

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

Algoritmo 7 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 $b \in R_p^{l\times 1}$:

$$b := \texttt{bits}(A \cdot s + h, \varepsilon_q, \varepsilon_p) \tag{3.55}$$

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 8 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:

$$\begin{aligned} b' &:= \texttt{bits}(A^T \cdot s' + h, \varepsilon_q, \varepsilon_p) \\ v' &:= b \cdot \texttt{bits}(s', \varepsilon_p, \varepsilon_p) + h_1 \end{aligned} \tag{3.56}$$

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

$$c := \texttt{bits}(v', \varepsilon_p - 1, \varepsilon_t) \tag{3.57}$$

4: Calcular el secreto compartido ss

$$ss' := \texttt{bits}(v', \varepsilon_p, 1) \tag{3.58}$$

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 9 Descifrado Mod-LWR

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

Salida: ss

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

$$v := b'^T \cdot \texttt{bits}(s, \varepsilon_p, \varepsilon_p) + h_1 \tag{3.59}$$

2: Calcular el valor del secreto compartido ss

$$ss := \texttt{bits}(v - 2^{\varepsilon_p - \varepsilon_t - 1} \cdot c + h_2), \varepsilon_p, 1 \tag{3.60}$$

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 3.3, 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 [40], la probabilidad de coincidencia del secreto depende de la distancia entre las variables v y v'. En particular, se cumple que

$$d = \|v - v'\| \quad \Rightarrow \quad \begin{cases} d > \dfrac{p}{4}\left(1 + \dfrac{1}{t}\right) = c_1, & \text{el secreto difiere.} \\ d < \dfrac{p}{4}\left(1 - \dfrac{1}{t}\right) = c_2, & \text{el secreto coincide.} \\ c_2 \leq d \leq c_1, & \text{el secreto coincide con probabilidad intermedia.} \end{cases} \tag{3.61}$$

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:

$$1 - \delta = \Pr\left[\|\Delta v + e_r\| < \frac{p}{4}\right] \tag{3.62}$$

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 1. 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:

$$\begin{aligned}\texttt{bits}(A\cdot s+h,\varepsilon_q,\varepsilon_p)&=\frac{p}{q}\cdot A\cdot s+e\\ \texttt{bits}(A^T\cdot s'+h,\varepsilon_q,\varepsilon_p)&=\frac{p}{q}\cdot A^T\cdot s'+e'\end{aligned}\tag{3.63}$$

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

$$\delta=\Pr\left[\left\|\left(s'^T\cdot e-e'^T\cdot s+e_r\right)\bmod^+\mathrm{p}\right\|_\infty>\frac{p}{4}\right]\tag{3.64}$$

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. En la demostración de este teorema se utilizan dos pasos, el primero [35] para demostrar que las condiciones propuestas son equivalentes a las de la ecuación 3.62 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 3.63:

$$\begin{aligned}b&=\frac{p}{q}\cdot A\cdot s+e\\ b'&=\frac{p}{q}\cdot A^T\cdot s'+e'\end{aligned}\tag{3.65}$$

A continuación se despejan v y v':

$$\begin{aligned}v&=\frac{p}{q}s'^T\cdot A\cdot s+h_1+s'^T\cdot e\bmod^+\mathrm{p}\\ v'&=\frac{p}{q}s'^T\cdot A\cdot s+h_1+e'^T\cdot s\bmod^+\mathrm{p}\end{aligned}\tag{3.66}$$

Simplemente restando se obtiene Δv y se claramente se ve la equivalencia a la ecuación 3.64.

Paso 2: Igualdad de secretos En este paso se hace la demostración empleando los parámetros de FireSaber de la tabla 3.3. Sea $Q_8\in R_{128}$:

$$Q_8=\left\lfloor\frac{v'}{8}\right\rfloor\tag{3.67}$$

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

$$v'=8\left\lfloor\frac{v'}{8}\right\rfloor+r\tag{3.68}$$

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:

$$c=\left\lfloor\frac{v'}{8}\right\rfloor\bmod^+64=64\cdot\omega+\left\lfloor\frac{v'}{8}\right\rfloor\bmod^+64\tag{3.69}$$

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

$$ss'=\left\lfloor\frac{8\cdot(64\cdot\omega+c)+r}{512}\right\rfloor\bmod^+2=\left\lfloor\omega+\frac{8\cdot c+r}{512}\right\rfloor\bmod^+2\tag{3.70}$$

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

$$ss' = \left\lfloor \omega + \frac{511}{512} \right\rceil \bmod^+ 2 = \omega \bmod^+ 2 \tag{3.71}$$

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

$$ss = \left\lfloor \frac{v - 8 \cdot c + h_2}{512} \right\rceil \bmod^+ 2 \tag{3.72}$$

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:

$$ss = \left\lfloor \frac{[8 \cdot (64 \cdot \omega + c) + r + \Delta] - 8 \cdot c + h_2}{512} \right\rceil \bmod^+ 2 \tag{3.73}$$

Despejando:

$$ss = \left\lfloor \omega + \frac{r + \Delta + h_2}{512} \right\rceil \bmod^+ 2 \tag{3.74}$$

Donde justamente si se hace el cambio de notación $e_r := r$ se obtiene la condición de la ecuación 3.64, el método falla si:

$$e_r + \Delta > (512 - h_2) = 260 > \frac{p}{4} = 256 \tag{3.75}$$

□

3.4.2.3. Parámetros empleados en Saber

A continuación, se presenta la tabla 3.3 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 3.3: 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.

3.4.3. 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) [41] 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 [23]. 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.

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

Para elaborar esta sección se ha utilizado el artículo sobre HQC, así como la entrega al NIST [41]. 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 $w = u \cdot v$

$$w_k = \sum_{i+j=k \bmod^+ n} u_i \cdot v_j \ \forall\ k \in 0, \ldots, n-1 \tag{3.76}$$

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

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

$$\texttt{rot}(v) = \begin{pmatrix} v_0 & v_{n-1} & \ldots & v_1 \\ v_1 & v_0 & \ldots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} & v_{n-2} & \ldots & v_0 \end{pmatrix} \tag{3.77}$$

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

$$u \cdot v = u \cdot \texttt{rot}(v)^T = (\texttt{rot}(u) \cdot v^T)^T = v \cdot u \tag{3.78}$$

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 $G \in \mathbb{F}_2^{k \times n}$:

$$C = \{m \cdot G,\ m \in \mathbb{F}_2^k\} \tag{3.79}$$

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:

$$\begin{aligned} C &= \{v \in \mathbb{F}_2^n \mid H \cdot v^T = 0\} \\ C^T &= \{u \cdot H,\ u \in \mathbb{F}_2^k\} \end{aligned} \tag{3.80}$$

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

$$H \cdot v^T \rightarrow \text{ si } v \in C \rightarrow H \cdot v^T = 0 \tag{3.81}$$

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

$$d = \min_{\substack{u,v \in C \\ u \neq v}} \omega(u - v) \tag{3.82}$$

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:

$$\delta = \left\lfloor \frac{d-1}{2} \right\rfloor. \tag{3.83}$$

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 [42].

Un código cuasi-cíclico se puede representar como un vector

$$c = (c_0, \ldots, c_{s-1}) \in \mathbb{F}_2^{sn}, \quad c_i \in \mathcal{R}. \tag{3.84}$$

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:

$$c \cdot X = (c_0 \cdot X, \ldots, c_{s-1} \cdot X) \in C. \tag{3.85}$$

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

$$H = \begin{pmatrix} I_n & 0 & \ldots & 0 & A_0 \\ 0 & I_n & \ldots & 0 & A_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & I_n & A_{s-2} \end{pmatrix} \in \mathcal{R}^{(s-1)n \times sn} \tag{3.86}$$

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 [41].

3.4.3.2. 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:

$$\mathbb{F}_q^{n_e} \cong \mathbb{F}_2^N \tag{3.87}$$

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 [43]. Esta estructura se puede ver en la figura 3.7.

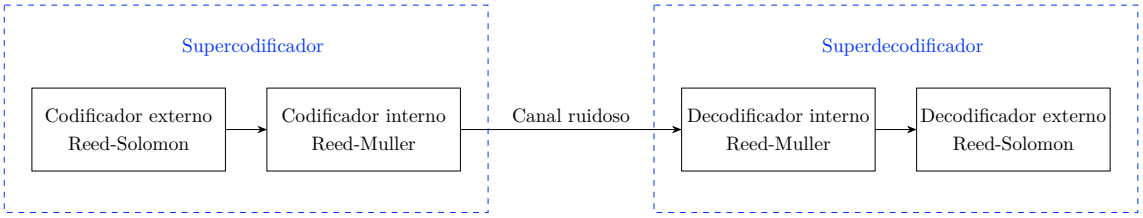

Figura 3.7: Representación del funcionamiento mediante códigos concatenados en HQC.

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:

$$C[128, 8, 64] \rightarrow C[640, 8, 320] \tag{3.88}$$

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:

$$g(x) = (x + \alpha)\left(x + \alpha^2\right) \cdots \left(x + \alpha^{2\delta}\right) \tag{3.89}$$

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:

$$1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8 \tag{3.90}$$

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

- **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:
  $$y(x) = u(x) \cdot x^{n-k} \tag{3.91}$$
  - Calcular el resto $b(x)$ de $y(x)$ dividido por el polinomio generador $g(x)$.

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

$$c(x) = b(x) + y(x) \tag{3.92}$$

- **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 $v(x)$:

$$r(x) = v(x) + e(x) \tag{3.93}$$

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:

$$r(\alpha^i) = e(\alpha^i) + v(\alpha^i) = e(\alpha^i),\ v(\alpha^i) = 0 \text{ porque pertenece al código} \tag{3.94}$$

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:

$$\begin{cases} S_1 = e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + \cdots + e_{j_t}\alpha^{j_t} \\ S_2 = e_{j_1}(\alpha^{j_1})^2 + e_{j_2}(\alpha^{j_2})^2 + \cdots + e_{j_t}(\alpha^{j_t})^2 \\ S_3 = e_{j_1}(\alpha^{j_1})^3 + e_{j_2}(\alpha^{j_2})^3 + \cdots + e_{j_t}(\alpha^{j_t})^3 \\ \vdots \\ S_{2\delta} = e_{j_1}(\alpha^{j_1})^{2\delta} + e_{j_2}(\alpha^{j_2})^{2\delta} + \cdots + e_{j_t}(\alpha^{j_t})^{2\delta} \end{cases} \tag{3.95}$$

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

$$\begin{cases} S_1 = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \cdots + e_{j_t}\beta_t \\ S_2 = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \cdots + e_{j_t}\beta_t^2 \\ S_3 = e_{j_1}\beta_1^3 + e_{j_2}\beta_2^3 + \cdots + e_{j_t}\beta_t^3 \\ \vdots \\ S_{2\delta} = e_{j_1}\beta_1^{2\delta} + e_{j_2}\beta_2^{2\delta} + \cdots + e_{j_t}\beta_t^{2\delta} \end{cases} \tag{3.96}$$

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

$$\sigma(x) = (1 + \beta_1 x)(1 + \beta_2 x) \cdots (1 + \beta_t x) = 1 + \sigma_1 x + \sigma_2 x^2 + \cdots + \sigma_t x^t \tag{3.97}$$

este polinomio se calcula utilizando el algoritmo de Berlekamp [44]:

$$\begin{pmatrix} S_1 & S_2 & \cdots & S_\delta \\ S_2 & S_3 & \cdots & S_{\delta+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_\delta & S_{\delta+1} & \cdots & S_{2\delta-1} \end{pmatrix} \begin{pmatrix} \sigma_\delta \\ \sigma_{\delta-1} \\ \vdots \\ \sigma_1 \end{pmatrix} = \begin{pmatrix} -S_{\delta+1} \\ -S_{\delta+2} \\ \vdots \\ -S_{\delta+\delta} \end{pmatrix}. \tag{3.98}$$

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:

$$\begin{cases} \sigma_0 = S_0 = 1 \\ Z(x) = \sum_{i=0}^{t} \sum_{j=0}^{i} S_j \cdot \sigma_{i-j} x^i \\ \\ e_{jl} = \dfrac{Z(\beta_l^{-1})}{\prod_{\substack{i=1 \\ i \neq l}}^{t} \left(1 + \beta_i \beta_l^{-1}\right)} \end{cases} \tag{3.99}$$

Por último, se descifra el mensaje v:

$$v(x) = r(x) - e(x) \tag{3.100}$$

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:
    $$c = m \cdot G \tag{3.101}$$
  - 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.
    $$(-1)^{x_1} + (-1)^{x_2} + (-1)^{x_3} + (-1)^{x_4} + (-1)^{x_5} \tag{3.102}$$
  - Realizar la transformada de Hadamard sobre el vector F creado, donde se define la matriz de Hadamart H_n como:
    $$H_{2^k} = \begin{pmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{pmatrix} \text{ con } H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.103}$$
    Al multiplicar esta matriz por F, $\hat{F}$ se obtiene el peso de cada fila:
    $$\hat{F} = H_n \cdot F \tag{3.104}$$
- 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):

$$\begin{cases} +1 \to 0 \\ -1 \to 1 \end{cases} \tag{3.105}$$

En el anexo D 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.

3.4.3.3. Parámetros empleados en HQC

A continuación, se presenta la tabla 3.4 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 3.4: 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.

3.5. 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.

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

Para redactar esta sección se usa el artículo [5] 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, no constituyen ataques físicos al sistema, sino análisis de seguridad formal:

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 3.8.
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 3.8.
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 3.8.

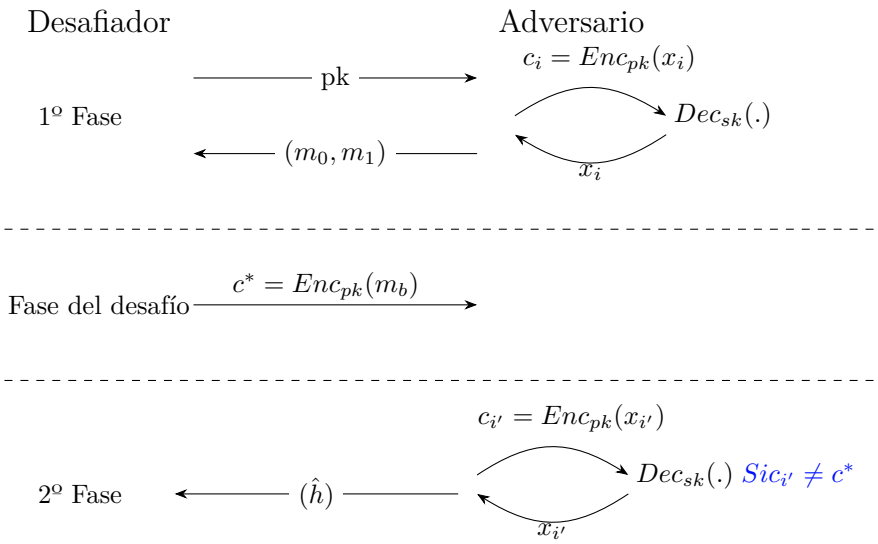

Figura 3.8: Representación de las fases posibles de ataque mediante los oráculos de cifrado y descifrado.

Estos objetivos y tipos de ataques se agrupan en conjunto creando un total de 6 posibles esquemas como se ve en la figura 3.9.

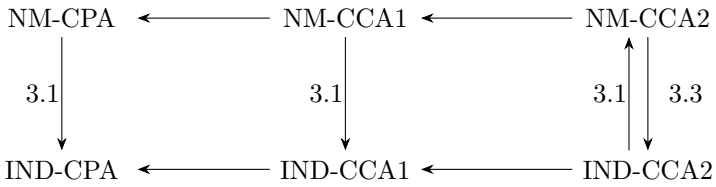

Figura 3.9: Representación de las equivalencias de seguridad entre los distintos esquemas de seguridad posible. Si existe una flecha entre dos esquemas A y B implica que si se da A entonces B también se cumple. Los números denotan los teoremas del artículo [5] que demuestran estas relaciones.

Para definir de manera formal los ataques IND se utiliza el siguiente experimento. Sea A_1 un algoritmo que, a partir de la llave pública pk, obtiene como salida (x_0, x_1, s), donde x_0 y x_1 son mensajes de la misma longitud y s es información del estado que se quiere conservar. Uno de estos mensajes se elige al azar denotado como x_b. Entonces, el reto y se obtiene al cifrar x_b con la llave pública. Finalmente, mediante el algoritmo A_2 se debe determinar cuál es el texto de entrada, valor de b, dado el estado s y el reto y.

Por tanto se definen los esquemas IND-{CPA, CCA1, CCA2}. Sea un esquema de cifrado asimétrico $\mathcal{PE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ con $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ la función generadora de claves, el cifrado y el descifrado respectivamente. Sea un adversario $A = (A_1, A_2)$ para los ataques $atq \in \{cpa, cca1, cca2\}$ y $k \in \mathbb{Z}$:

$$\mathrm{Adv}^{\text{ind-atk}}_{\mathcal{PE},A}(k) = \mathrm{P}\left(\mathtt{Exp}^{\text{ind-atk-1}}_{\mathcal{PE},A}(k) = 1\right) - \mathrm{P}\left(\mathtt{Exp}^{\text{ind-atk-0}}_{\mathcal{PE},A}(k) = 1\right) \tag{3.106}$$

donde Adv denota la ventaja de un adversario de lograr el ataque, P denota la probabilidad de un

suceso y Exp denota el experimento:

$$
\begin{array}{ll}
b \in \{0,1\} & \\
\mathtt{Exp}^{\text{ind-atk-1}}_{\mathcal{PE},A}(k) = & \left| \begin{array}{l} (pk, sk) := \mathcal{K}(k) \\ (x_0, x_1, s) := A_1^{\mathcal{O}_1(.)}(pk) \\ y := \mathcal{E}_{pk}(x_b) \\ d := A_2^{\mathcal{O}_2(.)}(x_0, x_1, s, y) \\ \text{Return } d \end{array} \right. \\
\text{con:} & \left| \begin{array}{lll} \text{atk=cpa} & \rightarrow \mathcal{O}_1(.) = \varepsilon & \mathcal{O}_2(.) = \varepsilon \\ \text{atk=cca1} & \rightarrow \mathcal{O}_1(.) = \mathcal{D}_{sk}(.) & \mathcal{O}_2(.) = \varepsilon \\ \text{atk=cca2} & \rightarrow \mathcal{O}_1(.) = \mathcal{D}_{sk}(.) & \mathcal{O}_2(.) = \mathcal{D}_{sk}(.) \end{array} \right.
\end{array}
\tag{3.107}
$$

donde ε denota que o esta disponible la función. Por tanto, viendo esta definición formal se puede ver que el esquema es seguro ande un ataque IND si para A en tiempo polinómico la ventaja obtenida es despreciable.

Un concepto relevante dentro de la seguridad es el concepto de "plaintext awarness" PA o consciencia de texto plano que formaliza la incapacidad de un adversario para crear un texto cifrado y sin conocer el texto plano x que lo genera. Normalmente este concepto solo se define en el modelo de oráculo aleatorio donde todas las partes tienen acceso a una función aleatoria H que conviertes del conjunto de todas las funciones a un dominio del rango adecuado. En la práctica H se implementa mediante una función de hashing. En la definición formal anterior implica modificar las funciones $\mathcal{E}_{pk} \rightarrow \mathcal{E}^H_{pk}$ y $\mathcal{D}_{sk} \rightarrow \mathcal{D}^H_{sk}$.

Estos cambios implican que un adversario B para tener consciencia de texto plano se dan una llave pública pk y acceso al oráculo aleatorio H. El adversario computa un texto cifrado y y, para ser consciente necesita conocer el valor del descifrado x de dicho texto cifrado. Esto se formaliza mediante un algoritmo K denominado extractor de textos planos que obtendría x solo al ver la llave pública pk, las consultas de B al oráculo H y sus respuestas $\mathcal{E}^H_{pk}$.

La función $\mathtt{run}\ B^{H,\mathcal{E}^H_{pk}}(pk)$ es una función auxiliar devuelve hH, C e y:

1. hH representa una lista de parejas (h_i, H_i) obtenidas mediante el oráculo aleatorio, es decir, una caja negra que genera números aleatorios.

2. C representa una lista de de todos los textos cifrados obtenidos como resultado de las $\mathcal{E}^H_{pk}$ peticiones.

3. y representa el texto cifrado final obtenido por el adversario a partir de toda esta información.

Por tanto, se define la probabilidad de que el adversario B consiga extraer mediante un extractor K correctamente el texto plano x a partir del texto cifrado y como:

$$
\mathrm{Succ}^{\text{pa}}_{\mathcal{PE},B,K}(k) = \mathrm{P}\left[\begin{array}{l} H \leftarrow \text{Hash} : \\ (pk, sk) = \mathcal{K}(k) : \\ (hH, C, y) = \mathtt{run}\ B^{H,\mathcal{E}^H_{pk}}(pk) : \\ \qquad\qquad \rightarrow \quad K(hH, C, y, pk) == \mathcal{D}^H_{sk} \end{array} \right]
\tag{3.108}
$$

Con esta definición y fijando que $y \notin C$ se dice que K es un $\lambda(k)$ extractor si K con tiempo de ejecución polinomial y para todo adversario B con $\mathrm{Succ}^{\text{pa}}_{\mathcal{PE},B,K}(k) \geq \lambda(k)$. Se dice que el algoritmo de cifrado asimétrico es seguro en PA si $1 - \lambda(k)$ es despreciable.

Por último es relevante destacar que se cumplen las siguientes afirmaciones demostradas en el articulo [5]:

1. Si un esquema es seguro en sentido PA también es seguro en el modelo de oráculo aleatorio en el sentido IND-CCA2.

2. Si un esquema es seguro en el modelo de oráculo aleatorio en el sentido IND-CCA2 no implica que sea seguro en sentido PA.

Este modelo presentado no es suficiente, pues los algoritmos Kyber, Saber y HQC no se enfrentan al oráculo aleatorio tradicional sino que su rival es un atacante cuántico. En el artículo [45] se desarrolla en detalle la equivalencia entre ambos aunque a continuación se presentan algunas nociones básicas.

Antes de pasar a comentar las diferencias con respecto a un atacante cuántico es necesario dar unas nociones básicas sobre el funcionamiento de este tipo de atacante [46]:

- El atacante trabaja mediante Qubits. Un Qubit se define como la combinación lineal de varios estados posibles, donde el Qubit más sencillo $|\varphi\rangle$ esta formado por dos estados o base $|0\rangle$ y $|1\rangle$:

$$\begin{aligned} |\varphi\rangle &= \alpha|0\rangle + \beta|1\rangle, \text{ con } \alpha, \beta \in \mathbb{C} \\ |0\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ |1\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned} \tag{3.109}$$

- Se habla de un estado en superposición porque los coeficientes α y β determinan las probabilidades (α^2 y β^2, respectivamente) de que el estado φ colapse en cada uno de sus estados posibles. Desde un enfoque estocástico, puede afirmarse que φ permanece en superposición hasta que se mide la variable, momento en el cual colapsa en uno de los dos estados. Además, dichas probabilidades deben satisfacer la condición:

$$\alpha^2 + \beta^2 = 1 \tag{3.110}$$

  Aunque este ejemplo se ha planteado con dos vectores como base, el concepto es completamente generalizable a bases de dimensión n.

- La principal ventaja de estos sistemas es que permiten trabajar en un espacio de dimensión n, lo que posibilita representar un número infinito de estados dentro de la esfera correspondiente.

  Un elemento fundamental en este marco es la **puerta cuántica CNOT**, definida como

$$\texttt{CNOT}(|A\rangle, |B\rangle) = |A, B \oplus A\rangle, \tag{3.111}$$

  donde $\oplus$ denota la suma módulo 2. Esta puerta puede entenderse como una generalización cuántica de la puerta clásica `XOR`. De manera análoga a cómo la puerta `NAND` permite construir cualquier operación lógica en la computación clásica, la puerta `CNOT` constituye un bloque básico universal para la computación cuántica.

Por tanto, la principal diferencia entre un oráculo clásico y un oráculo cuántico es la capacidad del segundo para poder evaluar la función en superposición, es decir, cuando se introduce un estado cuántico $|\varphi\rangle = \sum \alpha_x |x\rangle$ al oráculo O se obtiene el estado $|\varphi\rangle = \sum \alpha_x |O(x)\rangle$. Es decir, en cada iteración se evalúan exponencialmente más puntos.

Esto implica que las demostraciones clásicas no son suficientes para demostrar que un algoritmo cuánticamente seguro, pues demostraciones algunas demostraciones clásicas no se cumplen:

- Programación adaptativa: el modelo clásico permite simular las respuestas del oráculo para un adversario de manera adaptativa, sin embargo, en el modelo cuántico el adversario puede consultar el oráculo para un estado en superposición y, por tanto obtener una cantidad exponencial de información desde el inicio.
- Consciencia de preimagen: en algunas demostraciones es relevante conocer en que datos esta interesado el atacante, que en el caso cuántico puede estar escondido tras un estado en superposición.

- Simulación eficiente: en los algoritmos clásicos se pueden generar nuevas respuestas aleatorias bajo demanda para simular el crecimiento exponencial de un oráculo lo cual es imposible para un entorno cuántico porque todos los estados están presentes desde el inicio.

No obstante, muchas de las pruebas clásicas siguen siendo aplicables. En [45] y [47] se establecen las condiciones bajo las cuales una demostración en el modelo clásico puede extenderse y garantizar seguridad frente a adversarios cuánticos. Sin embargo, en otros casos es necesario recurrir a nuevas técnicas de reducción específicas para el paradigma de modelo de oráculo cuántico aleatorio, las cuales quedan fuera del alcance de este trabajo.

3.5.2. Transformadas Fujisaki-Okamoto TFO

Las transformadas Fujisaki-Okamoto (TFO) son una herramienta que permite convertir un algoritmo asimétrico débil en un algoritmo que cumple seguridad IND-CCA2 en el modelo de oráculo aleatorio [48].

Para lograr esta seguridad se crea un esquema híbrido entre el algoritmo asimétrico que se quiere blindar y un esquema de algoritmo simétrico implementado mediante una función de relleno.

Sea $\mathcal{PE}^{hy} = (\mathcal{K}^{hy}, \mathcal{E}^{hy}, \mathcal{D}^{hy})$ el esquema de intercambio de claves híbrido implementado a partir de los esquemas asimétrico $\mathcal{PE}^{asym}$ y simétrico $\mathcal{PE}^{sym}$. La función $\mathcal{K}^{hy}$ tiene el mismo funcionamiento que el algoritmo asimétrico, es decir, $\mathcal{K}^{hy} = \mathcal{K}^{asym}$. La función de cifrado $\mathcal{E}^{hy}$ se define como:

Algoritmo 10 Cifrado $\mathcal{E}^{hy}_{pk}$

Entrada: m

Salida: c

1: Obtener la semilla aleatoria γ del espacio de diseño del algoritmo asimétrico Ω.

2: Calcular el valor r_1 como aleatoriedad para el cifrado del algoritmo asimétrico a partir de la función de hash de salida extendida H:

$$r_1 := H(\gamma, \|m\|) \tag{3.112}$$

3: Calcular el valor r_2 como aleatoriedad para el cifrado del algoritmo simétrico a partir de la función de hash de tamaño fijo G:

$$r_2 := G(\gamma) \tag{3.113}$$

4: Obtener la primera mitad del texto cifrado c_1 a partir del algoritmo asimétrico con la introducción del ruido r_1:

$$c_1 := \mathcal{E}^{asym}_{pk}(\gamma||r_1) \tag{3.114}$$

5: Obtener la segunda mitad del texto cifrado c_2 a partir del algoritmo simétrico:

$$c_2 := \mathcal{E}^{sym}_{r_2}(m) \tag{3.115}$$

6: **return** $c_1||c_2$

En el algoritmo anterior la función de cifrado simétrico $\mathcal{E}^{sym}_a(m)$ al igual que su función de descifrado $\mathcal{D}^{sym}_a(c)$ se definen mediante el operador `XOR` como:

$$\begin{aligned} \mathcal{E}^{sym}_a(m) &= a \oplus m \\ \mathcal{D}^{sym}_a(c) &= a \oplus c \end{aligned} \tag{3.116}$$

Por otro lado, la función de descifrado del algoritmo con la transformada TFO $\mathcal{D}_{sk}^{hy}$ se define como:

Algoritmo 11 Descifrado $\mathcal{D}_{sk}^{hy}$

Entrada: c

Salida: m

1: Se obtiene el valor $\hat{\gamma}$ como el valor de la aleatoriedad calculado en la función de cifrado:

$$\hat{\gamma} := \mathcal{D}_{sk}^{asym}(c_1) \tag{3.117}$$

2: Calcular nuevamente el valor r_1 a partir de la aleatoriedad recuperada $\hat{\gamma}$:

$$r_1 := H(\hat{\gamma}, \|m\|) \tag{3.118}$$

3: Calcular el valor r_2:

$$r_2 := G(\hat{\gamma}) \tag{3.119}$$

4: Calcular el mensaje descifrado $\hat{m}$:

$$\hat{m} := \mathcal{D}_{r_2}^{sym}(c_2) \tag{3.120}$$

5: Comprobar si se obtiene el mismo mensaje:
6: **if** $c_1 == \mathcal{E}_{pk}^{asym}(\hat{\gamma}||r_1)$ **then**
7: Devolver el mensaje m' que coincide con el original m:

$$m := m' \tag{3.121}$$

8: **else**
9: El algoritmo ha fallado, se debe devolver un mensaje de rechazo:

$$m := \perp \tag{3.122}$$

10: **end if**
11: **return** m

Es importante recalcar que un atacante no puede distinguir si el mensaje obtenido es el original o un mensaje de rechazo, ya que este último mantiene la misma distribución y estructura aleatoria que un mensaje válido.

Por último, en el artículo [48] se demuestra que si el esquema asimétrico es γ-uniforme y (t_1, ε_1)-seguro en el cifrado de una dirección, entonces el esquema híbrido resultante es (t, ε)-seguro en el sentido de IND-CCA2. Los términos y parámetros de la definición se describen a continuación:

- Se define la seguridad de cifrado de una dirección para un algoritmo asimétrico como la probabilidad de que un adversario A sea capaz de obtener el valor de un mensaje plano x a partir de la llave pública pk y el texto cifrado y. Este adversario no tiene acceso a ninguna función del protocolo.

- Se habla de un algoritmo (t, ε)-seguro si se cumple que no existe ningún adversario que en tiempo t sea capaz de tener una ventaja mayor a ε.

- Se habla de que un algoritmo es γ-uniforme si para un texto plano x y un texto cifrado y cualesquiera se cumple que la probabilidad de que cifre en y es menor a γ. Esta propiedad asegura que los textos cifrados se distribuyen uniformemente y que conociendo y es difícil conocer x.

- En la demostración se calculan los valores de (t, ε) en función de los siguientes parámetros:

  - Tamaño del espacio de diseño de los algoritmos asimétrico l_1 y simétrico l_2.

  - Peticiones que hace el atacante a los distintos oráculos a los que tiene acceso en el modelo de oráculo aleatorio para atacar la parte del algoritmo asimétrico q_g y la parte del algoritmo simétrico q_h.
  - Se asume que el algoritmo simétrico es (t_2, ε_2)-seguro.

- En consecuencia la demostración muestra que el algoritmo asimétrico que tenia menores garantías de seguridad acaba siendo (t, ε)-seguro en el sentido de IND-CCA2:

$$\begin{aligned} t &= \texttt{min}(t_1, t_2) - \mathcal{O}((q_g + q_h)(l_1 + l_2)) \\ \varepsilon &= [2(q_g + q_h)\varepsilon_1 + \varepsilon_2 + 1]\left(1 - 2\varepsilon_1 - 2\varepsilon_2 - \gamma - 2^{-l_2}\right)^{-q_d} - 1 \end{aligned} \tag{3.123}$$

3.6. 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.

3.6.1. Seguridad esperada en Kyber

Para escribir esta sección se resume brevemente la información proporcionada en el artículo [1].

En primer lugar, el artículo presenta la seguridad frente a ataques IND-CCA2 tanto para adversarios clásicos como cuánticos. En este contexto, se establece una cota superior para la ventaja del adversario en romper la seguridad en el esquema Kyber:

- Se define la ventaja de un aversario A en el esquema M-LWE, denotada por $\text{Adv}^{\text{mlwe}}_{m,k,\eta}(A)$, en función de los parámetros de la tabla 3.2 y del número de muestras m. Dicha ventaja corresponde a la probabilidad de que un adversario distingua entre un par de muestras uniformes (a, b) y un par generado de acuerdo con el procedimiento descrito en la sección 3.4.1.2.

- Se denota la ventaja de un adversario A para distinguir una función pseudoaleatoria de una función verdaderamente aleatoria mediante $\text{Adv}^{\text{prf}}_{\text{PRF}}(A)$. Esta ventaja aparece porque las semillas γ utilizadas para generar la aleatoriedad en el cifrado se derivan de manera pseudoaleatoria, lo que las hace difíciles de diferenciar de valores verdaderamente aleatorios.

- En el caso de un oráculo aleatorio clásico (ROM), el artículo demuestra que se puede obtener la siguiente cota superior para la ventaja de un adversario A que realiza q_{ROM} consultas a los oráculos aleatorios. La cota se expresa en función de los adversarios B y C, que tienen un tiempo de ejecución similar al de A:

$$\text{Adv}^{\text{cca}}_{\text{Kyber}_{\text{ROM}}}(A) \leq \text{Adv}^{\text{mlwe}}_{k+1,k,\eta}(B) + \text{Adv}^{\text{prf}}_{\text{PRF}}(C) + 4q_{\text{ROM}}\delta \tag{3.124}$$

- En la deducción para el caso de un oráculo cuántico, se asume que la función de cifrado es pseudoaleatoria. Es decir, resulta difícil para un adversario distinguir un texto cifrado generado por la función de cifrado de uno elegido uniformemente al azar. Esta dificultad se cuantifica mediante la ventaja del adversario A, denotada por $\text{Adv}^{\text{pr}}_{\text{Cifrado}}(A)$. Esta conjetura se puede realizar debido a que como se dice en el artículo se desconoce la existencia de un ataque sobre la estructura de cifrado que sea más eficiente que atacar a la M-LWE.

- En el caso de un oráculo cuántico aleatorio (QROM), el artículo demuestra que se puede obtener la siguiente cota superior para la ventaja de un adversario A que realiza q_{QROM} consultas a los oráculos aleatorios. La cota se expresa en función de los adversarios B, C y D, que tienen un tiempo de ejecución similar al de A:

$$\text{Adv}^{\text{cca}}_{\text{Kyber}_{\text{QROM}}}(A) \leq 2\text{Adv}^{\text{mlwe}}_{k+1,k,\eta}(B) + \text{Adv}^{\text{pr}}_{\text{Cifrado}}(C) + \text{Adv}^{\text{prf}}_{\text{PRF}}(D) + 8q^2_{\text{ROM}}\delta \tag{3.125}$$

Considerando estas definiciones, para evaluar la seguridad real del esquema es necesario determinar la solidez del esquema M-LWE. Los demás términos se refieren a ataques contra los algoritmos simétricos implementados mediante las funciones del estándar FIPS [4], así como a la posibilidad de explotar fallos en el descifrado. Sin embargo, dado que el valor de δ es tan bajo, el número de peticiones requerido resulta tan elevado que no representa una amenaza práctica.

Por tanto, el análisis de seguridad se centra en los ataques conocidos contra la M-LWE, tal como se describe en [1]. Debido a las altas dimensiones de la retícula, actualmente no existen ataques prácticos que comprometan la estructura del anillo, por lo que todos los ataques conocidos se enfocan en romper la LWE.

Tal como se describe en [1] existen numerosos algoritmos para atacar la LWE. Sin embargo, en Kyber, dado que el atacante solo dispone de $m = (k + 1)n$ muestras, pueden descartarse los vectores de ataque Blum-Kalai-Wasserman [49] y los ataques de linealización [50]. En consecuencia, la seguridad práctica de Kyber, según lo descrito en [1], se analiza únicamente considerando los ataques basados en los algoritmos de Korkine-Zolatarev [51].

Por último, en el articulo se describe la seguridad frente a ataques de canal lateral:

- **Ataque de temporización**: en la implementación de Kyber no se utilizan ramas condicionales que dependan de información secreta. Es decir, las sentencias `if` únicamente evalúan valores binarios obtenidos a partir de funciones de comparación diseñadas para ejecutarse siempre en tiempo constante. Tampoco se emplean *look-up tables*, eliminando así otra posible fuente de filtración de información que un atacante puede aprovechar viendo el tiempo de ejecución del programa.

  Por otra parte, tampoco es viable explotar el tiempo requerido para la multiplicación de números en función de su tamaño, ya que se trabajan con entradas de 16 bits cuya multiplicación presenta un tiempo de ejecución constante. Finalmente, el problema de la reducción modular queda resuelto mediante algoritmos alternativos al uso directo del operador % en C, cuya duración de ejecución depende del divisor y del dividendo.

- **Ataque diferencial**: La implementación actual de Kyber no incluye protecciones específicas frente a ataques basados en la medición del consumo del dispositivo o de sus emisiones electromagnéticas durante la fase de descifrado. Para mitigar este tipo de ataques, se podrían emplear técnicas de enmascarado, aunque, según indica el artículo, esto podría aumentar el tiempo de ejecución hasta 5,5 veces.

- **Ataque de plantilla**: El esquema Kyber presenta actualmente dos vectores de ataque debido a su funcionamiento: el muestreo binomial [52] y la operación de la NTT [53]. Este riesgo podría mitigarse mediante algoritmos de enmascarado.

  Este ataque consiste en ejecutar el cifrado o descifrado con múltiples claves conocidas en un dispositivo similar al microcontrolador y analizar cómo varía su consumo para cada clave. Una vez precomputada esta base de datos, basta comparar las mediciones obtenidas durante la operación con las muestras registradas para extraer información sobre la clave desconocida. La diferencia con el ataque diferencial radica en que, dado que se conoce el comportamiento exacto del filtrado de información, se requiere recopilar menos datos para el análisis.

3.6.2. Seguridad esperada en Saber

Para escribir esta sección se resume brevemente la información proporcionada en el artículo [2].

En primer lugar, el artículo presenta la seguridad frente a ataques IND-CCA2 tanto para adversarios clásicos como cuánticos. En este contexto, se establece una cota superior para la ventaja del adversario en romper la seguridad en el esquema Saber:

- Al igual que en Kyber, en Saber uno de los adversarios A ataca la estructura de la Mod-LWR. Su ventaja se denota $\mathrm{Adv}^{\mathrm{mlwr}}_{l,l,\nu,q,p}(A)$, con los parámetros de la tabla 3.3. El objetivo de este adversario es distinguir entre muestras uniformes (a, b) y pares generados según el procedimiento descrito en la sección 3.4.2.2.
- Además, se define un adversario A cuya misión es distinguir si la matriz `A` usada para generar la clave pública proviene de una semilla o si es una matriz completamente uniforme. Esta ventaja se denota $\mathrm{Adv}^{\mathrm{prf}}_{\mathrm{gen}}(A)$. No obstante, en la práctica esta ventaja es nula, puesto que la semilla se muestrea de forma uniforme en el protocolo real.
- En el caso de un oráculo aleatorio clásico (ROM) para la seguridad IND-CPA, el artículo demuestra que la ventaja de un adversario A queda acotada por las ventajas de tres adversarios B, C y D. Dichos adversarios modelan, respectivamente, la generación, el cifrado y la pseudoaleatoriedad en la generación de claves, y sus tiempos de ejecución son comparables al de A. Por tanto, la cota de la ventaja de A:

$$\mathrm{Adv}^{\mathrm{cpa}}_{\mathrm{Saber}}(A) \leq \mathrm{Adv}^{\mathrm{mlwr}}_{l,l,\nu,q,p}(B) + \mathrm{Adv}^{\mathrm{mlwr}}_{l+1,l,\nu,q,q/\xi}(C) + \mathrm{Adv}^{\mathrm{prf}}_{gen}(D) \tag{3.126}$$

  donde $\xi = \texttt{min}\left(\frac{q}{p}, \frac{p}{t}\right)$
- Para seguridad IND-CCA2 el artículo demuestra que para un adversario A que realiza $q_{\mathcal{H}}$ consultas al oráculo $\mathcal{H}$ (función de hash de salida fija) y $q_{\mathcal{G}}$ consultas al oráculo $\mathcal{G}$ (función de salida extendida), entonces su ventaja viene acotada por una expresión que depende de las ventajas de un adversario B de la forma:

$$\mathrm{Adv}^{\mathrm{cca}}_{\mathrm{Saber}_{\mathrm{ROM}}}(A) \leq 3\mathrm{Adv}^{\mathrm{cpa}}_{\mathrm{Saber}}(B) + q_{\mathcal{G}}\delta + \frac{2q_{\mathcal{G}} + g_{\mathcal{H}} + 1}{2^{256}} \tag{3.127}$$

- En el caso de un oráculo cuántico aleatorio (QROM) con seguridad IND-CCA2, el artículo demuestra que la ventaja de un adversario A que realiza $q_{\mathcal{H}}$ consultas al oráculo $\mathcal{H}$ y $q_{\mathcal{G}}$ consultas al oráculo $\mathcal{G}$, para mensajes de tamaño $|M|$ y en presencia de un adversario B, puede acotarse superiormente de la siguiente manera:

$$\mathrm{Adv}^{\mathrm{cca}}_{\mathrm{Saber}_{\mathrm{QROM}}}(A) \leq \frac{2q_{\mathcal{H}}}{\sqrt{2^{256}}} + 4q_{\mathcal{G}}\sqrt{\delta} + 2(q_{\mathcal{G}} + q_{\mathcal{H}})\sqrt{\mathrm{Adv}^{\mathrm{cpa}}_{\mathrm{Saber}}(B) + \frac{1}{\|M\|}} \tag{3.128}$$

Con estas definiciones presentes, para evaluar la seguridad del esquema resulta fundamental analizar todos los elementos de la ecuación 3.128. Al igual que en Kyber, aparecen términos lineales que dependen de la cantidad de consultas a los oráculos aleatorios, con un orden de magnitud similar y que, en la práctica, pueden despreciarse. Sin embargo, también existe un término no lineal con respecto al número de consultas realizadas y a la ventaja obtenida en la Mod-LWR. En consecuencia, este esquema puede considerarse menos seguro que Kyber, pues la ventaja escala de manera no lineal con el número de consultas.

Además, como se señala en [35], no se han identificado ataques específicos contra la Mod-LWR, por lo que se considera que la ventaja es equivalente a la de romper la LWE mediante ataques basados en los algoritmos de Korkine-Zolatarev [51]. Sin embargo, el propio NIST descartó a Saber en la tercera ronda, ya que, aunque comparte la misma base en retículas que Kyber, disponía de menos respaldo en la literatura que confirmara que la Mod-LWR no presentase vulnerabilidades particulares [26].

Por último, en el articulo se describe la seguridad frente a ataques de canal lateral:

- **Ataque de temporización**: no supone un problema en Saber debido a que todos los algoritmos se diseñan para funcionar en tiempo constante. Además, el uso del módulo 2 facilita estas operaciones.
- **Ataque diferencial**: aunque existen ataques dirigidos a la estructura de Saber, el artículo original sugiere una implementación enmascarada que protege frente a ataques simples o de primer orden. En este trabajo no se utiliza dicha implementación enmascarada, sino la versión "limpia". Sin embargo, debido a que las operaciones se realizan en módulo 2, la máscara resulta ser muy eficiente.
- **Ataque de plantilla**: al igual que en el esquema Kyber, existen ataques dirigidos a la implementación de las multiplicaciones matriciales en Saber. Aunque aún no se han incorporado técnicas para reordenar aleatoriamente las operaciones en este esquema, se ha demostrado que estas estrategias son muy eficaces para algoritmos similares sin tener mucha pérdida de rendimiento.

3.6.3. Seguridad esperada en HQC

Para escribir esta sección se resume brevemente la información proporcionada en el artículo [3].

En primer lugar y al igual que para los algoritmos anteriores se presenta la reducción de seguridad en el esquema IND-CCA2:

- En este caso el ataque fundamental que realiza un adversario A es contra el problema decisional de descifrado de síndromes en códigos cuasi-cíclicos (`s-QCSD`) tal como se describe en la sección 3.4.3.1. La ventaja $\mathrm{Adv}_{\text{s-DQSCSD}(A)}$ se define como la capacidad del adversario de difrenciar parejas aleatorias (a, b) de las creadas mediante los códigos lineales.
- Teniendo esto en cuenta esto, en el modelo de oráculo aleatorio clásico ROM se modela la ventaja de un adversario A en el esquema IND-CPA en base a dos adversarios B y C que representan la ventaja que tiene el adversario en romper los bloques u y v del algoritmo 27 respectivamente:

$$\mathrm{Adv}^{\mathrm{cpa}}_{\mathrm{HQC}_{\mathrm{ROM}}}(A) \leq \mathrm{Adv}_{\text{2-DQSCSD}(B)+\mathrm{Adv}_{\text{3-DQSCSD}(C)}}(3.129)$$

- En el modelo IND-CCA2 se tiene en cuenta el error introducido por la resolución de muestreador, el número de peticiones al oráculo aleatorio q_{ROM} y al oráculo de descifrado $q_{\mathbb{D}}$. Por tanto, la ventaja en de un adversario A en función del adversario B queda como:

$$\mathrm{Adv}^{\mathrm{cca}}_{\mathrm{HQC}_{\mathrm{ROM}}}(A) \leq \frac{1}{2^{256} \cdot 2^{16}} + \frac{3q_{\mathrm{ROM}}}{2^{256}} + (q_{\mathrm{ROM}} + q_D)\delta + 2\mathrm{Adv}^{\mathrm{cpa}}_{\mathrm{HQC}_{\mathrm{ROM}}}(B) \tag{3.130}$$

- En agosto de 2025 no existe, que conozca, una reducción QROM completa que demuestre seguridad IND-CCA2 para HQC. Los trabajos existentes tratan nociones más genéricas o más débiles y no se aplican directamente. [3] admite además una pérdida de seguridad de aproximadamente la mitad en el QROM, por lo que la reducción sería similar a la de Saber. En consecuencia, la confianza en su resistencia cuántica es menor que la de Kyber o Saber, aunque HQC fue seleccionado en la ronda 4 del NIST entre otros candidatos basados en código que comparten esta problemática [27].

Con esta reducción de seguridad, aunque no exista una demostración rigurosa para el modelo QROM y, a diferencia de los esquemas Kyber y Saber cuyas garantías se basan en la LWE, la seguridad de HQC se fundamenta en la resistencia al problema de decodificación de síndromes en códigos QC. Por ello fue seleccionado en la ronda 4 del NIST [27].

La vulnerabilidad principal por tanto reside en la decodificación de síndromes, un problema ampliamente estudiado durante más de cincuenta años [3] donde el mejor ataque contra este esquema [54] tiene complejidad exponencial. Por otro lado, en cuanto a ataques que explotan la estructura

QC, se ha descrito un ataque [55] que depende de que el polinomio $X^n - 1$ se descomponga en muchos factores lineales. No obstante, para $n = 57637$, valor elegido en HQC-5, dicha descomposición contiene únicamente dos factores lineales. Por tanto, ese ataque no compromete la seguridad del esquema.

Por último, se describe las vulnerabilidades de HQC frente a ataques de canal lateral. En el artículo [3] solo se abordan ataques de temporización.

- **Ataque diferencial**: Para analizar este ataque se recurre al artículo [56], en el que se examinan las trazas de consumo del dispositivo durante la fase de descapsulado. Los autores demuestran que el decodificador Reed–Muller presenta fugas de información y proponen, como contramedida, dividir las palabras procesadas en un mayor número de bloques y tratar cada bloque de forma independiente. Según ellos, esta técnica incrementa la resistencia frente a ataques diferenciales. No obstante, no aportan un análisis detallado del coste ni del impacto en la eficiencia de la implementación. Otros estudios que realizan un análisis general del enmascarado en HQC [57] concluyen que este tipo de enmascarado puede resultar muy costoso.

Capítulo 4

Desarrollo

En este capítulo se describe el código y la implementación propia realizada en base a los algoritmos proporcionados por el NIST.

4.1. Adaptación de Primitivas Criptográficas

La primera fase del desarrollo se centró en la extracción y depuración de los núcleos algorítmicos en lenguaje C, con el objetivo de garantizar su portabilidad independientemente de la plataforma escogida.

Las implementaciones de referencia presentan dependencias de compilación ligadas a entornos UNIX, presencia de trazas de depuración (*debug prints*) dispersas en múltiples ficheros y cierto acoplamiento entre la lógica criptográfica y los vectores de prueba. Por ello, en esta sección se detallan las refactorizaciones realizadas y se describe la estructura final de los ficheros fuente para cada algoritmo.

4.1.1. Kyber y Saber: Implementaciones de Referencia

Para los algoritmos Kyber y Saber se han integrado las implementaciones de referencia presentadas por los autores en las rondas 3 y 4 del NIST [26] [27].

Se han aplicado modificaciones estructurales menores, siendo la más relevante el desacoplamiento del módulo de generación de aleatoriedad. En la versión original, el código dependía de `rng.h` para generar secuencias deterministas a partir de una semilla fija. Esta dependencia se ha sustituido por la interfaz genérica `randombytes.h`, la cual permite inyectar una fuente de entropía real del sistema. Para más detalles sobre esta gestión de aleatoriedad, consultar la sección 4.3.

4.1.1.1. Estructura del código en Kyber

En la tabla 4.1 se describe brevemente la estructura lógica del código de Kyber tras la limpieza del código [1].

Categoría	Ficheros (.c/.h)	Descripción Funcional
Núcleo del esquema	`kem, indcpa`	Implementan el protocolo de intercambio de claves. En `indcpa` se encuentran los esquemas básicos de la M-LWE para obtener seguridad CPA mientras que en el fichero `kem` mediante la TFO se implementa la seguridad CCA2.
	`api`	Archivo de api original del NIST. Tiene la problemática de ser poco flexible y dar problemas de enlazado si se usan varios cifrados distintos.
	`kyber_wrapper`	Envoltorio desarrollado para abstraer las funcionalidades de Kyber.
	`verify`	Función para comparar dos vectores en tiempo constante.
Matemáticas	`poly, polyvec`	Definen las estructuras de datos para operaciones algebraicas sobre el anillo R_q.
	`ntt, reduce`	Implementan la multiplicación rápida de polinomios mediante la NTT y la reducción modular eficiente (se usa un módulo no potencia de 2 $q = 3329$).
	`cbd`	Generación de ruido determinista necesario para la seguridad LWE mediante la distribución binomial.
Hashing	`fips202, sha, symetric`	Implementación del estándar SHA-3 (Keccak) junto a las interfaces necesarias para kyber.
Sistema	`randombytes`	Función para generar la aleatoriedad en el esquema.
	`params`	Tabla de parametros del esquema de seguridad a utilizar.

Tabla 4.1: Desglose funcional de los ficheros fuente de Kyber.

4.1.1.2. Estructura del código en Saber

En la tabla 4.2 se describe brevemente la estructura lógica del código de Saber tras la limpieza del código [2].

Categoría	Ficheros (.c/.h)	Descripción Funcional
Núcleo del esquema	`kem, SABER_indcpa`	Implementan el mecanismo de encapsulamiento. En `SABER_indcpa` se define la lógica base del problema Mod-LWR para seguridad CPA, mientras que `kem` aplica la TFO para alcanzar seguridad CCA2.
	`api`	Archivo de api original del NIST. Tiene la problemática de ser poco flexible y dar problemas de enlazado si se usan varios cifrados distintos.
	`saber_wrapper`	Envoltorio desarrollado para abstraer las funcionalidades de Saber.
	`verify`	Función auxiliar para comparación de vectores en tiempo constante.
Matemáticas	`poly, pack_unpack`	Estructuras de datos para el anillo R_q y rutinas de optimización del ancho de banda.
	`poly_mul`	Implementa la multiplicación polinomial mediante Toom-Cook y Karatsuba.
	`cbd`	Generación de ruido determinista mediante la Distribución Binomial Centrada.
Hashing	`fips202`	Implementación del estándar SHA-3 (Keccak).
Sistema	`randombytes`	Función para generar la aleatoriedad en el esquema.
	`SABER_params`	Tabla de parametros del esquema de seguridad a utilizar.

Tabla 4.2: Desglose funcional de los ficheros fuente de Saber.

4.1.2. HQC: Selección de PQClean

Para HQC, se descartó la implementación de la ronda 4 del NIST en favor de la versión mantenida por el proyecto PQClean [58] [59]. Esta decisión se fundamenta en la arquitectura de la versión original, la cual presenta un fuerte acoplamiento con el generador pseudoaleatorio determinista diseñado para pruebas Known Answer Test o Test de Respuesta Conocida(KAT). La versión de PQClean ofrece una interfaz para modificar la fuente de aleatoriedad de manera sencilla. Esto facilita el uso de una semilla fija para validación o una fuente de aleatoriedad real (TRNG).

Es importante señalar que la implementación actual no garantiza seguridad IND-CCA2. Esto se debe a que, durante la fase de desencapsulado, el mecanismo de verificación carece de rechazo implícito. Si el texto cifrado recalculado no coincide con el recibido, la función retorna un fallo explícito en lugar de generar un valor pseudoaleatorio indistinguible para un atacante. Esta limitación ha sido asumida dentro del alcance del proyecto priorizando la claridad conceptual, ya que, aunque su inclusión afectaría marginalmente a las métricas de rendimiento, no compromete la validación funcional de los algoritmos. Si se quisiera implementar la solución para un entorno de producción sería necesario utilizar la TFO.

4.1.2.1. Estructura del código en HQC

En la tabla 4.3 se describe brevemente la estructura lógica del código de HQC tras la limpieza del código [58].

Categoría	Ficheros (.c/.h)	Descripción Funcional
Núcleo del esquema	`code, hqc`	En `code` se define una interfaz a los mecanismos de corrección de errores, mientras que `hqc` implementa todo el mecanismo de intercambio de claves.
	`api`	Archivo de api original del NIST. Tiene la problemática de ser poco flexible y dar problemas de enlazado si se usan varios cifrados distintos.
	`hqc_wrapper`	Envoltorio desarrollado para abstraer las funcionalidades de HQC.
	`parsing`	Funciones auxiliares para dar formato a las claves y los textos cifrados.
Matemáticas	`reed_solomon, reed_muller`	Implementan la codificación y decodificación de códigos correctores de errores.
	`gf, gf2x, fft`	Implementa la aritmética en cuerpos de Galois y multiplicación de polinomios eficiente mediante la Transformada Rápida de Fourier.
	`vector`	Implementa la aritmética entre vectores de bits.
Hashing	`fips202, shake_ds, shake_prng`	Implementación del estándar SHA-3 (Keccak) junto a las interfaces necesarias para HQC.
Sistema	`randombytes`	Función para generar la aleatoriedad en el esquema.
	`parameters, domains`	Tabla de parametros del esquema de seguridad a utilizar.

Tabla 4.3: Desglose funcional de los ficheros fuente de HQC.

4.2. Unificación de Interfaces (Capa Wrapper)

Se ha desarrollado un módulo envoltorio (*wrapper*) para cada algoritmo con el objetivo de encapsular las implementaciones de referencia del NIST [26] [27] bajo una interfaz común. La estructura general de los ficheros de cabecera, denominados `AlgorithmWrapper.h`, se detalla en el Listing 4.1.

El propósito principal de esta capa de abstracción es homogeneizar las distintas implementaciones provistas por el NIST. Esto permite instanciar los algoritmos mediante un inicializador común, configurado según el nivel de seguridad definido durante la compilación de la librería. Asimismo, este diseño resuelve los conflictos de enlazado (*linking*) derivados del uso de símbolos idénticos en las APIs originales, facilitando así la intercambiabilidad dinámica entre los distintos algoritmos criptográficos.

Finalmente, en lo relativo a la gestión de memoria, se ha adoptado una filosofía donde la responsabilidad recae íntegramente en el usuario, quien debe encargarse de la asignación y liberación de los *buffers*.

```c
// Estructura comun para Kyber, HQC y Saber
// Se sustituye "ALG" por el nombre del algoritmo especifico

#include "alg_wrapper.h"
#include "api.h"

// Nivel de seguridad actual (Estado interno)
static alg_security_level_t current_security_level;
static int is_initialized = 0;

// Inicializacion del esquema
int alg_init(alg_security_level_t security_level) {
  // Validacion del nivel de seguridad
  if (security_level != ALG_SEC_LEVEL_1 &&
  security_level != ALG_SEC_LEVEL_3 &&
  security_level != ALG_SEC_LEVEL_5) {
    return -1;
  }

  current_security_level = security_level;
  is_initialized = 1;
  return 0;
}

// Getters de tamaños (Wrappers directos a la API nativa)
size_t alg_get_public_key_size(void) { return CRYPTO_PUBLICKEYBYTES; }
size_t alg_get_secret_key_size(void) { return CRYPTO_SECRETKEYBYTES; }
size_t alg_get_ciphertext_size(void) { return CRYPTO_CIPHERTEXTBYTES; }
size_t alg_get_shared_secret_size(void) { return CRYPTO_BYTES; }

// Generacion de claves
int alg_keypair(unsigned char *pk, unsigned char *sk) {
  if (!is_initialized) return -1;
  return crypto_kem_keypair(pk, sk);
}

// Encapsulamiento
int alg_encapsulate(unsigned char *ct, unsigned char *ss, const unsigned char *pk) {
  if (!is_initialized) return -1;
  return crypto_kem_enc(ct, ss, pk);
}

// Desencapsulamiento
int alg_decapsulate(unsigned char *ss, const unsigned char *ct, const unsigned char *
  sk) {
  if (!is_initialized) return -1;
  return crypto_kem_dec(ss, ct, sk);
}
```

Listing 4.1: Estructura genérica del wrapper para los algoritmos PQC

4.3. Precompilados y gestión de la aleatoriedad.

Para agilizar la integración de los algoritmos de cifrado en sistemas de propósito general, se ha optado por encapsular la lógica en bibliotecas estáticas (.lib). Gracias a esto, la interacción a nivel de usuario se reduce al uso de una interfaz simplificada, similar a la mostrada en el Listing 4.1, facilitando su incorporación en proyectos externos.

Es importante señalar que, debido a la arquitectura actual, la selección del nivel de seguridad es estática. Para alternar dinámicamente entre distintos niveles, sería necesario refactorizar el código para desacoplarlo de los archivos de parámetros dependientes de la compilación al incluirlos en el init de los algoritmos. Sin embargo, dado que el alcance de este trabajo se centra en el análisis de algoritmos con un nivel de seguridad equivalente a 256 bits, no se ha considerado prioritaria dicha flexibilidad en esta etapa.

En cuanto a la gestión de aleatoriedad, como se mencionó anteriormente, existen dos implementaciones en función de la función que se desea que cumpla el algoritmo:

- **Versión determinista (`rng.h`):** Utilizada para generar vectores de prueba conocidos (KAT). Genera los secretos compartidos a partir de una semilla inicial fija. Esta implementación inicializa la entropía con un valor conocido y utiliza `AES256` para derivar una secuencia de números pseudoaleatorios, permitiendo así la verificación mediante archivos `.req` y `.rsp`.

- **Versión de producción (`randombytes.h`):** Implementa un generador de números aleatorios criptográficamente seguro. En sistemas de propósito general, se obtiene la entropía directamente del sistema operativo, mientras que en la plataforma PSOC utiliza el generador de hardware (TRNG) nativo de la placa.

  Para la implementación en sistemas de propósito general se ha integrado la librería disponible en [60], utilizada también en el proyecto PQClean [59]. De este modo, se garantiza el uso de un generador criptográficamente seguro, cumpliendo con los requisitos de seguridad que serán validados posteriormente mediante baterías de tests estadísticos.

4.4. Arquitectura del Agente Criptográfico y de las comunicaciones (C++)

Sobre la base de las librerías en C descritas anteriormente, se ha construido una capa de aplicación en C++ que integra las capacidades criptográficas con un módulo de comunicaciones serie.

4.4.1. Modelo de comunicaciones

Para implementar las comunicaciones, por su sencillez se ha optado por la comunicación mediante UART conectando directamente la placa del PSOC al ordenador por USB. Los parámetros generales de las comunicaciones utilizados son:

- **BaudRate:** 115200 baudios.
- **Data Bits:** 8 bits.
- **Paridad:** Ninguna (*No Parity*).
- **Bits de parada:** 1 bit.
- **Control de flujo:** Ninguno. La gestión de la integridad se delega a los buffers de software, aunque se mantiene activa la señal DTR[1] para garantizar la estabilidad del enlace.
- **Buffers (I/O):** 8192 bytes para reducir la latencia en transferencias grandes.
- **Timeouts:** Configuración no bloqueante con un tiempo de espera máximo de 50 ms por lectura.

Tal como se puede ver en la figura 4.1 las comunicaciones se implementan mediante tres clases:

4.4.1.1. Clase `SerialComunication`

Esta es la clase fundamental dentro del modelo de comunicaciones pues se encarga de abstraer la API de Windows (`Win32 API`) para manejar de manera sencilla los datos enviados por el canal serie. Para ello, a continuación se describen brevemente los elementos de esta clase:

[1] La señal DTR (*Data Terminal Ready*) indica al dispositivo que el PC está listo para establecer la comunicación. Esta señal se fuerza a estar activa para habilitar el canal de datos y/o prevenir que la placa de desarrollo no se reinicie.

- **Inicialización y selección del puerto**: para iniciar la comunicación es necesario identificar primero el puerto asignado al dispositivo PSOC. Para ello, se utiliza el método `AvailablePorts`, que escanea el registro del sistema para listar los puertos COM activos. Una vez seleccionado el puerto objetivo, las funciones `openPort` y `closePort` gestionan la apertura del descriptor de archivo y la liberación de recursos, respectivamente.
- **Envío de mensajes**: para enviar mensajes el mecanismo es sencillo pues basta con usar la función `send` con el mensaje deseado y se enviará por el canal serie automáticamente.
- **Recepción de mensajes**: para recibir mensajes y no tener que bloquear el flujo principal del programa, se ha implementado un patrón Productor-Consumidor mediante multihilo:
  - **Hilo de Captura**: mediante la función `startReceiving`, se despliega un hilo secundario que ejecuta un bucle de lectura. Este hilo monitoriza constantemente el puerto serie y, al detectar bytes entrantes, los transfiere inmediatamente a un buffer intermedio (`receiveBuffer`) y así liberar espacio del buffer del puerto serie. Para evitar el consumo excesivo de CPU durante la espera, se incluye un breve retardo (*sleep*) de 10 ms entre ciclos de lectura.
  - **Sincronización y acceso**: dado que el buffer de recepción es un recurso compartido, todas las operaciones de escritura (desde el hilo) y lectura (desde la aplicación) están protegidas por un `std::mutex` para evitar condiciones de carrera.
  - **Interfaz de Consumo**: para recuperar los datos enviados se utiliza el método `getReceivedMessage`, el cual extrae de forma segura los mensajes almacenados en el buffer circular, garantizando la integridad de los datos en un entorno concurrente.

4.4.1.2. Clase `CircularBuffer`

Esta clase implementa una estructura de datos de tipo FIFO sobre un espacio de memoria preasignado. Su función crítica es actuar como almacenamiento intermedio o buffer entre el hilo de recepción de la UART y la lógica de aplicación. Al utilizar un puntero de lectura y uno de escritura que rotan sobre el array, se consigue una inserción y extracción de datos eficiente sin la sobrecarga computacional que implicaría el redimensionamiento dinámico de vectores.

4.4.1.3. Clase `SerialPacketTransport`

Esta clase es esencial para el modelo criptográfico mediante agentes descrito en la figura 3.1 y para garantizar que no se sature el microcontrolador pues los tamaños de claves pueden ser muy elevados entre 1312 y 14421 bytes como se describió en las tablas 3.2, 3.3 y 3.4. Para ello, se estructura la comunicación mediante un protocolo de enpaquetado ligero como una capa superior a la clase `SerialComunication`, permitiendo el envío de bloques de datos discretos. Esta clase tiene las siguientes características:

- **Protocolo de encabezado**: para distinguir el inicio y fin de cada mensaje en el flujo continuo del puerto serie, la clase antepone a cada paquete una cabecera de 16 bits (2 bytes) en formato Little Endian. Esta cabecera indica la longitud exacta de la carga útil, permitiendo gestionar paquetes de hasta 65.535 bytes (64 KB).
- **Empaquetación en el envío**: el método `sendPacket` no envía el mensaje completo de golpe, si no que realiza una fragmentación en bloques de 64 bytes. Esta estrategia se utiliza para evitar saturar los buffers internos del PSOC y mejorar la estabilidad de la transmisión.
- **Recepción con reensamblado y Timeout**: el método `receivePacket` implementa una lógica de lectura en dos fases: primero espera la recepción de los 2 bytes de cabecera para conocer el tamaño esperado y, posteriormente, acumula los fragmentos recibidos hasta completar el paquete. Para evitar bloquear el programa permanentemente se introduce un tiempo de espera o timeout, que aborta la operación si el paquete no se completa dentro del tiempo estipulado.

Cabe justificar que, si bien la capa inferior utiliza un modelo multihilo para la captura de datos, esta clase expone una interfaz bloqueante. Esta decisión se alinea con la naturaleza secuencial del protocolo de intercambio de claves, donde cada paso criptográfico depende estrictamente de la finalización del paso anterior y la recepción completa de la respuesta del otro agente, haciendo innecesaria una gestión asíncrona a nivel de aplicación.

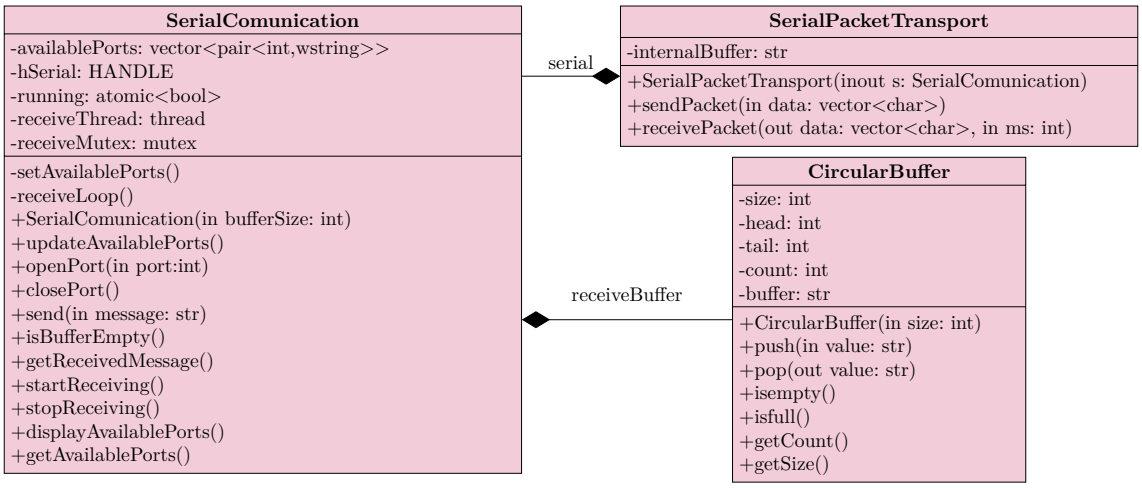

Figura 4.1: Diagrama de clases en UML para implementar las comunicaciones por serial.

4.4.2. Modelo criptográfico

Para implementar el modelo de agentes de la figura 3.1 se utiliza una interfaz común `keyexchange` para acceder a los distintos algoritmos de cifrado postcuántico y mediante la clase `KEMProtocol` se implementa el protocolo. Además, para ejecutar los tests de rendimiento o benchmarks para cada algoritmo se utiliza la clase auxiliar `AlgorithmTester`.

4.4.2.1. Plantilla `SecureVector`

En el desarrollo de software criptográfico robusto, normalmente se recomienda el uso de memoria estática (`stack`) para el almacenamiento de material sensible, como las claves privadas. Esto se debe a que la gestión de memoria dinámica (`heap`) introduce una superficie de ataque que puede ser explotada para extraer información residual o fragmentada.

No obstante, dada la naturaleza modular de este banco de pruebas y la necesidad de instanciar dinámicamente algoritmos con requisitos de memoria heterogéneos, el uso del `heap` resulta inevitable. Para mitigar los riesgos de seguridad asociados, se ha implementado la plantilla `SecureVector`.

La plantilla `SecureVector` actúa como un contenedor seguro basado en `std::vector`. La principal modificación reside en la sobrecarga de la rutina de desasignación: antes de liberar el bloque de memoria y devolverlo al sistema operativo, la clase sobrescribe explícitamente el contenido con ceros. Esta medida preventiva garantiza que no permanezcan trazas de información confidencial en la memoria no asignada.

4.4.2.2. Clase `keyexchange`

Dado que los métodos de los algoritmos de cifrado postcuántico son comunes, se ha creado una interfaz unificada que resuelve su instanciación mediante polimorfismo. De esta forma, las clases derivadas representan los algoritmos específicos, añadiendo una capa de abstracción sobre los envoltorios en C descritos en el listing 4.1.

4.4.2.3. Clase `KEMProtocol`

Esta clase implementa el protocolo criptográfico descrito en la figura 3.1 mediante una máquina de estados finitos. Dado que el flujo de ejecución es secuencial y dependiente del extremo de la comunicación, el constructor recibe dos parámetros fundamentales: el rol a desempeñar (Alice como `INITIATOR` o Bob como `RESPONDER`) y una referencia al algoritmo criptográfico deseado (Saber, Kyber o HQC).

Para gestionar la variabilidad de algoritmos se aplica el patrón de diseño Estrategia. Esto permite abstraer las operaciones primitivas (generación de claves, encapsulado y decapsulado) tras una interfaz común, desacoplando la lógica de control de estado de la implementación matemática específica de cada algoritmo.

El comportamiento de la máquina de estados varía según el rol asignado:

- **Rol INITIATOR (Alice)**: Comienza en el estado `INIT`. Al invocar el método `start _negotiation`, la clase genera el par de claves, transiciona al estado `WAITING_FOR_CT` y devuelve la clave pública. Esta clave queda disponible y debe ser transmitida por el canal de comunicación (en este proyecto, enviada vía serie al PSoC).
- **Rol RESPONDER (Bob)**: Inicia en el estado `INIT` y permanece a la espera. Su transición ocurre únicamente cuando se solicita procesar un mensaje externo mediante el método `process_message`.

Cuando la clase recibe un mensaje mediante el método `process_message` y se encuentra en un estado válido de espera (es decir, en `INIT` si es Bob o en `WAITING_FOR_CT` si es Alice), ejecuta la lógica criptográfica correspondiente: encapsulación para el responder o decapsulación para el initiator. Tras finalizar la operación con éxito, la máquina transiciona al estado final `ESTABLISHED`, completando el intercambio.

Para proteger la información crítica, el secreto compartido y la clave privada se gestionan mediante la plantilla `SecureVector` mencionada anteriormente. De esta manera el sistema contempla un estado de `FAILURE` al que se transiciona si se solicita una operación inválida para el rol actual, etapa actual o si falla alguna operación del intercambio de claves.

Cabe destacar que para cumplir seguridad IND-CCA2 las funciones del intercambio de claves no pueden fallar debido a que ello implicaría que el atacante puede obtener información del oráculo. Si bien en algunas implementaciones se contempla la posibilidad de fallo si la función `randombytes` se queda sin entropía al consultar el TRNG, en el alcance de este trabajo no se contempla dicha posibilidad, asumiendo disponibilidad de entropía infinita.

4.4.2.4. Clase `AlgorithmTester`

Esta clase actúa como un entorno de ejecución controlado para realizar benchmarks de los algoritmos de cifrado. Aprovechando el patrón Estrategia, la clase `AlgorithmTester` interactúa con los algoritmos a través de su interfaz común, lo que permite someter a cualquiera de los esquemas implementados (Kyber, Saber, HQC) a las mismas baterías de pruebas bajo condiciones idénticas. La descripción detallada de la metodología de pruebas y sus resultados se encuentra en la sección 4.7.

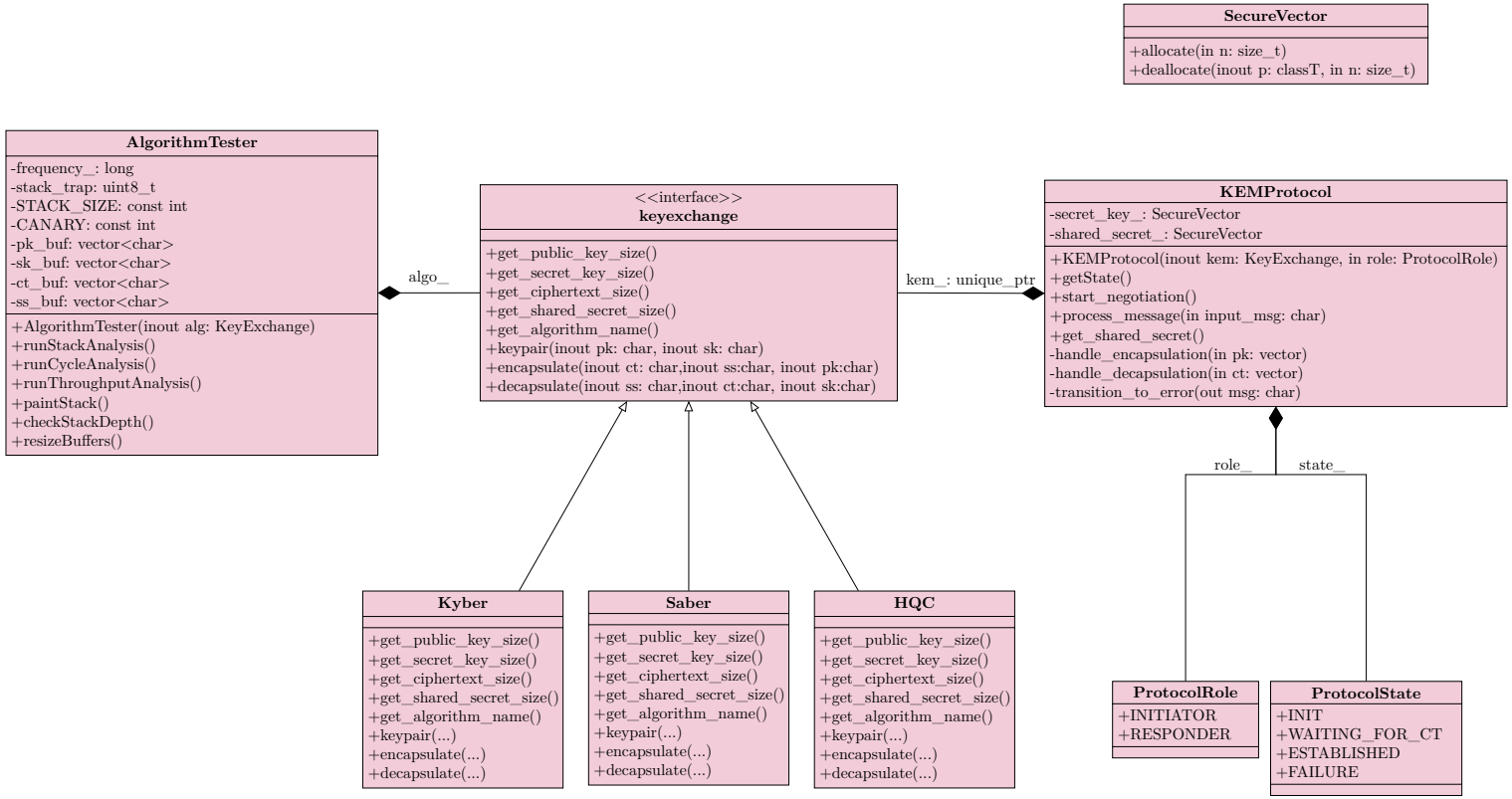

Figura 4.2: Diagrama de clases en UML para implementar la lógica criptográfica.

4.5. Implementación de algoritmos en el microcontrolador

La integración de algoritmos postcuánticos en sistemas embebidos presenta desafíos significativos debido a la limitación de recursos. No obstante, la migración del código fuente resulta directa gracias a las implementaciones portables. La única adaptación necesaria a nivel de plataforma es la implementación de la función `randombytes`, para la generación de los TRNG mediante el hardware.

Sin embargo, el desafío principal residió en la selección y configuración del entorno de desarrollo (IDE). Inicialmente se evaluó el uso de *PSoC Creator*, la herramienta tradicional para esta familia de microcontroladores. Durante las pruebas con el algoritmo HQC, que demanda una cantidad elevada de memoria (aprox. 170 KB), la ejecución fallaba sistemáticamente derivando en excepciones de hardware.

El análisis en tiempo de depuración reveló un comportamiento anómalo en la gestión de memoria: el puntero de pila (Stack Pointer) o los accesos a memoria apuntaban a direcciones en el rango `0x78...`, una región inválida situada aproximadamente a 1.8 GB de distancia del espacio direccionable real de la SRAM, cuya base se encuentra en `0x08...` según el manual de arquitectura [61]. Adicionalmente, PSoC Creator impone restricciones en la arquitectura de seguridad, limitando el acceso directo a los registros del TRNG desde el núcleo principal (Cortex-M4), forzando una arquitectura maestro-esclavo con el núcleo Cortex-M0+ que añade complejidad innecesaria para este proyecto.

Debido a estas limitaciones críticas, se optó por migrar el desarrollo al entorno *ModusToolbox*. Esta plataforma moderna permite el uso directo de la *Peripheral Driver Library* (PDL) para controlar el TRNG desde el núcleo M4 y gestiona el mapa de memoria de forma más eficiente, permitiendo la correcta ejecución de HQC. Aunque no se ha aislado la causa raíz del fallo en el entorno anterior, el éxito en ModusToolbox se atribuye a una combinación de factores en la gestión del *Linker Script*:

- **Gestión dinámica de memoria (Heap vs Stack):** Para soportar tamaños de clave variables, las claves y textos cifrados se reservan dinámicamente en el heap. Se hipotetiza que el script de enlazado de PSoC Creator define regiones rígidas o contiguas que provocan una colisión temprana entre el stack (creciendo hacia abajo) y el heap (creciendo hacia arriba) cuando se demandan grandes bloques contiguos para HQC.
- **Optimizaciones del Compilador GCC:** ModusToolbox integra versiones más recientes del toolchain GCC ARM. Es probable que las diferencias en las optimizaciones de gestión de pila y la definición de las secciones de memoria en el script de arranque eviten el desbordamiento o la corrupción de punteros que se observaba anteriormente.

4.6. Diagrama de secuencia

En la figura 4.3 se detalla el diagrama de secuencia diseñado para validar el funcionamiento del protocolo de establecimiento de claves. La interacción sigue una arquitectura estricta de Maestro-Esclavo, donde el servidor (ordenador de propósito general) orquesta la comunicación y el PSOC actúa como periférico. Esta decisión de diseño permite desacoplar la lógica de control del dispositivo embebido, centralizando la gestión de estados y el acceso al canal serie en el servidor, lo cual facilita la escalabilidad en un hipotético entorno industrial con múltiples nodos.

El flujo de ejecución consta de tres fases diferenciadas:

1. **Inicialización y Verificación (Echo Test)**: tras la apertura del puerto COM y una espera inicial de 2 segundos para garantizar el arranque del PSOC, se ejecuta una prueba de eco (Echo Test). Esta fase es crítica para validar la integridad física del canal y la sincronización de la configuración UART. Si esta validación falla o no se recibe respuesta en la ventana de tiempo estipulada, el servidor aborta la operación inmediatamente cerrando el puerto.

2. **Negociación de Roles**: superado el test inicial, el servidor determina arbitrariamente qué rol desempeñará (Alice o Bob) y comunica al PSOC el rol complementario. Como se observa en la figura, esto bifurca la ejecución en dos escenarios simétricos. Es relevante notar la diferencia en la gestión del tiempo:
   - **En el PSOC:** se utilizan esperas bloqueantes para la recepción de datos, aprovechando que el dispositivo dedica sus recursos exclusivamente a esta tarea.
   - **En el Servidor:** se implementan timeouts conservadores (20s y 25s) para la recepción de la clave pública y el texto cifrado respectivamente. Estos márgenes amplios son necesarios no solo para cubrir el tiempo de cómputo criptográfico del microcontrolador, si no también para compensar la latencia introducida por el buffer del puerto serie y la planificación de procesos del sistema operativo del PC.
3. **Verificación del Secreto y Limitaciones**: finalmente, para confirmar el éxito del protocolo, ambos extremos intercambian el secreto compartido obtenido. Cabe destacar dos consideraciones técnicas sobre esta etapa:
   a) **Gestión del Buffer:** se identificó un cuello de botella en la transmisión Servidor → PSoC. Para evitar el desbordamiento del buffer de recepción del microcontrolador y la consecuente pérdida de datos, se fragmenta el envío en paquetes pequeños, lo que justifica parte de la latencia observada.
   b) **Seguridad de la Validación:** el envío del secreto compartido en texto claro por el canal serie se realiza estrictamente con fines de depuración y validación académica. Tal como se discutió en la introducción (ver sección 3.1), en un entorno de producción esto constituiría una vulnerabilidad crítica. La verificación de integridad debería realizarse mediante un Código de Autenticación de Mensaje Basado en Hash (HMAC), sin revelar jamás el secreto original.

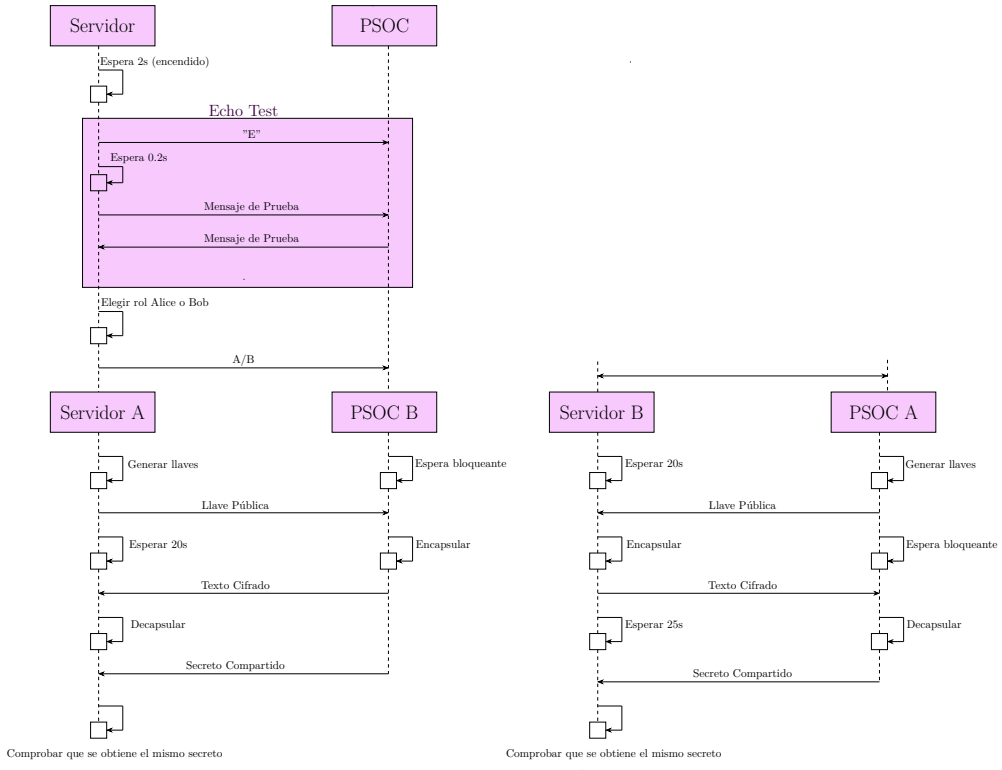

Figura 4.3: Diagrama de secuencia en UML que representa un ejemplo de intercambio de claves entre un Servidor y un PSOC.

4.7. Tests de rendimiento realizados

basado en la metodología en [62]

4.7.1. Ciclos de CPU

4.7.2. Operaciones por segundo

4.7.3. Uso de Pila

4.7.4. Coste de Comunicación

4.7.5. Suite de aleatoriedad a randombytes

limitaciones ent y funcionamiento [63] , dieharder [64], Do1 [65] y nist [66]

Capítulo 5

Resultados y discusión

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

5.1. Test de respuesta conocida

Este test se realiza para comprobar que la compilación e integración de los algoritmos funciona matemáticamente igual que las implementaciones de referencia presentadas al NIST. Para ello, el procedimiento estándar en un *Known Answer Tests* consiste en generar un fichero de respuesta (`.rsp`) a partir de una semilla inicial fija con la cual mediante el generador `rng.h` pseudoaleatorio se derivan todas las claves y textos cifrados. Si la implementación es correcta el fichero ha de ser igual al proporcionado por los autores.

5.1.1. Kyber

La ejecución de los tests para Kyber se completó sin incidencias. Se procesaron los 100 vectores de prueba definidos por el NIST. Al utilizar la misma semilla de entrada, la implementación generó claves y criptogramas idénticos a la referencia.

5.1.2. Saber

5.1.3. HQC

5.2. CPU cicles

5.2.1. Comparacion por plataforma

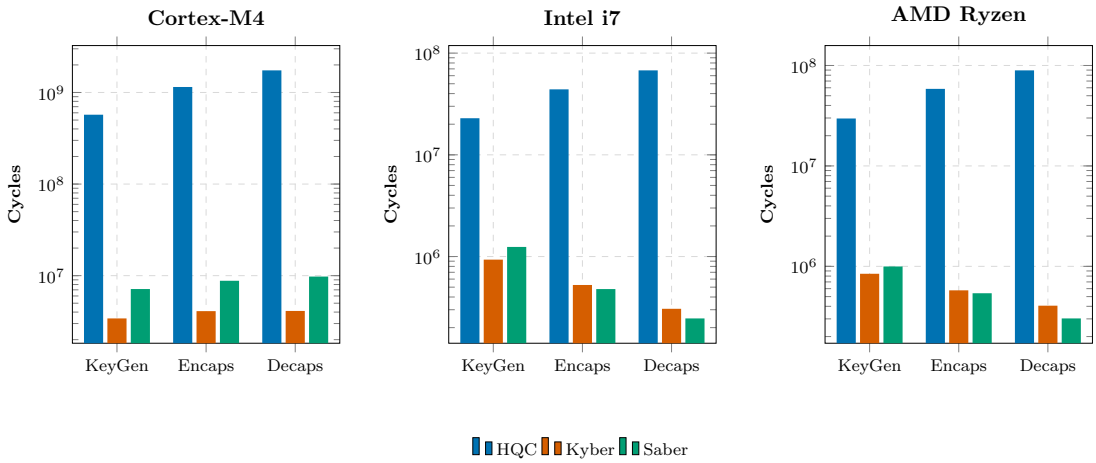

Figura 5.1:

5.2.2. Comparacion por algoritmo

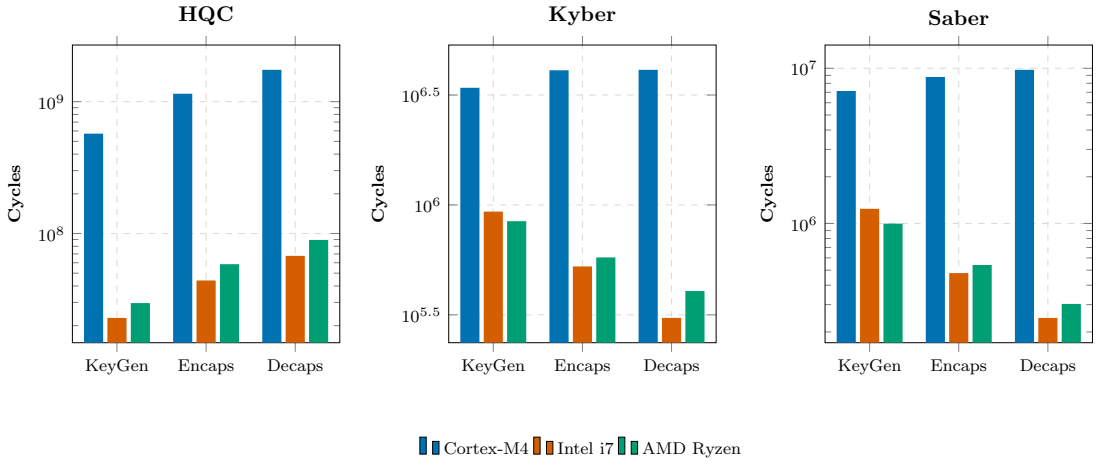

Figura 5.2:

5.3. Throughput

5.3.1. Comparacion por plataforma

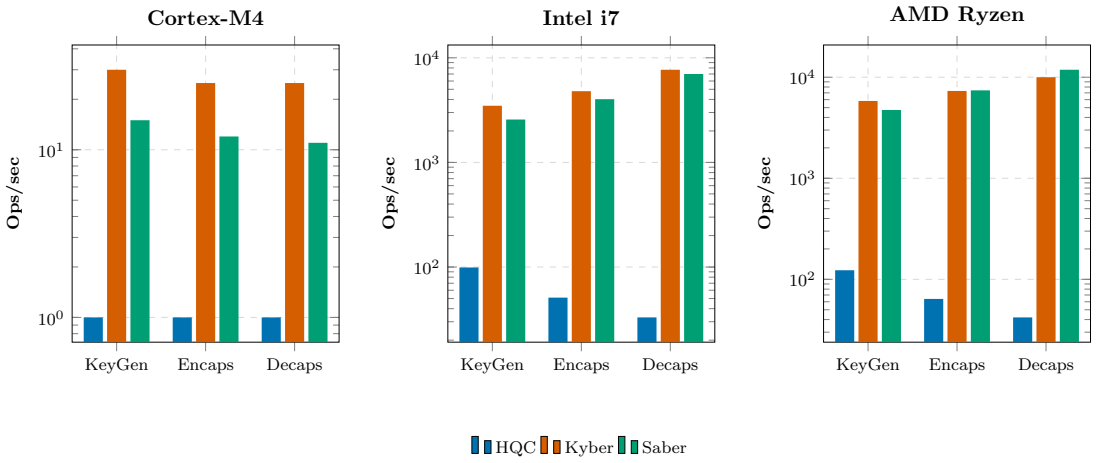

Figura 5.3:

5.3.2. Comparacion por algoritmo

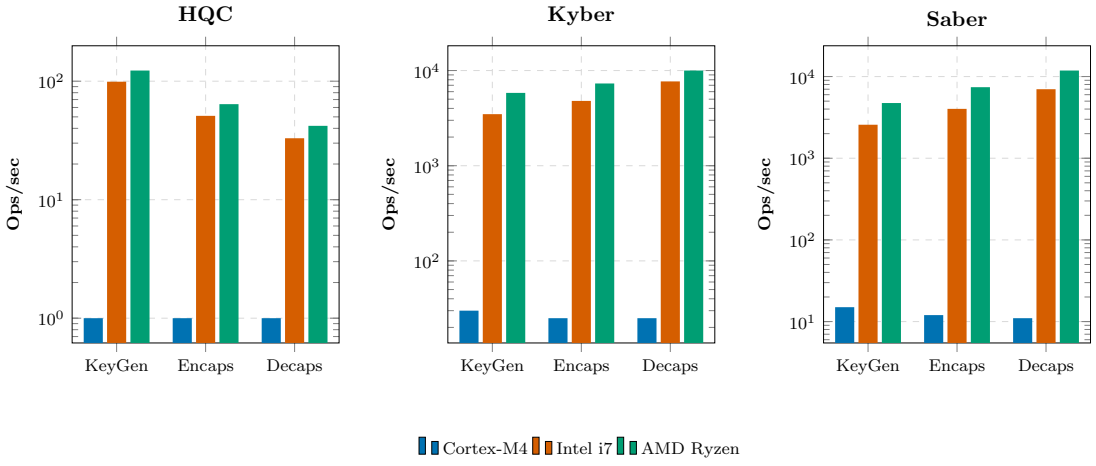

Figura 5.4:

5.4. Uso de pila

5.4.1. Comparacion por plataforma

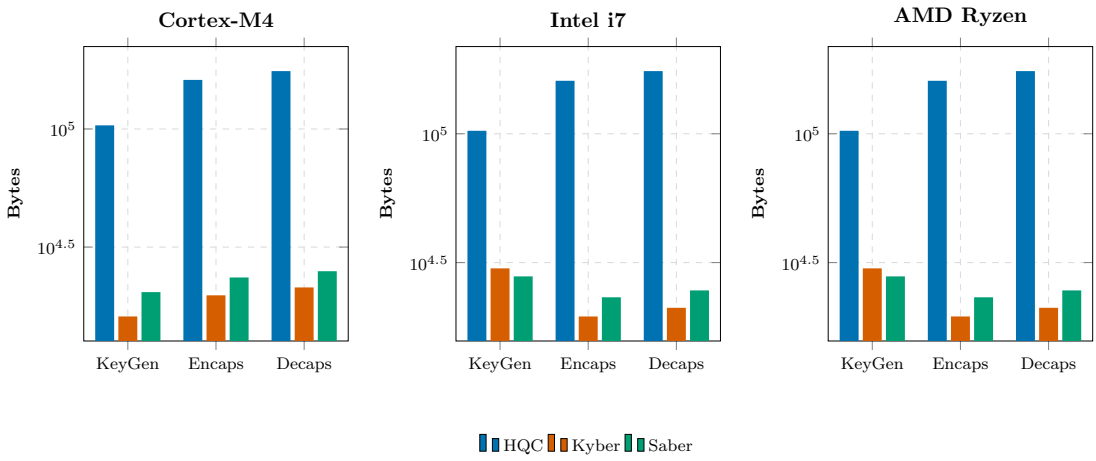

Figura 5.5:

5.4.2. Comparacion por algoritmo

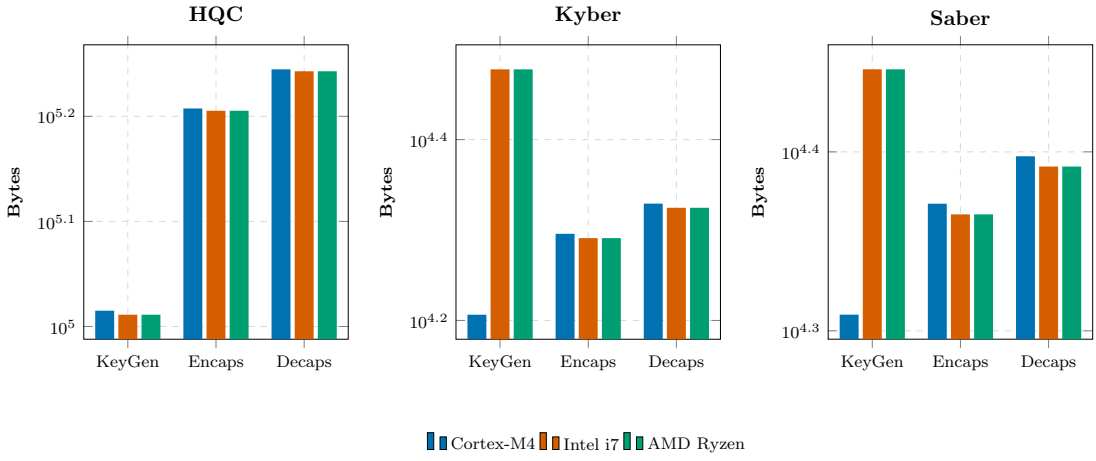

Figura 5.6:

5.5. Coste de comunicación y tiempo teórico de ejecución de los protocolos

5.6. Test de aleatoriedad

5.6.1. Plataforma Windows

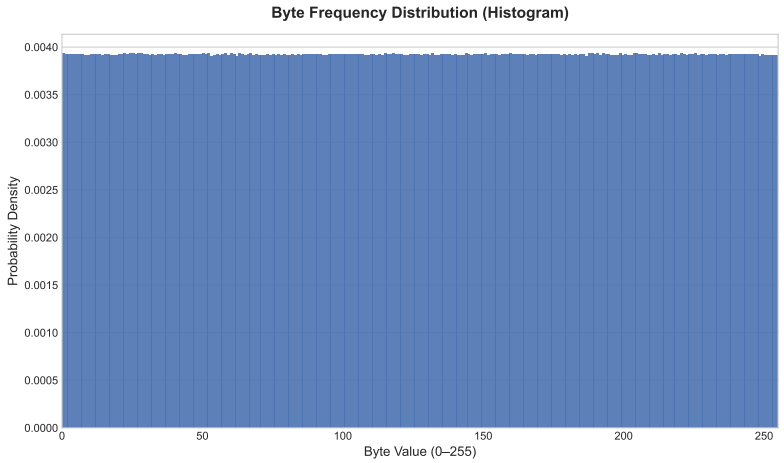

Figura 5.7:

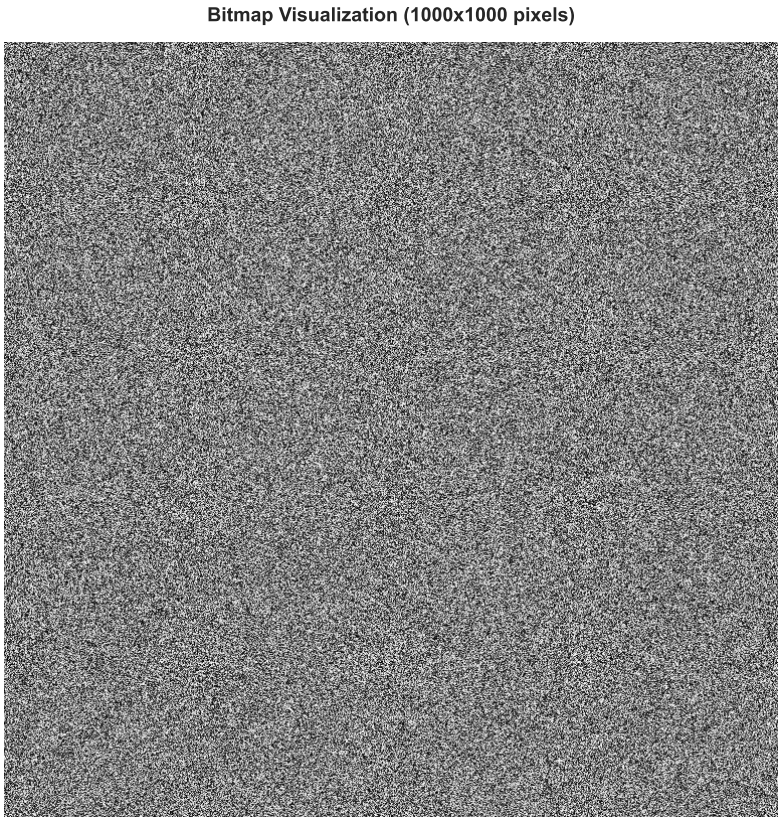

Figura 5.8:

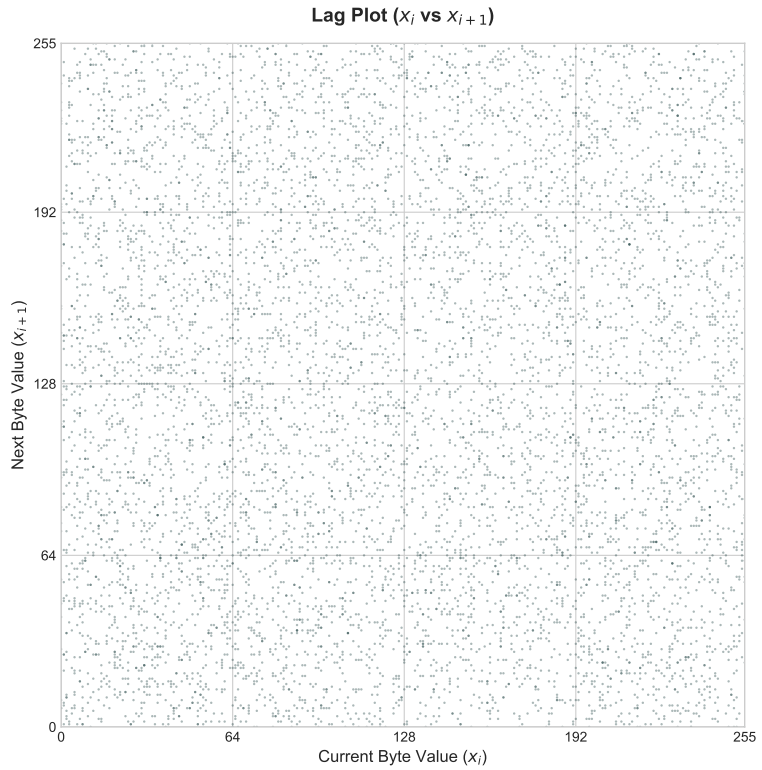

Figura 5.9:

5.6.2. PSOC

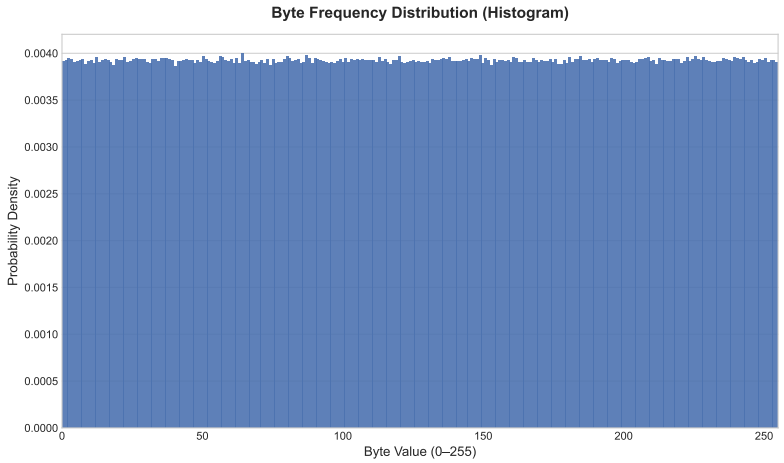

Figura 5.10:

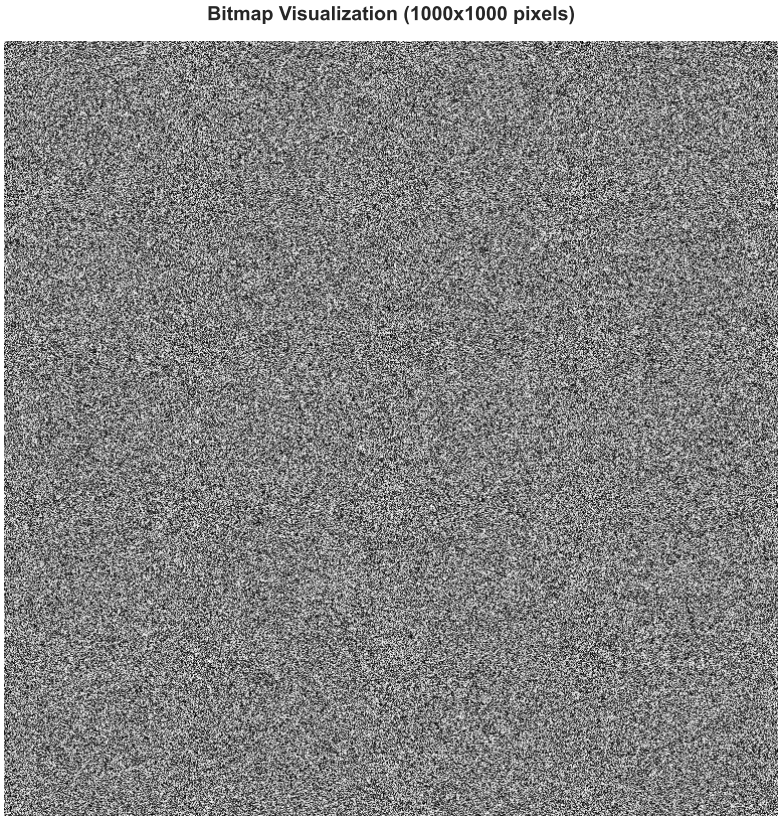

Figura 5.11:

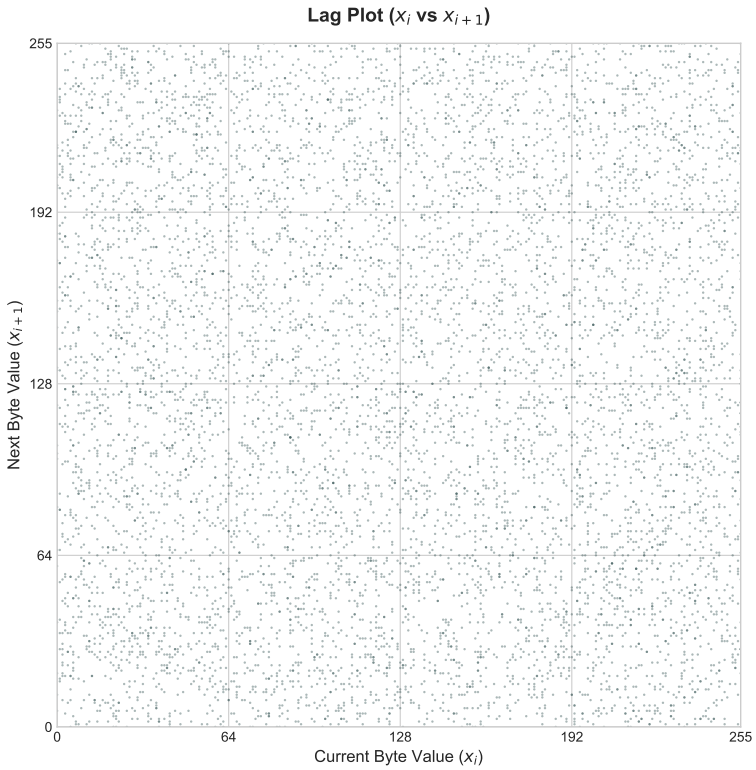
Lag Plot (x_i vs x_{i+1})
255
192
128
64
0
Next Byte Value (x_{i+1})
0
64
128
192
255
Current Byte Value (x_i)

Figura 5.12:

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

Pseudocódigo de los algoritmos postcuánticos

A.1. Transformadas Keccak-p

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$(x+1 \bmod 5, z-1 \bmod$ w$)$. Para ello, sigue los siguientes pasos:

Algoritmo 12 Transformada θ en Keccak-p

Entrada: A

Salida: A'

1: Calcular la paridad de cada columna, C:

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

2: Combinar la paridad de ambas columnas, D:

$$D(x, z) := C(x - 1 \text{ mód } 5, z) \oplus C(x + 1 \text{ mód } 5, z - 1 \text{ mód } w) \tag{A.2}$$

3: Realizar la XOR con el estado:

$$A'(x, y, z) := A(x, y, z) \oplus D(x, z) \tag{A.3}$$

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 A.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 A.1: Offsets de la transformada ρ.

Para realizar estas rotaciones sigue los siguientes pasos:

Algoritmo 13 Transformada ρ en Keccak-p

Entrada: A

Salida: A'

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

$$A'(0, 0, z) := A(0, 0, z) \tag{A.4}$$

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

3: Asignar el valor de la tabla A.1 modulo w a cada punto:

$$A'(x, y, z) := A(x, y, [z - (t + 1)(t + 2)/2 \bmod \mathrm{w}]) \tag{A.5}$$

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 A.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 A.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 14 Transformada π en Keccak-p

Entrada: A

Salida: A'

1: Calcular la rotación:

$$A'(x, y, z) := A(x + 3y \bmod 5, x, z) \tag{A.6}$$

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 A.1.

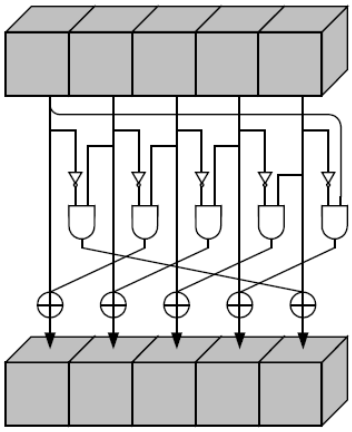

Figura A.1: Representación de la transformada χ realizada en cada fila [4]. Arriba la matriz $A(x, y, z)$ y abajo la matriz $A'(x, y, z)$.

Para realizar esta rotación se sigue el siguiente algoritmo:

Algoritmo 15 Transformada χ en Keccak-p

Entrada: A

Salida: A'

1: Calcular la rotación:

$$A'(x, y, z) := A(x, y, z) \oplus \{[A(x + 1 \bmod 5, y, z) \oplus 1] \& A(x + 2 \bmod 5, y, z)\} \tag{A.7}$$

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 16 Transformada ι en Keccak-p

Entrada: A, i_r

Salida: A'

1: Asignar:

$$A'(x, y, z) := A(x, y, z) \tag{A.8}$$

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

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

4: Asignar

$$RC(2^j - 1) := \texttt{rc}(j + 7i_r) \tag{A.9}$$

5: **end for**

6: Asignar:

$$A'(0, 0, z) := A'(0, 0, z) \oplus RC(z) \tag{A.10}$$

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 17 rc

Entrada: t

Salida: $rc(t)$

if t mod $255 = 0$ **then**
    return 1
end if
for i=1:t mod 255 **do**
    $R := R||R$
    $R[0] := R[0] \oplus R[8]$
    $R[4] := R[4] \oplus R[8]$
    $R[5] := R[5] \oplus R[8]$
    $R[6] := R[6] \oplus R[8]$
    $R := \texttt{Trunc}_8[R]$
end for
return $R[0]$

A.2. 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 3.2.

- 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 18 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:

$$(\rho, \sigma) := \text{SHA3-512}(d) \tag{A.11}$$

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:

$$\begin{aligned} \hat{s} &:= \text{NTT}(s) \\ \hat{e} &:= \text{NTT}(e) \\ \hat{t} &:= \hat{A} \circ \hat{s} + \hat{e} \end{aligned} \tag{A.12}$$

18: Se calcula la llave pública:

$$pk := \text{Encode}_{12}(\hat{t} \bmod^{+} \text{q})||\rho \tag{A.13}$$

▷ 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:

$$sk := \text{Encode}_{12}(\hat{s} \bmod^{+} \text{q})||pk||\text{SHA3-256}(pk)||z \tag{A.14}$$

▷ 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 19 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:

$$\begin{aligned}
&\hat{r} := \text{NTT}(r)\\
&u := \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1\\
&v := \text{NTT}^{-1}(\hat{t}^T \circ \hat{r}) + e_2 + \text{Decompress}_q[\text{Decode}_1(m), 1]\\
&c_1 := \text{Encode}_{d_u}[\text{Compress}_q(u, d_u)]\\
&c_2 := \text{Encode}_{d_v}[\text{Compress}_q(v, d_v)]
\end{aligned} \tag{A.15}$$

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 20 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:

$$\begin{aligned}
&\hat{t} := \text{Decode}_{12}(pk)\\
&p := pk + 12 \cdot k \cdot n/8
\end{aligned} \tag{A.16}$$

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:

$$\begin{aligned}
&m := \text{SHA3-256}(m')\\
&(\kappa, \gamma) := \text{SHA3-512}[m||\text{SHA3-256}(pk)]
\end{aligned} \tag{A.17}$$

5: Obtener el texto cifrado:

$$c \leftarrow \text{Cifrado Kyber}(pk, m, \gamma) \tag{A.18}$$

6: Calcular el secreto compartido:

$$k := \text{SHAKE-256}[\kappa||\text{SHA3-256}(c)] \tag{A.19}$$

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 21 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:

$$\begin{aligned} u &:= \text{Decompress}_q[\text{Decode}_{d_u}(c, d_u)] \\ v &:= \text{Decompress}_q[\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8, d_v)] \\ \hat{s} &:= \text{Decode}_{12}(sk) \end{aligned} \tag{A.20}$$

2: Obtener el mensaje m' cifrado anteriormente:

$$m' := \text{Encode}_1[\text{Compress}_q\left(v - \text{NTT}^{-1}(\hat{s}^T \circ \text{NTT}(u)), 1\right)] \tag{A.21}$$

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':

$$\begin{aligned} pk &:= sk + 12 \cdot k \cdot n/8 \\ h &:= sk + 24 \cdot k \cdot n/8 + 32 \\ (\kappa, \gamma) &:= \text{SHA3-512}[m'||h] \\ c' &\leftarrow \text{Cifrado Kyber}(pk, m', \gamma) \end{aligned} \tag{A.22}$$

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

$$z := sk + 24 \cdot k \cdot n/8 + 64 \tag{A.23}$$

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**

A.3. 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 3.3. 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 3.6.
- 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 3.4.2.2.

Algoritmo 22 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:

$$seed_A := \texttt{SHAKE-128}(seed_A, 32) \tag{A.24}$$

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

$$buf := \texttt{SHAKE-128}(seed_A, l^2 \cdot n \cdot \varepsilon_q/8) \tag{A.25}$$

▷ 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:

$$buf := \texttt{SHAKE-128}(seed_s, l \cdot n \cdot \mu/8) \tag{A.26}$$

▷ 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:

$$b := \left(A^T \circ s + h \bmod^{+} \mathrm{q}\right) / 2^{\varepsilon_q - \varepsilon_p} \tag{A.27}$$

22: se calcula la llave pública

$$pk := seed_A || \texttt{POLVEC}_q\texttt{2BS}(b) \tag{A.28}$$

23: Se calcula la llave secreta

$$sk := z || \texttt{SHA3-256}(pk) || pk || \texttt{POLVEC}_q\texttt{2BS}(s) \tag{A.29}$$

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 23 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':

$$b' := \left(A \circ s' + h \bmod^{+} \mathrm{q}\right) / 2^{\varepsilon_q - \varepsilon_p} \tag{A.30}$$

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

$$b := \texttt{BS2POLVEC}_p(pk') \tag{A.31}$$

5: se calcula el parámetro auxiliar v':

$$v' := b^T \circ (s' \bmod^{+} \mathrm{p}) \bmod^{+} \mathrm{p} \tag{A.32}$$

6: Se calcula el texto cifrado c

$$c := (v' - m \cdot 2^{\varepsilon_p - 1} + h_1 \bmod^{+} \mathrm{p})/2^{\varepsilon_p - \varepsilon_t} \tag{A.33}$$

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 24 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:

$$\begin{aligned} m &:= \texttt{SHA3-256}(m) \\ hash_{pk} &:= \texttt{SHA3-256}(pk) \\ buf &:= hash_{pk}||m \end{aligned} \tag{A.34}$$

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

$$(\gamma||r) := \texttt{SHA3-512}(buf) \tag{A.35}$$

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

$$c := \texttt{Cifrado Saber}(pk, m, \gamma) \tag{A.36}$$

5: Se calcula el secreto compartido k:

$$k := \texttt{SHA3-256}(r||\texttt{SHA3-256}(c)) \tag{A.37}$$

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 25 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

$$s := \texttt{BS2POLVEC}_q(sk') \tag{A.38}$$

3: se descompone el texto cifrado en sus partes

$$(c_m||ct) := c \tag{A.39}$$

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

$$\begin{aligned} &c_m := c_m \cdot 2^{\varepsilon_p-\varepsilon_t} \\ &b' := \texttt{BS2POLVEC}_p(ct) \\ &m' := (b'^T \circ (s \bmod^+ \mathrm{p}) - c_m + h_2 \bmod^+ \mathrm{p})/2^{\varepsilon_p-1} \end{aligned} \tag{A.40}$$

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

$$m := \texttt{POL}_2\texttt{2BS}(m') \tag{A.41}$$

6: Se calculan las variables γ y r:

$$(\gamma||r) := \texttt{SHA3-512}(buf) \tag{A.42}$$

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

$$c' := \texttt{Cifrado Saber}(pk, m, \gamma) \tag{A.43}$$

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

9: Asignar k:

$$k := \texttt{SHA3-256}(r||\texttt{SHA3-256}(c')) \tag{A.44}$$

10: **else**

11: Asignar k:

$$k := \texttt{SHA3-256}(z||\texttt{SHA3-256}(c')) \tag{A.45}$$

12: **end if**

13: **return** k

A.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 3.4.

- La función `SHAKE256.absorb` implementa el mecanismo de absorción o inicialización de la función como se puede ver en la figura 3.4. 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 3.4. 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 26 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

$$ctx := \texttt{SHAKE256.absorb}(seed) \tag{A.46}$$

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

$$(seed||\sigma) := \texttt{SHAKE256.absorb}(ctx, (32, 256)) \tag{A.47}$$

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

$$(seed_{sk}||seed_{pk}) := \texttt{SHA3-512}(seed||\text{I\_SEPARATOR}) \tag{A.48}$$

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

$$\begin{aligned} ctx_{pk} &:= \texttt{SHAKE256.absorb}(seed_{pk}) \\ ctx_{sk} &:= \texttt{SHAKE256.absorb}(seed_{sk}) \end{aligned} \tag{A.49}$$

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

$$(x||y) := \texttt{SampleFixedWeightVector}_{\$}(ctx_{sk}, \omega) \tag{A.50}$$

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

$$h := \texttt{SampleVect}(ctx_{pk}) \tag{A.51}$$

8: Se calcula el valor s:

$$s := x + h \cdot y \tag{A.52}$$

9: Se calcula la llave pública:

$$pk := seed_{pk}||s \tag{A.53}$$

10: Se calcula la llave privada:

$$sk := pk||seed_{sk}||\sigma||seed \tag{A.54}$$

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 `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 27 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

$$ctx_{pk} := \texttt{SHAKE256.absorb}(seed_{pk}) \tag{A.55}$$

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

$$h := \texttt{SampleVect}(ctx) \tag{A.56}$$

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

$$ctx_\theta := \texttt{SHAKE256.absorb}(\gamma) \tag{A.57}$$

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

$$(r_2||e||r_1) := \texttt{SampleFixedWeightVector}(ctx_\theta) \tag{A.58}$$

6: Se calcula el valor de u:

$$u := r_1 + h \cdot r_2 \tag{A.59}$$

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

$$v := \texttt{Encode}(m) + \texttt{Truncate}(s \cdot r_2 + e, l) \tag{A.60}$$

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 28 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:

$$(k||\gamma) := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{A.61}$$

3: Calcular el texto cifrado:

$$c := \texttt{Cifrado Saber}(pk, m, \gamma)||salt \tag{A.62}$$

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 29 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:

$$ctx := \texttt{SHAKE256.absorb}(seed_{sk}) \tag{A.63}$$

3: Se muestrea el vector y:

$$y := \texttt{SampleFixedWeightVector}_{\$}(ctx_{sk}, \omega) \tag{A.64}$$

4: Se computa el mensaje:

$$m' := \texttt{Decode}(v - \texttt{Truncate}(u \cdot y, l)) \tag{A.65}$$

5: Se calcula el secreto compartido:

$$(k'||\gamma') := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{A.66}$$

6: Calcular el texto cifrado:

$$c' := \texttt{Cifrado Saber}(pk, m', \gamma')||salt \tag{A.67}$$

▷ 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:

$$(k||-) := \texttt{SHA3-512}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||m||salt||\text{G\_SEPARATOR}) \tag{A.68}$$

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

$$k := \texttt{SHA3-256}(\texttt{SHA3-256}(pk||\text{H\_SEPARATOR})||\sigma||c||\text{J\_SEPARATOR}) \tag{A.69}$$

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

Apéndice B

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:

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

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.

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

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:

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

Con estos valores se calculan los textos cifrados:

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

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

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

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 $q/4 = 4{,}25$.

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

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

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

Apéndice C

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 D

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:

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

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

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

Codificando mediante Reed-Solomon:

1. Obtener $y(x)$

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

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 $b(x)$:

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

3. El mensaje codificado c quedaría como:

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

Ahora mediante Reed-Muller se codifica el mensaje c:

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

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

- 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 D.1.

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

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

$$v = 11001100101010101001100110101010010101011001100100110011 \tag{D.8}$$

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

$$v' = v + r = 11001000101010100110010010111010010111011001101100010011 \tag{D.9}$$

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

- Dividir el mensaje v' en bloques

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

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

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

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

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

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

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

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

- 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:

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

A continuación se decodifica con Reed-Solomon:

- Se calcula el número de síndromes posibles 2δ:

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

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

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

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

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

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

- Se obtienen las raíz β^{-1} de σ

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

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

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

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

- La magnitud del error:

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

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

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

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

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

Bibliografía

[1] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, D. Stehlé, *et al.*, "Crystals-kyber algorithm specifications and supporting documentation," *NIST PQC Round 3 submission*, January 2021.

[2] A. Basso, J. M. B. Mera, J.-P. D'Anvers, A. Karmakar, S. S. Roy, M. V. Beirendonck, and F. Vercauteren, "Saber: Mod-lwr based kem (round 3 submission)," *NIST PQC Round 3 submission*, September 2020.

[3] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "Hamming quasi-cyclic (hqc) specification," *NIST PQC Round 4 submission*, Aug. 2025.

[4] M. J. Dworkin *et al.*, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015.

[5] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Annual International Cryptology Conference*, pp. 26–45, Springer, 1998.

[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] G. Alagic, G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, *et al.*, "Status report on the third round of the nist post-quantum cryptography standardization process," tech. rep., 2022.

[8] G. Alagic, G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, *et al.*, "Status report on the third round of the nist post-quantum cryptography standardization process," tech. rep., 2022.

[9] Y. Zhang, Y. Bian, Z. Li, S. Yu, and H. Guo, "Continuous-variable quantum key distribution system: Past, present, and future," *Applied Physics Reviews*, vol. 11, no. 1, 2024.

[10] M. Kumar and B. Mondal, "A brief review on quantum key distribution protocols," *Multimedia Tools and Applications*, pp. 1–40, 2025.

[11] A. Atadoga, O. A. Farayola, B. S. Ayinla, O. O. Amoo, T. O. Abrahams, and F. Osasona, "A comparative review of data encryption methods in the usa and europe," *Computer Science & IT Research Journal*, vol. 5, no. 2, pp. 447–460, 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)*, pp. 616–622, IEEE, 2021.

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

[14] D. Heinz, M. J. Kannwischer, G. Land, T. Pöppelmann, P. Schwabe, and A. Sprenkels, "First-order masked kyber on arm cortex-m4," *Cryptology ePrint Archive*, 2022.

[15] M. Anastasova, P. Kampanakis, and J. Massimo, "Pq-hpke: post-quantum hybrid public key encryption," *Cryptology ePrint Archive*, 2022.

[16] C. Ugwuishiwu, U. Orji, C. Ugwu, and C. Asogwa, "An overview of quantum cryptography and shor?s algorithm," *Int. J. Adv. Trends Comput. Sci. Eng*, vol. 9, no. 5, 2020.

[17] D. Aasen, M. Aghaee, Z. Alam, M. Andrzejczuk, A. Antipov, M. Astafev, L. Avilovas, A. Barzegar, B. Bauer, J. Becker, *et al.*, "Roadmap to fault tolerant quantum computation using topological qubit arrays," 2025.

[18] F. Samiullah, M.-L. Gan, S. Akleylek, and Y. Aun, "Quantum resistance saber-based group key exchange protocol for iots," *IEEE Open Journal of the Communications Society*, 2024.

[19] "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, Aug. 2024.

[20] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[21] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," *arXiv preprint quant-ph/0301141*, 2003.

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

[23] V. Weger, N. Gassner, and J. Rosenthal, "A survey on code-based cryptography," *arXiv preprint arXiv:2201.07119*, 2022.

[24] G. Alagic, E. Barker, L. Chen, D. Moody, A. Robinson, H. Silberg, and N. Waller, "Recommendations for key-encapsulation mechanisms," tech. rep., National Institute of Standards and Technology, 2025.

[25] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "The keccak reference," tech. rep., NIST SHA-3 Competition, 2011. Round 3 submission.

[26] "Post-quantum cryptography - round 3 submissions," 2020. Consultado: 2025-05-05.

[27] "Post-quantum cryptography - round 4 submissions," 2022. Consultado: 2025-05-05.

[28] A. Satriawan, R. Mareta, and H. Lee, "A complete beginner guide to the number theoretic transform (ntt)," *Cryptology ePrint Archive*, 2024.

[29] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*. Cambridge University Press, revised ed., 1994. Ver Teorema 2.8, p g. 46.

[30] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[31] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of the ACM (JACM)*, vol. 60, no. 6, pp. 1–35, 2013.

[32] D. Micciancio and O. Regev, "Lattice-based cryptography," in *Post-Quantum Cryptography*, pp. 147–191, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[33] Y. Wang and M. Wang, "Module-lwe versus ring-lwe, revisited," *Cryptology ePrint Archive*, 2019.

[34] L. Ducas and J. Schanck, "Security estimation scripts for kyber and dilithium," 2023. Consultado: 2025-05-31.

[35] J.-P. D?Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," 2018.

[36] A. Karatsuba, "Multiplication of multidigit numbers on automata," vol. 7, pp. 595–596, 1963.

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

[38] 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*, pp. 17–24, 2007.

[39] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980.

[40] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the ring! practical, quantum-secure key exchange from lwe," 2016.

[41] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, and I. Bourges, "HQC: Hamming Quasi-Cyclic (Fourth Round Submission)," *NIST PQC Round 4 submission*, October 2022.

[42] P. Gaborit, "Shorter keys for code based cryptography," in *Proceedings of the 2005 International Workshop on Coding and Cryptography (WCC 2005)*, pp. 81–91, 2005.

[43] G. D. Forney, *Concatenated Codes*. Cambridge, MA: MIT Press, 1966.

[44] S. Lin and D. J. Costello, *Error control coding*, vol. 2. Prentice hall Scarborough, 2001.

[45] D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry, *Random oracles in a quantum world*, pp. 41–69. 2011.

[46] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge: Cambridge University Press, 2010.

[47] T. Yamakawa and M. Zhandry, "Classical vs quantum random oracles," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 568–597, Springer, 2021.

[48] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Annual international cryptology conference*, pp. 537–554, Springer, 1999.

[49] P. Kirchner and P.-A. Fouque, "An improved bkw algorithm for lwe with applications to cryptography and lattices," in *Annual Cryptology Conference*, pp. 43–62, Springer, 2015.

[50] S. Arora and R. Ge, "New algorithms for learning in presence of errors," in *International Colloquium on Automata, Languages, and Programming*, pp. 403–415, Springer, 2011.

[51] C.-P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Mathematical programming*, vol. 66, no. 1, pp. 181–199, 1994.

[52] E. C.-Y. Peng and M. G. Kuhn, "Adaptive template attacks on the kyber binomial sampler," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, pp. 470–492, June 2025.

[53] P. Pessl and R. Primas, "More practical single-trace attacks on the number theoretic transform," 2019.

[54] I. Dumer, "On minimum distance decoding of linear codes," in *Proc. 5th Joint Soviet-Swedish Int. Workshop Inform. Theory*, pp. 50–52, Moscow, 1991.

[55] N. Sendrier, "Decoding one out of many," in *International Workshop on Post-Quantum Cryptography*, pp. 51–67, Springer, 2011.

[56] G. Goy, A. Loiseau, and P. Gaborit, "A new key recovery side-channel attack on hqc with chosen ciphertext," in *International Conference on Post-Quantum Cryptography*, pp. 353–371, Springer, 2022.

[57] G. Goy, M. Spyropoulos, N. Aragon, P. Gaborit, R. Pacalet, F. Perion, L. Sauvage, and D. Vigilant, "Side-channel sensitivity analysis on hqc: Towards a fully masked implementation," *Cryptology ePrint Archive*, 2025.

[58] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "PQClean: Clean, portable, tested implementations of post-quantum cryptography," 2025.

[59] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, (Los Alamitos, CA, USA), pp. 19–30, IEEE Computer Society, 2022.

[60] D. Sprenkels, "randombytes: A portable c library for generating crypto-secure random bytes." `https://github.com/dsprenkels/randombytes`, 2017.

[61] Infineon Technologies, *PSoC? 6 MCU: CY8C62x6, CY8C62x7 Architecture Technical Reference Manual*. Infineon Technologies AG.

[62] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4." Cryptology ePrint Archive, Paper 2019/844, 2019.

[63] E. Almaraz Luengo, B. Ala a Olivares, L. J. Garc a Villalba, J. Hernandez-Castro, and D. Hurley-Smith, "StringENT test suite: ENT battery revisited for efficient P value computation," *Journal of Cryptographic Engineering*, vol. 13, pp. 235–249, jun 2023.

[64] R. G. Brown, *DieHarder: A Gnu Public License Random Number Tester*. Duke University Physics Department, Durham, NC, 2006. Version 3.31.2beta.

[65] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, Aug. 2007.

[66] L. Bassham, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, S. Leigh, M. Levenson, M. Vangel, N. Heckert, and D. Banks, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010-09-16 2010.