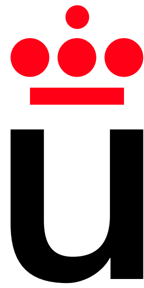

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso 2023-2024

Trabajo Fin de Grado

ESTUDIO Y COMPARATIVA DE LOS PRINCIPALES ALGORITMOS DE CRIPTOGRAFÍA POST-CUÁNTICA

Autor: Daniel Alfonsel García
Tutor: Ana Isabel Gómez Pérez

Agradecimientos

En agradecimiento a mi familia, por apoyarme en todo momento; y a mis profesores del instituto, por guiarme cuando no encontraba el camino.

Especial mención a mis profesoras de física, por dotarme de la curiosidad para intentar entender como funciona el mundo de lo grande y de lo pequeño.

Resumen

Durante el desarrollo del Trabajo de Fin de Grado se echa un vistazo al estado de la criptografía actual: criptografía simétrica y asimétrica dando énfasis al esquema criptográfico RSA en la parte de asimétrica; explicando los problemas que plantea; y conceptos sobre la seguridad de la misma como son los modelos de ataque, metas de seguridad, y la relación entre ellos. También se explica el método de encapsulamiento de claves criptográficas, su procedencia y por qué el National Institute of Standards and Technology (NIST) pide que los candidatos cumplan con esta estructura para la propuesta de nuevos métodos de criptografía post-cuántica. Tras una breve introducción a la computación cuántica, se introduce el Algoritmo de Shor, tanto en su forma convencional como en su forma cuántica; un algoritmo capaz de poner en jaque nuestros modelos de seguridad, dándonos la capacidad de encontrar los factores de un número en un tiempo de computación asequible. También se echa un vistazo a los paradigmas que se están aplicando en los sistemas de cifrado que prometen ser resistentes a esta amenaza: criptografía basada en retículos, códigos o «hashes», y criptografía multivariante, explicándolos de forma breve para su entendimiento.

Una vez pasada esta puesta en contexto, se explica la convocatoria de estandarización del NIST sobre algoritmos post-cuánticos; presentando a los candidatos de encapsulamiento (Kyber, Classic McEliece o Saber, entre otros) y de firma digital (Dillithium, Falcon, SPHINCS y Rainbow). Una vez presentados, se estudia su rendimiento, comparándolos entre sí, para al final arrojar unos resultados y unas conclusiones, reflexionando en el camino sobre por qué pueden estar producidos esos resultados.

Palabras clave:

- Ciberseguridad
- Criptografía Post-cuántica
- Algoritmo de Shor
- KEM
- Estandarización NIST

Índice de contenidos

Índice de tablas X

Índice de figuras XII

1. Introducción 1

1.1. Contexto y Alcance 1

2. Contenidos principales 3

2.1. Criptografía tradicional 3

2.1.1. Criptografía simétrica 4

2.1.2. Seguridad criptográfica 6

2.1.3. Criptografía asimétrica 8

2.2. Criptografía post-cuántica 16

2.2.1. Computación cuántica y criptografía clásica 17

2.2.2. Sistemas de cifrado 20

2.3. Concurso de estandarización del NIST 24

2.3.1. Niveles de seguridad del NIST 24

2.3.2. Algoritmos candidatos 25

3. Ejecución y resultados 27

3.1. Ejecución y código 27

3.1.1. Librerías utilizadas 27

3.1.2. Código 30

3.1.3. Algoritmos KEM 31

3.1.4. Algoritmos de Firma digital 31

3.2. Resultados 31

3.2.1. Algoritmos KEM 31

3.2.2. Algoritmos de firma digital 49

4. Conclusiones y trabajos futuros 57

Bibliografía 58

Apéndices 63

A. Tablas de algoritmos **65**

B. figuras y tablas de SPHINCS$^+$ **69**

C. Código **73**

D. Licencia **85**

Índice de tablas

2.1. Codificación de los dígitos 0-9 en UTF-8 5
2.2. Tabla de tiempo de factorización en función de n 14
2.3. Tabla de valores función f(x) en algoritmo de Shor 18

3.1. Características de algoritmos KEM de nivel 1 32
3.2. Media de ciclos de ejecución algoritmos KEM de nivel 1. 33
3.3. Características de algoritmos KEM de nivel 3. 35
3.4. Coeficientes de crecimiento (cociente entre el nivel 3 y nivel 1 KEM: características. 36
3.5. Media de ciclos de ejecución algoritmos KEM de nivel 3. 36
3.6. Coeficientes de crecimiento en media de ciclos nivel 1 - nivel 3 KEM: Cociente entre el nivel 3 y el nivel 1. 37
3.7. Características de algoritmos KEM de nivel 5. 38
3.8. Coeficientes de crecimiento nivel 3 - nivel 5 KEM: características. Cociente entre nivel 5 y nivel 3 39
3.9. Media de ciclos de ejecución algoritmos KEM de nivel 5. 40
3.10. Coeficientes de crecimiento nivel 3 - nivel 5 KEM: cociente entre media de ciclos nivel 5 y nivel 3. 41
3.11. Tamaños en bytes del algoritmo Classic McEliece. 43
3.12. Media de ciclos algoritmo Classic McEliece. 44
3.13. Tamaño en bytes de algoritmos de firma de nivel 1 50
3.14. Media de ciclos de algoritmos de firma de nivel 1 50
3.15. Tamaño en bytes de firma de nivel 3 51
3.16. Media de ciclos de algoritmos de firma de nivel 3 52
3.17. Características de algoritmos de firma de nivel 5 53
3.18. Media de ciclos de algoritmos de firma de nivel 5 54
3.19. Tamaño en bytes de algoritmo de firma SPHINCS$^+$ 55

A.1. Tabla de algoritmos KEM 66
A.2. Tabla de algoritmos de firma digital 1 67
A.3. Tabla de algoritmos de firma digital 2 68

B.1. Media de ciclos de algoritmos de SPHINCS$^+$-128-Haraka 69
B.2. Media de ciclos de algoritmos de SPHINCS$^+$-192-Haraka 69

B.3. Media de ciclos de algoritmos de SPHINCS$^+$-256-Haraka 69
B.4. Media de ciclos de algoritmos SPHINCS$^+$-SHA265/SHAKE256 . . 70

Índice de figuras

2.1. Funciones de cifrado y descifrado 5
2.2. Funcionamiento de PKE. 11
2.3. Firma RSA. 16
2.4. Retículo en $\mathbb{R}^2$. 21
2.5. Arbol de Merkle 23

3.1. Gráfico características de algoritmos KEM de nivel 3 32
3.2. Gráfico características de algoritmos KEM de nivel 1 33
3.3. Gráfico características de algoritmos KEM de nivel 3. 34
3.4. Gráfico características de algoritmos KEM de nivel 1 (color oscuro) vs nivel 3 (color claro). 35
3.5. Gráfico media de ciclos de ejecución algoritmos KEM de nivel 3. . . 36
3.6. Gráfico media de ciclos de algoritmos KEM de nivel 1 (color oscuro) vs nivel 3 (color claro). 37
3.7. Tamaño en bytes para cada fase de los algoritmos KEM de nivel 5. 38
3.8. Tamaño de clave de algoritmos KEM de nivel 3 (color oscuro) vs nivel 5 (color claro) 39
3.9. Gráfico media de ciclos de algoritmos KEM de nivel 5. 40
3.10. Gráfico media de ciclos de algoritmos KEM de nivel 3 (color claro) vs nivel 5 (color oscuro) 41
3.11. Tamaño en bytes de las etapas del algoritmo Classic McEliece. . . 42
3.12. Gráfico media de ciclos generación algoritmo Classic McEliece. . . 43
3.13. Gráfico media de ciclos algoritmo Classic McEliece. 44
3.14. Gráfico consistencia algoritmo BIKE. 45
3.15. Gráfico consistencia algoritmo FrodoKEM-AES. 46
3.16. Gráfico consistencia algoritmo FrodoKEM-SHAKE. 46
3.17. Gráfico consistencia algoritmo HQC. 47
3.18. Gráfico consistencia algoritmo Kyber. 47
3.19. Gráfico consistencia algoritmo Kyber-90s. 48
3.20. Gráfico consistencia algoritmo Classic McEliece. 48
3.21. Tamaño en bytes de algoritmos de firma de nivel 1 49
3.22. Gráfico media de ciclos de algoritmos de firma de nivel 1 50
3.23. Tamaño en bytes de algoritmos de firma de nivel 3 51

3.24. Gráfico media de ciclos de algoritmos de firma de nivel 3 52
3.25. Gráfico características de algoritmos de firma de nivel 3 53
3.26. Gráfico media de ciclos de algoritmos de firma de nivel 5 54
3.27. Gráfico tamaño de firma de algoritmos de SPHINCS+ 55
3.28. Gráfico media de ciclos de algoritmos de SPHINCS+-192 56
3.29. Gráfico media de ciclos de algoritmos de SPHINCS+-256 56

B.1. Gráfico media de ciclos de algoritmos de SPHINCS+-128-Haraka . 71
B.2. Gráfico media de ciclos de algoritmos de SPHINCS+-192-Haraka . 71
B.3. Gráfico media de ciclos de algoritmos de SPHINCS+-256-Haraka . 72
B.4. Gráfico media de ciclos de algoritmos de SPHINCS+-128 72

1
Introducción

1.1. Contexto y Alcance

Aunque parezca algo actual, la criptografía ha estado presente en gran parte de la historia del ser humano. Ya en la época de los egipcios se encuentran "jeroglíficos no estándares", que se cree que tenían como objetivo dotar de un mayor dramatismo a la historia contada. Estos, son considerados los primeros ejemplos de "escritura oculta". También en La Biblia se hace referencia al Atbash, un sistema de sustitución utilizado para cifrar mensajes. De sobra conocido el Cifrado César, atribuido a Julio César, que utilizaba un desplazamiento de 3 letras. Durante numerosas épocas hay ejemplos de criptografía: Al-Kindi en la edad media, sentando las bases para el análisis de frecuencia; o los Estados Pontificios durante el renacimiento, cuna de la creación del Cifrado de Alberti, un sistema de codificación mecánico basado en discos.

No fue hasta La Segunda Guerra Mundial que la criptografía moderna comenzó a gestarse: gracias a la máquina Enigma se podían cifrar las comunicaciones de los Aliados. Tras La Segunda Guerra Mundial, Claude Shannon publicó ≪*A Communication Theory of Secrecy Systems*≫, lo que permitió transformar las técnicas de codificación en procesos matemáticos.

Tras todos estos avances y con la llegada de la computación se sucedieron otros esquemas de criptografía como por ejemplo DES, RSA, AES u otros que hoy en día que permiten transmitir de forma segura en las comunicaciones. Todos estos algoritmos parten de la misma premisa: un problema matemático relativamente fácil de resolver en un sentido, pero relativamente complicado de resolver a la

inversa.

Los avances en la capacidad de cómputo han obligado a reforzar todos estos métodos de seguridad incrementando el esfuerzo requerido para romper la seguridad, con el fin de que nuestras comunicaciones sean hoy en día todavía seguras. No obstante, el estudio de la computación cuántica ha avanzado lo suficiente como para empezar a plantearse los riesgos que puede suponer para nuestros cifrados. Este tipo de ordenadores son capaces de realizar operaciones a una velocidad abrumadora en comparación con nuestros ordenadores actuales: aprovechándose del principio de superposición cuántica, son capaces de realizar múltiples operaciones "al mismo tiempo".

El punto de este TFG es un repaso por la criptografía actual para luego explicar qué peligros entraña la computación cuántica respecto a la misma, viendo las distintas soluciones que se han propuesto en el concurso de estandarización NIST para algoritmos post-cuánticos, es decir, algoritmos resistentes a la cuántica.

Los objetivos del TFG son los siguientes:

- **Estado del arte.** Dar un repaso sobre el estado de la criptografía actual; estudiando la criptografía simétrica, la criptografía asimétrica, y los modelos de seguridad criptográfica que se utilizan hoy en día.
- **Introducción a la computación cuántica.** Dar una breve introducción a los fundamentos de la computación cuántica: el qubit, la notación Dirac, y las matemáticas subyacentes de forma muy simplificada.
- **Plantear los problemas que supone la computación cuántica para la criptografía actual.** Introducir un ejemplo práctico de cómo el algoritmo de Shor sería capaz de romper algoritmos actuales como RSA o basados en el mismo problema.
- **Explicar las soluciones planteadas.** El NIST realizó una convocatoria para estandarizar algoritmos resistentes a la cuántica. Explicar en qué se basan la mayoría de estos algoritmos, viendo los problemas matemáticos que plantean.
- **Análisis de los algoritmos presentados.** Estudiar el rendimiento computacional de los algoritmos planteados, comparando e intentando explicar el por qué los resultados obtenidos.

2

Contenidos principales

2.1. Criptografía tradicional

En esta sección se presentan los conceptos básicos por los que se rige la criptografía que utilizamos hoy en día. En concreto, se tratan la criptografía simétrica (Sección 2.1.1), introduciendo los sistemas de cifrado más comunes y las funciones hash; y los conceptos que modelan la seguridad criptográfica (Sección 2.1.2), explicando los modelos de ataque y las metas de seguridad definidas para clasificar los algoritmos de cifrado. Por otro lado, se introduce el problema de reparto de claves y la forma que se aborda en la actualidad, mediante la criptografía asimétrica (Sección 2.1.3), presentando los sistemas de cifrado más comunes, los mecanismos de encapsulado de clave, el algoritmo de *Rivest, Shamir y Adleman* y los problemas que se presenta, y los sistemas de firma digital.

Antes de hablar de los dos tipos distintos de criptografía, simétrica y asimétrica, es necesario introducir algunos conceptos. El primero de estos conceptos es el funcionamiento de la criptografía en nuestras comunicaciones, así como la forma en la que un atacante puede amenazar estas comunicaciones. Para ello, se suele plantear el siguiente escenario: una **emisora**, Alice, quiere enviarle un mensaje a Bob, **receptor**, que solo Bob sea capaz de leer. Para ello, Alice debe utilizar un **esquema criptográfico**. Estos esquemas están conformados por varios algoritmos matemáticos creados para generar un conjunto de **claves** con la capacidad de modificar un mensaje o unos datos en base a las claves generadas. Un sistema criptográfico suele estar compuesto por tres tipos de algoritmos: un algoritmo generador de clave(s), un algoritmo de cifrado y un algoritmo de descifrado.

Los sistemas criptográficos se crearon para alcanzar ciertas metas. Estas metas varían en función de a quién se le pregunte, pero suelen ser las siguientes:

- **Confidencialidad:** La información solo está disponible para usuarios autorizados.
- **Integridad:** La información no se puede manipular por agentes externos.
- **Autenticación:** Confirmación de la autenticidad de la información y/o de la identidad del usuario emisor.
- **No repudio:** Evitar la posibilidad de que un usuario deniegue acciones previas.

Para que un sistema criptográfico se considere bueno, debe cumplir el **principio de Kerchoff**. Este principio estipula que la seguridad de un sistema criptográfico debe recaer en la seguridad de la clave, no en la ocultación del funcionamiento del sistema.

Si Alice decidiera no usar un sistema criptográfico para cifrar la información, Eve podría intervenir en la comunicación, ya sea leyendo información que no va dirigida a ella o incluso modificándola, para que en el momento en el que llegue al receptor, está sea incorrecta. En esta situación, se denomina a Alice y a Bob como **usuarios legítimos**, mientras que a Eve se le denomina como usuario ilegitimo, usuario no autorizado o directamente **atacante**.

Hay múltiples criptosistemas (sistemas criptográficos) capaces de cifrar y descifrar. Para clasificarlos, se suele utilizar el número de claves que utilizan.

2.1.1. Criptografía simétrica

En el ámbito de la criptografía simétrica se da por supuesto que todos los integrantes de una comunicación comparten un fragmento de información secreta, denominado *clave simétrica*. Esta clave simétrica debe ser difícil de conocer por terceras partes, pero a su vez acordada de antemano de forma segura entre los participantes antes del establecimiento de la comunicación. El último requisito es el más complicado de cumplir y es el denominado *problema de reparto de claves*; Supondremos resuelto este problema y que los participantes han tenido acceso a un canal seguro para este propósito.

Como se ha mencionado antes, un criptosistema se compone de tres funciones. En el caso de la criptografía simétrica, la función de generación crea una clave. Esta clave servirá como entrada tanto para la función de cifrado como la de descifrado, es decir, **ambas operaciones utilizan la misma clave**. Siguiendo con la función de cifrado, esta toma el mensaje o plaintext como entrada a parte

de la clave generada anteriormente, dando como salida el texto cifrado o *ciphertext* Por último, la función de descifrado toma como entrada la misma clave generada anteriormente y el ciphertext, dando como salida el plaintext.

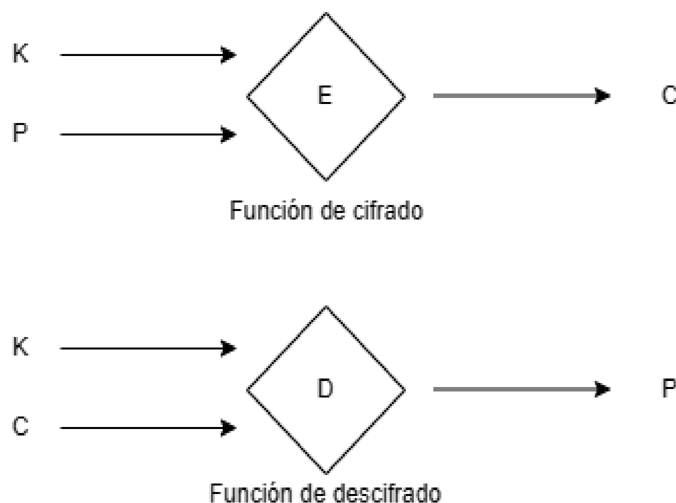

Figura 2.1: Funciones de cifrado y descifrado

En algunas ocasiones, estas funciones de cifrado y descifrado se transforman en funciones de codificación y descodificación. La diferencia con una función de cifrado/descifrado, es que mientras que las de cifrado/descifrado aplican un algoritmo que altera la estructura del mensaje para que sea inentendible por terceros, un par de funciones codificación/descodificación convierten la información de tal forma que no pueda ser entendido por terceros. Un ejemplo de codificación es el estándar UTF-8 [1]. Por ejemplo, la tabla 2.1 muestra la codificación de los 10 dígitos en UTF-8:

Decimal	0	1	2	3	4	5	6	7	8	9
UTF-8 Hex	30	31	32	33	34	35	36	37	38	39

Tabla 2.1: Codificación de los dígitos 0-9 en UTF-8

Dentro de los esquemas de cifrado simétrico propuestos (también denominados de clave secreta) se distinguen dos tipos: cifrados de flujo o cifrados de bloque. En el caso de los primeros, el tráfico se cifra bit a bit. Algunos algoritmos conocidos son RC6 [2] o Salsa20 [3]. En el segundo tipo, el tráfico se divide en bloques y se cifran con el mismo algoritmo y la misma clave. Dependiendo del tratamiento que se le de a la relación entre los bloques, se consigue un cifrado más robusto. Algunos algoritmos conocidos son 3DES, (actualmente obsoleto) [4], o AES, sustituto de 3DES y estandarizado por el *National Institute of Standards and Technology* (NIST de ahora en adelante) [5].

2.1.2. Seguridad criptográfica

Según Jean-Philippe Aumasson, un cifrado se puede considerar seguro si «no se puede aprender nada del comportamiento de cifrado cuando se aplica a otros textos, ya sean planos o cifrados» [6]. Los *modelos de ataque* y las *metas de seguridad* proveen de un baremo: una forma de identificar contra qué y contra quién es seguro un algoritmo.

Modelos de ataque

Los modelos de ataque son un conjunto de asunciones sobre como un atacante actúa, delimitando sus capacidades computacionales. En general, se suele asumir el escenario más ventajoso para el atacante ya que en la realidad se desconocen las posibles amenazas de forma concreta y además que pretenden modelar la realidad para la realización de comparativas entre protocolos seguros.

Se distinguen dos modelos:

- **Modelos de caja negra**: En este tipo de modelos el atacante desconoce o no tiene acceso a los detalles internos del esquema de cifrado. Lo único que puede observar es la entrada y la salida del mismo.

  En orden creciente de seguridad, se distinguen cuatro subtipos:

  1. *Ciphertext-only attackers (COA)*: En este modelo, el atacante toma un rol pasivo y observa los distintos *textos cifrados*, pero no tiene información sobre el texto plano ni la capacidad de cifrar bajo petición.
  2. *Known-plaintext attackers (KPA)*: En este modelo, el atacante es pasivo y observa los distintos *textos cifrados* y sabe a qué textos planos están asociados.
  3. *Chosen-plaintext attackers (CPA)*: Los atacantes tiene un rol activo en el que pueden realizar consultas de cifrado de textos planos generados por ellos mismos.
  4. *Chosen-ciphertext attackers (CCA)*: Los atacantes activos pueden realizar consultas tanto de cifrado como de descifrado. Este tipo de modelos busca ver si, aún teniendo la capacidad de cifrar y descifrar, es posible recuperar las claves utilizadas.

  Con este tipo de clasificación, los requisitos van aumentando con cada nuevo nivel. Si un algoritmo se considera seguro para un cierto modelo, implícitamente lo es para los modelos de niveles inferiores. Por ejemplo, si un algoritmo es seguro en el modelo CPA, lo es también en el modelo KPA y COA, pero no tiene porque mantener la seguridad en el modelo CCA.

- **Modelos de caja gris**: En este tipo de modelos, el atacante tiene acceso a la implementación del algoritmo de cifrado. Se distinguen dos subtipos:
    1. *Ataques de canal lateral o Side-channel attacks*: Este tipo de ataques se aprovechan de alguna debilidad de la implementación del objetivo que filtren información. Los ataques de este tipo son muy variados, por ejemplo observando el consumo de energía o el incremento de temperatura de un sistema que ejecute funciones de cifrado o descifrado.
    2. *Ataques invasivos*: Son mas poderosos que los ataques de canal lateral, pero también requieren más recursos. Este tipo de ataques manipulan el objetivo de forma directa, ya sea a nivel de software o de hardware.

Metas de seguridad

En las metas de seguridad se formaliza mediante definiciones matemáticas la seguridad de un sistema. Se distinguen dos tipos:

- **Indistinguible (IND)**: Los *textos cifrados* deben ser indistinguibles de textos generados de forma aleatoria. Esto es, si un atacante recibe dos *textos cifrados* y posee un texto plano, la probabilidad de asociarlo con el texto cifrado que le corresponde es similar al lanzamiento de una moneda, incluido si es capaz de realizar una petición de cifrado bajo diferentes modelos de caja negra.
- **No-maleabilidad (NM)**: Dado un *texto cifrado*, no es posible construir otro válido, cuyo texto plano esté relacionado con el texto plano que genero el primero. Al igual que en el caso anterior se pueden asumir diferentes modelos de caja negra.

Relación entre modelos de ataque y metas de seguridad

Como se ha nombrado anteriormente, las metas de seguridad se suelen combinar con los modelos de ataque para generar una nueva clasificación. De esta forma, la notación IND-CPA, quiere decir *Indistinguishability against chosen-plaintext attackers*, mientras que NM-CCA quiere decir *Nonmaleability against chosen-ciphertext attackers.*

En la literatura científica, si un esquema de cifrado cumple con un modelo frente a un ataque, la prueba matemática presenta los retos correspondientes y se demuestra que los supera. Por ejemplo, para probar si un algoritmo es IND-CPA, se plantea el siguiente escenario, en el que participan dos usuarios:

1. Uno de los usuarios, llamémosle *contendiente*, genera claves.

2. El adversario genera dos mensajes, y le envía uno al contendiente.
3. El contendiente genera un texto cifrado con el mensaje recibido. Una vez generado, se envía al adversario.
4. El adversario puede realizar todas las peticiones de cifrado al contendiente como desee, de forma aleatoria, adaptativa, etc.
5. Por último, el adversario debe determinar a qué mensaje pertenece el texto cifrado que recibió del contendiente inicialmente.

Si el adversario eligiese un mensaje de forma aleatoria, tendría una probabilidad de 50 % de acertar. En el reto, se considera que el adversario ha ganado si es capaz de acertar a qué mensaje corresponde el texto cifrado con una probabilidad sensiblemente mayor al 50 %. Si el atacante tiene éxito, el esquema de cifrado no es IND-CPA.

El problema que se plantea aquí es que el adversario podría volver a enviar los dos mensajes, comparar los textos cifrados, y saber con un 100 % de certeza a qué mensaje pertenece el texto cifrado del reto. Es por esto que para que un esquema de cifrado supere este reto, debe implementar algún sistema de aleatoriedad en la generación del texto cifrado, de forma que un mensaje genere dos textos cifrados distintos.

Un ejemplo de esta aleatoriedad es el **vector de inicialización** en el cifrado de bloques encadenados o *Cipher Block Chaining* (CBC). De forma simplificada, este cifrado toma el resultado de cifrar los bloques anteriores para cifrar los siguientes. La principal debilidad de este tipo de cifrado es que, tal y como está planteado, un mismo plaintext da como resultado un mismo ciphertext, independientemente de las veces que se cifre. Esto se puede corregir con la presencia de un vector de inicialización. Este vector, escogido de forma aleatoria, se utiliza para cifrar el primer bloque. Como este vector cambia entre cifrados, hace que, si se cifra un mismo plaintext, se de como resultado dos ciphertexts distintos.

2.1.3. Criptografía asimétrica

A diferencia de la criptografía simétrica, en la criptografía asimétrica se utiliza un par de claves, denominadas pública (PK) y privada (SK). La seguridad de este modelo se basa en problemas que, computacionalmente, son eficientes de realizar en una dirección, pero requieren mucha capacidad de cómputo para realizarlos en dirección contraria. Un ejemplo de esto es el algoritmo de Rivest, Shamir y Adleman (RSA de ahora en adelante), tratado en la sección 2.1.3. Los esquemas de clave pública se basan en cuatro premisas que hoy en día se consideran básicas:

1. Descifrar un mensaje cifrado tiene como resultado el mensaje

2. Las funciones de cifrado y descifrado son eficientes de calcular
3. Revelar la función de cifrado no facilita averiguar la función de descifrado. Esto está relacionado con el Principio de Kerckhoff explicado al principio de la sección.
4. El resultado de aplicar la función de descifrado a un mensaje en claro y luego aplicar la función de cifrado tiene como resultado el mensaje

Existen diversas propuestas de sistemas de cifrado asimétrico para solucionar el problema de intercambio de claves en sistemas simétricos. Es usual que se utilice los esquemas asimétricos como medio para envío de una clave compartida. Esto es debido a la diferencia de rendimiento y eficiencia entre los esquemas de cifrado simétrico y asimétrico. De nuevo se puede hacer una clasificación de los esquemas de cifrado asimétrico en función de su utilidad:

- Public Key Encryption (PKE): Esquema en el que cada participante tiene un par de claves (PK,SK), pudiendo cifrar mediante PK y descifrar mediante SK.
- Key Encapsulation Method (KEM): Esquema en el que mediante las claves pública y privada se puede derivar una clave privada que se encapsula dentro de un mensaje.
- Key Exchange (KEX): Esquema que permite a Alice y Bob acordar un clave compartida simétrica. Esta clave se calcula independientemente por cada participante. En ocasiones para generarse es necesario que la participación de todos los usuarios y otras veces solamente un usuario elige la clave. Un algoritmo muy conocido de KEX es el de Diffie-Hellman. En este algoritmo, Bob y Alice generan un par de claves pública-privada, y se comparten las claves públicas. Bob genera el secreto con la clave pública de Alice y su clave privada, mientras que Alice genera el secreto con la clave pública de Bob y su clave privada. Este secreto es el que se utiliza para cifrar la comunicación.
- Authenticated Key Exchange (AKE): Esquema que añade a las funcionalidades de un Esquema KEX, la autenticación mediante firma de la clave. El AKE más desarrollado es el algoritmo *Transport Layer Security* (TLS), utilizado en la capa de transporte de los protocolo de TCP/IP.

El proceso de firma electrónica es un esquema que se puede derivar de cualquier esquema PKE. Lo habitual es cifrar con la clave secreta un mensaje y adjuntar el mensaje original. Por motivos de eficiencia, en vez de firmar todo el mensaje se suele firmar un *resumen (hash)*. Una función *hash* es una función determinista que toma como entrada un valor y devuelve una salida, normalmente de longitud fija, denominada *hash* o resumen. Estas funciones deben cumplir las siguientes propiedades:

- Dependencia de bits: El resumen debe depender de todos los bits de la clave. Si se cambia un bit, de media deberían cambiar la mitad de los bits.
- Resistencia a las colisiones: Debe ser computacionalmente difícil encontrar una colisión, es decir, encontrar dos mensajes que tengan el mismo resumen.
- Resistencia a la segunda preimagen: Dado un mensaje, debe ser computacionalmente difícil encontrar otro cuyos resúmenes coincidan.

Son ejemplos de funciones hash MD5 [7], de uso no recomendado debido al encuentro de colisiones [8], o SHA-2 y SHA-3, este último estandarizado por el NIST [9].

Mecanismos de encapsulado de clave: PKE y KEM

Como veremos en secciones posteriores, la aparición de amenazas por la computación cuántica ha motivado la apertura de concursos del NIST con el objetivo de estandarizar uno o más algoritmos de clave pública que sean resistentes a cuántica. Estos algoritmos deben cumplir una estructura de tipo KEM, con un nivel de seguridad IND-CCA. Antes de explicar en qué se basa el mecanismo tipo KEM, hay que explicar más en detalle qué son los esquemas PKE.

En un esquema PKE existen tres fases diferenciadas, como se ve en la figura 2.2 :

- En la fase de generación, Bob genera un par de claves público-privada, y le envía su clave pública a Alice.
- Con esta clave pública Alice comienza la segunda fase: el *cifrado*. Alice genera un *texto cifrado* al aplicar la clave pública de Bob al mensaje de Alice.
- Finalmente, Alice envía este texto cifrado a Bob que comienza la última fase: el *descifrado*. En esta fase, Bob utiliza su clave privada para descifrar el texto cifrado, recuperando así el mensaje.

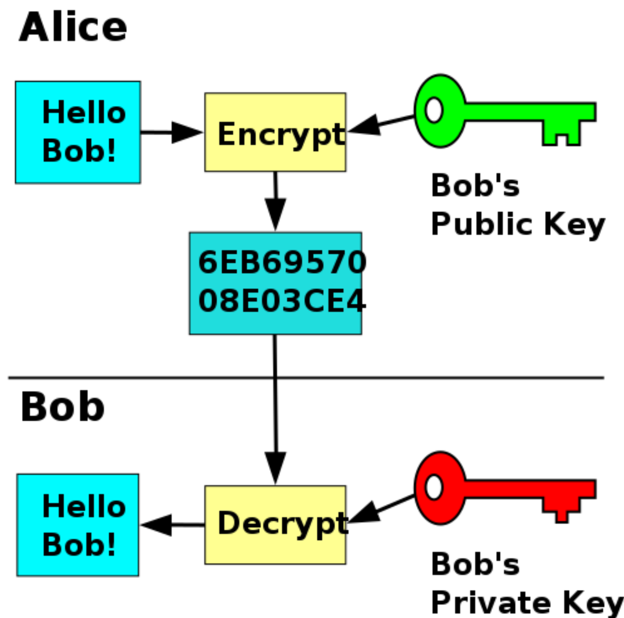

Figura 2.2: Funcionamiento de PKE.

El problema de este esquema es que solo ofrece resistencia *IND-CPA*, siempre y cuando la generación del texto cifrado tenga algún componente aleatorio. Los esquemas KEM solucionan este problema, siendo los solicitados por el concurso de estandarización NIST para la definición de esquemas de clave pública. Como veremos este esquema utiliza los mecanismos de PKE pero con distintas funciones:

- En la fase de generación, se utiliza el mecanismo de generación de PKE para generar el par de claves pública-privada.
- La fase de cifrado pasa a denominarse *encapsulado*. En esta fase Alice utiliza la clave publica de Bob y el mecanismo de cifrado de PKE para cifrar el secreto compartido, y se lo envía a Bob.
- La última fase pasa a denominarse *desencapsulado*. En esta fase se utiliza el mecanismo de descifrado de PKE con el secreto compartido cifrado que Alice ha enviado para descifrarlo.

Finalizados estos pasos, Bob y Alice pueden cifrar la comunicación con el secreto compartido. De esta forma, si un atacante se interpone en la comunicación no tiene información adicional de este secreto, ya que la comunicación se está cifrando con una clave derivada del secreto compartido, no con el par de claves como sucede en PKE. Por esto, se dota a esta estructura de seguridad IND-CCA [10].

La técnica que se utiliza en KEM se denomina *Hybrid Encription* o Cifrado Híbrido: consiste en la utilización de cifrado de clave asimétrico para el paso de una clave simétrica con la que se cifrará el resto de comunicación; teniendo como ventajas un aumento en la velocidad de la comunicación ya que el coste computacional del cifrado simétrico es menor que el del asimétrico. Otra de las ventajas de los sistemas KEM es la propuesta de una función de «Key Derivation Function» (KDF) que permiten generar la clave simétrica a partir de un valor secreto (normalmente una *password* o una *passphrase*) sin tener que utilizar otros mecanismos que pueden debilitar la seguridad como esquemas de relleno (*padding*) al introducir patrones.

Esquema RSA

En 1978, Ron Rivest, Asi Shamir y Leonard Adleman presentaron lo que hoy es uno de los criptosistemas de clave pública más conocido: un algoritmo criptográfico para securizar el sistema de correo electrónico, garantizando la privacidad del mensaje y la capacidad de firmar los mensajes [11]. Este algoritmo se denominó RSA, siguiendo las iniciales de sus creadores. Basado en el criptosistema de clave pública presentado por Whitfield Diffie y Martin E. Hellman dos años antes [12], RSA era el primer algoritmo en instanciarlo en la práctica.

El algoritmo de RSA consta de tres pasos:

1. **Generación de claves:** La generación de claves utiliza dos números primos distintos aleatorios, p y q. Además se define un número n, tal que $n = p \cdot q$. A continuación, se utiliza la función de Euler $\phi(n)$, que devuelve la cantidad de números menores que n y coprimos (primos entre sí) con n. Utilizando las siguientes propiedades:

   a) $\phi(n) = n - 1$, si n es primo.

   b) $\phi(p \cdot q) = \phi(p) \cdot \phi(q)$.

   Se llega a la conclusión de que:

   $$\phi(n) = (p-1) \cdot (q-1)$$

   Esta expresión es computacionalmente eficiente de calcular. Una vez se tiene $\phi(n)$, se escoge un numero entero e menor y coprimo con $\phi(n)$. El par (n,e) conforma la clave pública. Para el cálculo de la clave privada d, se calcula la congruencia $e * d \equiv 1 \ (mod\ \phi(n))$.

2. **Cifrado:** Alice le comunica su clave pública a Bob. Con la clave pública de Alice, Bob calcula $c \equiv m^e \ (mod\ n)$, siendo m el mensaje en claro. Envía c a Alice.

3. **Descifrado:** Alice recibe c de Bob, y con su clave privada d, calcula $m \equiv c^d$ $(mod\ n)$, consiguiendo el mensaje original.

La seguridad del algoritmo de RSA se basa en la dificultad de, sabiendo n, hallar tanto e como d. Esto está relacionado con el problema del *logaritmo discreto* y el problema de la *factorización de enteros*.

El problema del logaritmo discreto

Antes de definir qué es un logaritmo discreto y el problema del logaritmo discreto, hay que definir lo que es un *grupo abeliano*, el *orden de un elemento* y un *grupo multiplicativo*.

Un *grupo abeliano* se compone de un conjunto G y una operación $\cdot$ que combina dos elementos a y b que pertenecen a G para formar otro elemento de G. Para que esta combinación se considere un grupo abeliano, debe cumplir los cuatro *axiomas de grupo abeliano*:

- **Asociatividad:** Todo elemento de G cumple la propiedad asociativa con todos los elementos de G.
- **Existencia de elemento de identidad:** Existe un elemento, que denotaremos por 1 de forma que para cualquier elemento a de G se cumple que $a \cdot 1 = 1 \cdot a = a$.
- **Existencia de elemento inverso:** Para cada elemento a de G, existe un elemento b tal que $a \cdot b = 1$.
- **Conmutatividad:** Todo elemento de G cumple la propiedad conmutativa con todos los elementos de G.

Necesitamos introducir la siguiente notación, a^x donde a pertenece a G y x es un número entero positivo denota $a \cdot a \cdots a$ x veces.

El *orden de un elemento* $a \in G$ es el menor entero positivo p tal que $a^p = 1$. Existen subconjuntos especiales de un grupo abeliano, llamados *grupos cíclicos*, que son aquellos generados por un elemento a de orden p tal que $< a >= \{1, a, a^2, \cdots, a^{p-1}\}$

Un *grupo multiplicativo* de enteros módulo n se define como un conjunto finito de enteros positivos menores que n coprimos de n. Por ejemplo, el grupo multiplicativo de 20 se denota de la siguiente forma:

$$F_{20}^{*} = \{1, 3, 7, 9, 11, 13, 17, 19\}, \phi(20) = 8,$$

siendo ϕ la función de Euler.

Sea entonces G un grupo abeliano finito, con el elemento $g \in G$ de orden n y un elemento $a \in < g > \subset G$, entonces se define el *logaritmo discreto* de a en base g como el entero k, $0 \leq k \leq n-1$ tal que:

$$g^k = a$$

El problema del logaritmo discreto consiste en, dados un grupo abeliano G, g y a, calcular k, lo cual, computacionalmente es muy costoso. Dar el grupo G suele estar representado por una operación matemática, como la multiplicación en curvas elípticas o en los enteros módulo n.

En RSA, el cifrado se realiza con $c \equiv m^e \ (mod\ n)$, y el descifrado con $m \equiv c^d$ $(mod\ n)$. En el caso del cifrado, esto equivale a resolver el problema del logaritmo discreto donde $m^e = c$, mientras que en el caso del descifrado, equivale al problema del logaritmo discreto donde $c^d = m$.

El problema de factorización

El problema de la factorización de un numero entero en tiempo polinómico es uno de los retos actuales de la ciencia de la computación. Este problema consiste en la descomposición de un entero positivo en el producto de otros dos enteros. Este problema está planteado si el número a factorizar es muy grande. Además, su complejidad aumenta en gran medida si este número se trata de un número primo. RSA basa su seguridad en este problema: la factorización de un numero entero muy grande y, además compuesto de dos números primos grandes. Rivest, Shamir y Adleman presentan la siguiente tabla en su presentación de RSA [11] con el número de dígitos de la clave, y el número de operaciones y el tiempo que se tardaría en factorizar:

Digits	Number of operations	Time
50	$1{,}4 * 10^{10}$	3.9 hours
75	$9{,}0 * 10^{12}$	104 days
100	$2{,}3 * 10^{15}$	74 years
200	$1{,}2 * 10^{23}$	$3{,}8 * 10^9$ years
300	$1{,}5 * 10^{29}$	$4{,}9 * 10^{15}$ years
500	$1{,}3 * 10^{39}$	$4{,}2 * 10^{25}$ years

Tabla 2.2: Tabla de tiempo de factorización en función de n

Los autores recomendaban un tamaño de clave de 200, aunque esto ha variado con el tiempo a la vez que la potencia de cálculo y los avances teóricos en diseño de algoritmos. La recomendación actual es un tamaño de clave de 2656 bits, si se quiere asegurar que no se pueda descifrar hasta 2040, según las estimaciones realizadas con las ecuaciones de Lenstra actualizadas [13][14][15].

Relacionado con este problema, el 18 de marzo de 1991, *RSA Laboratories* lanzó la convocatoria a los *RSA Factoring Challenge* [16]. El objetivo del concurso era la factorización exitosa de números semiprimos (números generados a través de la multiplicación de dos números primos), conocidos como *RSA-Numbers*. Aunque el concurso fue finalizado de forma oficial en 2007, hoy en día se siguen intentando factorizar. Originalmente eran 41 números, y en 1997 se agregaron 4 [17]. Actualmente, se superado el reto RSA250 que consta de 829 bits.

Sistemas de firma digital

En sus orígenes, los sistemas de firma digital se basaban, al igual que el algoritmo de RSA, en el modelo de clave pública presentado por Diffie y Hellman. En un sistema de firma se define *una función de firma* que utiliza la clave privada y *una función de verificación*, que depende la clave pública, que se apoyan en las funciones de cifrado y descifrado.

Basándose en la cuarta propiedad de un esquema de clave pública (el resultado de aplicar la función de descifrado a un mensaje en claro y luego aplicar la función de cifrado tiene como resultado el mensaje), se puede aplicar la función de firma (descifrado) al mensaje para obtener la firma. De esta forma, puede enviarse la firma al destinatario cifrada con la clave pública del mismo. Una vez el destinatario recibe la firma y la descifra, puede aplicar la función de verificación (cifrado), y así obtener el mensaje (Figura 2.3).

Hoy en día existen distintos algoritmos de firma: Firma RSA (basado en el algoritmo ya descrito), Firma ElGamal [18] (basada en el esquema de Diffie-Hellman [12]) o Firma DSA [19] (una variante de ElGamal), entre muchas otras. Además, se suele firmar un hash del mensaje, no el propio mensaje al completo.

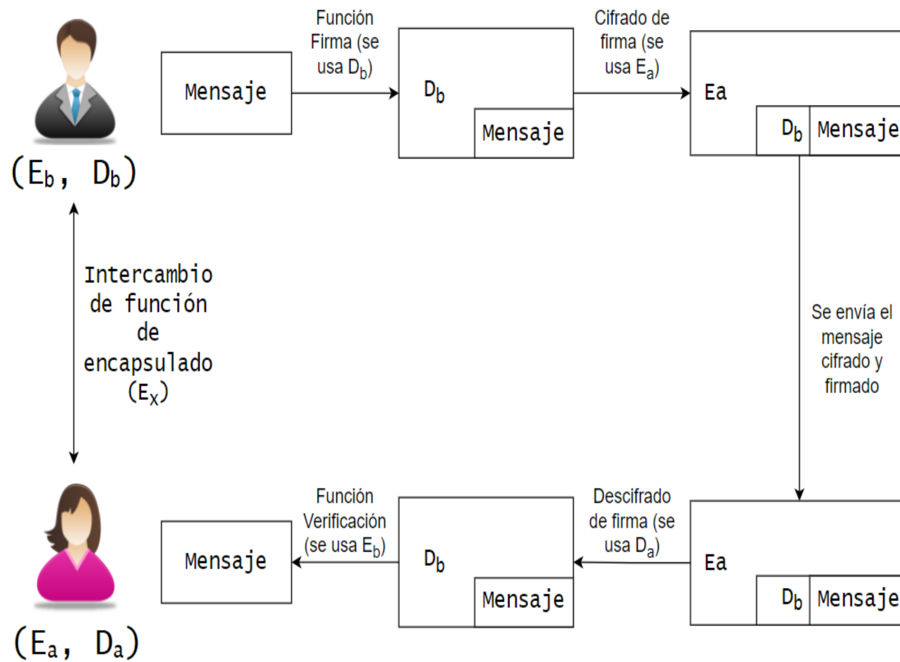

Figura 2.3: Firma RSA.

2.2. Criptografía post-cuántica

En esta sección se da una breve introducción a la computación cuántica (de forma muy simplificada) y a la notación Dirac. Esto es necesario debido a que también se presenta el algoritmo de Shor, compuesto por una parte clásica y una parte cuántica. También se presenta cómo este algoritmo puede romper RSA.

Al igual que el bit es la unidad básica de información en la computación clásica, la unidad de información básica es el qubit en la computación cuántica.

A diferencia de un bit, que se encuentra definido en 0 o 1, por el principio de superposición el qubit se encuentra en «ambos estados al mismo tiempo». En notación Dirac (también conocida como notación bra-ket), esto se representa con los denominados *ket* ($|\cdot>$). De esta forma, los dos estados posibles se representan como:

$$ket\ cero: \ |0\rangle$$
$$ket\ uno: \ |1\rangle$$

Los ket se pueden representar de forma matricial:

$$ket\ cero:\ |0\rangle\ = (1\ 0)$$
$$ket\ uno:\ |1\rangle = (0\ 1)$$

Durante los cálculos, el qubit no se encuentra definido en ninguno de los estados: se encuentra en estado de superposición. Para representar esta superposición, se utiliza la siguiente notación:

$$\alpha|0\rangle\ +\ \beta|1\rangle$$

donde el módulo al cuadrado de α y β implica las posibilidades de colapsar a ket cero o las posibilidades de colapsar a ket uno, respectivamente. Saber si un estado contiene un cero o un uno consiste en hacer un medición con probabilidad α y β de salir cualquiera de los dos símbolos. Este proceso de medición destruye el estado y queda en el ket que se ha medido.

De forma matricial, para representar el estado de superposición se utiliza una matriz de Hadamard:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

Por ejemplo:

$$(1\ 0)\cdot\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = (\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}})$$

Este estado con notación Dirac sería:

$$\frac{1}{\sqrt{2}}|0\rangle\ +\ \frac{1}{\sqrt{2}}|1\rangle$$

Que implica una probabilidad del 50 % para cada uno de los estados al colapsar $|\frac{1}{\sqrt{2}}|^2 + |\frac{1}{\sqrt{2}}|^2 = 1$ [20][21].

2.2.1. Computación cuántica y criptografía clásica

El algoritmo de Shor toma su nombre de Peter Shor, que lo propuso en 1994 [22]. Este algoritmo tiene como fin la descomposición en factores de un número N en un tiempo $O((log\ N)^3)$. Se definen las dos versiones del algoritmo, tanto su implementación en un ordenador «clásico» como su versión cuántica, donde se tiene en cuenta las propiedades de la computación cuántica.

La idea de este algoritmo es factorizar un número N utilizando la siguiente expresión $X^2 - 1 = 0\ mod\ N$, encontrando soluciones de la ecuación y comprobando

si el máximo común divisor de $x \pm 1$ con N es diferente de 1.

Algoritmo de Shor: implementación clásica

Dado N, la implementación clásica del algoritmo de Shor es la siguiente:

1. Escoger un número aleatorio $a < N \in [2, N]$.
2. Calcular mcd(a,N). Si es distinto de 1, volver al paso 1. Si mcd(a,N)=1, implica que a es coprimo de N, por lo que se puede proseguir.
3. Encontrar el periodo r de la función $f(x) = a^x \ mod \ N$.
4. Si r es impar, volver al principio.
5. Si $a^{r/2} \equiv -1 \ mod \ N$, volver al principio.
6. Los factores de N son el mcd($a^{r/2} \pm 1$,N)

Por ejemplo, para N = 63 y a = 2, la función $f(x) = a^x \ mod \ N$ toma los siguientes valores:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
f(x)	1	2	4	8	16	32	1	2	4	8	16	32	1	2	...

Tabla 2.3: Tabla de valores función f(x) en algoritmo de Shor

Observando los valores de la tabla 2.3 se observa que el periodo de la función es de 6, luego $r = 6$.

6 es par, y se cumple que $2^3 \neq -1 \ mod \ 63$. Luego, los factores de 63 son $mcd(7, 63) = 7$ y $mcd(9, 63) = 9$ [20].

Algoritmo de Shor: implementación cuántica

El algoritmo de Shor es factible cuando N es un número pequeño (como en el ejemplo). Los autores de RSA recomendaban un tamaño de clave de 200 dígitos decimales, lo que nos deja con $10^{199} - 1$ números con los que buscar factores, lo cual, requiere un numero exponencial de operaciones según aumenta el tamaño de la clave.

Es en el punto 3 del algoritmo 2.3, encontrar el periodo r de la función $f(x) = a^x \ mod \ N$, donde la computación cuántica es útil para reducir la complejidad computacional.

La entrada del algoritmo son 2 registros de qubits. El segundo registro contiene un número entero no negativo menor que N, por lo que se necesitan al menos $n = log_2 N$ qubits. El primer registro permite controlar la precisión de la aproximación realizada: Craig Gidney presentó un circuito en 2017 que conseguía realizar el cálculo con $m = 2n + 1$ qubits [23]. Esto deja una entrada como la siguiente:

$$|\psi >= |0m, 0n\rangle$$

Siendo 0m y 0n cadenas de qubits de longitud m y n, respectivamente. Al aplicar el estado de superposición con una puerta de Hadamard, que aplica este mismo estado a los qubits de la cadena, se obtiene:

$$|\psi >= \frac{\sum_{x \in \{0,1\}^m} |x, 0n\rangle}{\sqrt{2^m}}$$

Gracias a la computación cuántica y a la propiedad de la superposición, se puede evaluar la función $f(x) = a^x \ mod \ N$ en todos los estados al mismo tiempo. Esto es:

$$|\psi >= \frac{\sum_{x \in \{0,1\}^m} |x, a^x \ mod \ N\rangle}{\sqrt{2^m}}$$

Utilizando el mismo ejemplo del anterior apartado, para N = 63, n = 6 y m = 12, obteniendo que:

$$\frac{|0,1\rangle + |1,2\rangle + |2,4\rangle + |3,8\rangle + |4,16\rangle + |5,32\rangle + |6,1\rangle + |7,2\rangle + \ ...}{\sqrt{4096}}$$

De esta forma se están comparando todos los estados posibles de la función. Al saber que la función es periódica, se puede reducir aun más la cantidad de estados, ya que lo que interesa es el periodo en sí. Esto se expresa con:

$$|\psi >= \frac{\sum_{j=0}^{2^m/r-1} |t_0 + jr, a^x \ mod \ N\rangle}{\frac{2^m}{r}}$$

En la expresión superior, t_0 es la primera vez donde $a^{t_0} \equiv a^x \ mod \ N$. Se realiza una medición sobre 0n, forzando que salga de su estado de superposición. Siguiendo con el caso particular, al medirla, por ejemplo, la cadena de qubits 0n toma el valor 4 (puede tomar cualquier valor de f(x) con la misma probabilidad), lo que nos deja:

$$\frac{|2,4\rangle + |8,4\rangle + |14,4\rangle + |20,4\rangle + |26,4\rangle + ...}{4096/6}$$

Antes de realizar la última medición para obtener un resultado, hay que pasar la superposición por la transformada de Fourier cuántica. Si no se realiza esto, es posible que se nos devuelva una respuesta incorrecta, y, por propiedades de la mecánica cuántica, el resto de respuestas se destruyen. La transformada de Fourier cuántica permite que, con una alta probabilidad, la respuesta devuelta sea la correcta. Tras esto, solo quedaría sacar los factores de N con mcd$(a^{r/2} \pm 1, N)$ [20].

2.2.2. Sistemas de cifrado

Como se ha visto, con el algoritmo de Shor se puede factorizar un número N. Este algoritmo no solo permite esto, si no que es también capaz de resolver el problema del logaritmo discreto en un tiempo similar, lo cual pone aún más en riesgo los estándares de seguridad de hoy en día.

Para esto, se buscaron nuevos paradigmas que no dependiesen tanto de la factorización, con tal de crear algoritmos resistentes a los ordenadores cuánticos.

Criptografía basada en retículos

Un **retículo** o **red** (*lattice*) es una estructura matemática que consiste en una serie de puntos en un espacio n-dimensional que se repiten con una estructura periódica. Los retículos suelen generarse a través de un punto origen, y n vectores de dirección, que forman una base en $\mathbb{R}^n$, que permite representar cualquier punto en el retículo. En el caso de dos dimensiones, se puede representar como en la figura 2.4.

En retículos se ha propuesto utilizar en criptografía los dos problemas matemáticos siguientes:

1. ***Shortest Vector Problem (SVP)***: Este problema consiste en encontrar el vector más corto en el retículo, que no sea el vector nulo, combinando una base de vectores pertenecientes al retículo. En criptografía se utiliza una aproximación de este problema: encontrar un vector no nulo cuya norma se encuentra en el rango $[n_s, n_s + \lambda]$, donde n_s es la norma del vector más corto y *lambda* es una aproximación. La clave pública y la privada suelen estar compuestas por una "base mala"(bases compuestas por vectores con norma muy grande y no ortogonales entre sí) y una "base buena"(bases compuestas por vectores con norma pequeña y ortogonales entre sí), respec-

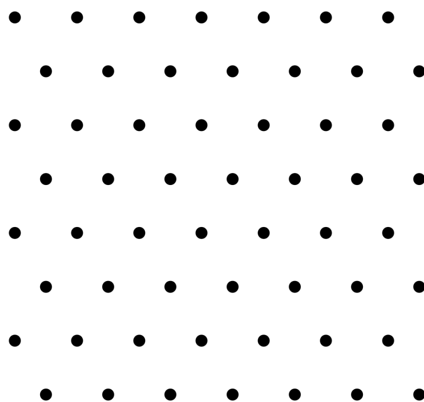

Figura 2.4: Retículo en $\mathbb{R}^2$.

tivamente. [24][25] De forma matemática, esto se traduce en lo siguiente: siendo conocidos el par (A,b), donde A es una matriz aleatoria de tamaño $m \cdot n$ y $b = A \cdot s \ mod \ q$, siendo q un número primo, hallar el vector corto s de n componentes.

2. ***Learning With Errors (LWE)***: Este problema es muy parecido a SVP en concepto: consiste en encontrar el vector s. No obstante, este vector s es escogido por el usuario. A este vector se le agrega un error e, que proviene de una distribución de probabilidades (normalmente la Gaussiana). En este caso, la clave pública viene definida por el par $\{A,B\}$, donde A es un conjunto de vectores y B es otro conjunto de vectores que viene definido por $b = A \cdot s + e \ mod \ q$, siendo q un número primo, mientras que la clave privada es el propio vector s. [26] El problema tiene una versión decisional en la que dado s decidir si se ha utilizado para calcular b o ha sido escogido de forma aleatoria o la versión de búsqueda en la que se pide calcular s directamenta.

Los problemas anteriores se pueden utilizar para cifrar de las siguiente manera: Se definen s como la clave secreta y (A, b) como clave pública.

Para cifrar el mensaje m, se elije una distribución de error y se generan e', s' y e''. Se definen entonces:

$$b' = s' \cdot A + e'$$
$$v = s' \cdot b + e'' + m$$

Una vez recibido v, entonces para descifrar se revierte la operación de la siguiente forma

$$v - b' \cdot s = m + \widehat{e}$$

Existen otro tipo de problemas basados en retículos, pero éstos solo presentan

un reto computacional cuando son el caso más difícil, lo cual no es aplicable cuando se habla de criptografía, al requerir que el caso medio sea difícil [6].

Criptografía basada en códigos

Dado un alfabeto $\mathcal{A}$ con el que se pueden formar mensajes m_x, se denomina código al conjunto de mensajes codificados p_x. Cada mensaje codificado se denomina *palabra*. Estas palabras son transmitidas por un canal y decodificadas por el receptor. Si lo que el receptor recibe es una palabra del código, se decodifica. Si lo que el receptor recibe *no* es una palabra del código, se lanza un error o se intenta *detectar los errores y corregirlos*. Para que un código sea apto para la detección de errores, su distancia mínima debe de ser estrictamente mayor de uno. La distancia en un código es el número de bits en los que difieren dos palabras del texto. La distancia mínima viene dada por el mínimo de todas las distancias del código.

Por ejemplo, sean las palabras $p_1 = (1001)$ y $p_2 = (1101)$, si al transmitir p_1 se produce un error en el segundo bit, se recibiría como p_2, palabra válida, por lo que se decodificaría. Este no es un comportamiento deseado, ya que lo que en realidad se ha producido es un error. Por esto, la distancia mínima para que un código sea apto a la corrección de errores debe ser estrictamente mayor de uno. La criptografía basada en códigos utiliza estos códigos de corrección de errores.

El primero de estos sistemas fue el criptosistema de McEliece [27]. A pesar de ser un criptosistema presentado en 1978, aun no se conoce forma de romperlo, lo cual lo vuelve muy seguro. El problema de este tipo de criptografía es que genera claves públicas muy grandes, lo cual les dota de un rendimiento mucho menor que el del resto de sistemas de cifrado [6].

Criptografía multivariante

La criptografía multivariante basa su seguridad en la resolución de sistemas de ecuaciones multivariantes. Una **ecuación multivariante** consiste en una ecuación compuesta por la suma de términos de grado uno o grado dos. Por ejemplo:

$$x_1x_5 + x_4x_3 + x_2 + x_1 = 1$$

Las incógnitas pueden tomar valores reales o enteros. Puede ser también un conjunto de los mismos. Normalmente en criptografía, las incógnitas suelen tomar valores construidos sobre el módulo de algún numero primo o binarios (0 ó 1).

El problema que se presenta es la resolución de un sistema de ecuaciones cuadrático. En este tipo de criptografía, la clave pública se compone de un con-

junto de ecuaciones multivariantes, mientras que la clave privada contiene lo necesario para invertir las ecuaciones de la clave pública.

Debido a su naturaleza, un atacante podría calcular las inversas de estas ecuaciones, revirtiendo el proceso. Es por esto que es necesario elegir los parámetros de forma muy cuidadosa [6].

Criptografía basada en hashes

La criptografía basada en hashes se basa en la seguridad que proporcionan las propias funciones hash. Es utilizado principalmente para la construcción de algoritmos de firma. El problema que presentan es que son de uso limitado: en el uso de funciones hash para la firma, se deben utilizar claves distintas cada vez. Si se utilizase la misma clave todo el rato, un atacante podría aprovecharse de esto y falsificar las firmas. Para evitarlo, se utilizan técnicas como la presentada por Ralph Merkle en 1987 [28], conocida hoy en día como Árbol Merkle o Árbol Hash (figura 2.5). Este árbol se basa en las siguientes premisas:

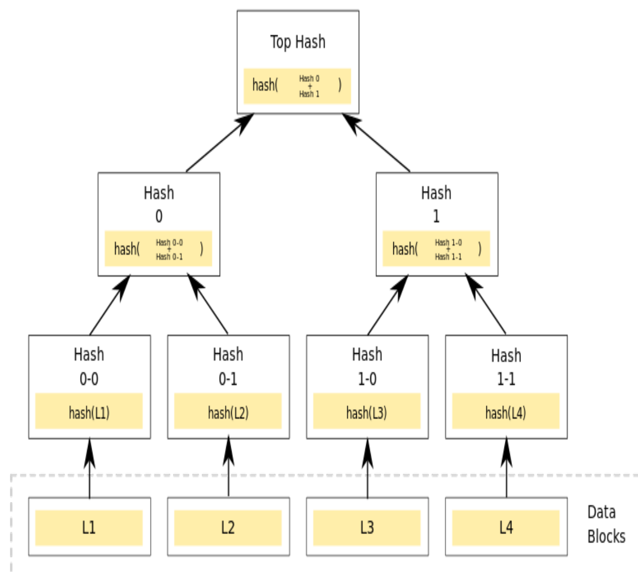

Figura 2.5: Arbol de Merkle

1. La raíz del árbol es autenticada por ser de dominio público. Hoy en día, normalmente este nodo va firmado para asegurar su integridad y fiabilidad.

2. Un nodo del árbol autentica los subnodos de la izquierda

3. Un nodo del árbol autentica los subnodos de la derecha

4. Un nodo tiene la capacidad de firmar un único mensaje

Algoritmos basados en hash que utilizan este tipo de estructuras o similares se denominan *algoritmos de estado*, mientras que los que no las utilizan se denominan *algoritmos sin estado*. Estos últimos utilizan otro tipo de técnicas, como el uso de algoritmos de firma de un único uso y producen firmas mayores que los del primer tipo. En la convocatoria del NIST, se estipula que los algoritmos deben de soportar un mínimo de 2^{64} firmas seguras. [29]

2.3. Concurso de estandarización del NIST

En 2017, el NIST cerró las candidaturas de un concurso de carácter público con la finalidad de «solicitar, evaluar y estandarizar uno o mas algoritmos de clave pública resistentes a cuántica» [30]. La primera ronda se cerró con 69 candidatos [31], de los cuales pasaron 26 a la siguiente ronda: 17 para intercambio de clave y 9 para firma digital [32].

En 2022, el NIST ha definido un primer grupo de ganadores. Para intercambio de clave, el ganador es *CRYSTALS-Kyber*, mientras que para firma digital, los ganadores son *CRYSTALS-Dilithium*, *Falcon* y *SPHINCS+*. [33]. En verano de 2023, se abrió una nueva ronda para firma digital que pedía nuevos esquemas que no estuvieran basados en criptografía de retículos y que se encuentra actualmente en evaluación [1].

2.3.1. Niveles de seguridad del NIST

Para cumplir con las metas del concurso, el NIST define cinco niveles distintos de seguridad en su convocatoria [29]. Estos niveles se basan en la comparación con el coste computacional que tendría realizar cierto ataque. En concreto, los cinco niveles que se definen son:

1. Cualquier ataque que rompa la seguridad debe requerir un coste computacional comparable o mayor a los necesarios para la búsqueda de claves en un bloque cifrado con una clave de 128 bits (por ejemplo, AES128)

2. Cualquier ataque que rompa la seguridad debe requerir un coste computacional comparable o mayor a los necesarios para la búsqueda de una colisión en una función hash de 256 bits (por ejemplo, SHA256/SHA3-256)

[1] Anuncio del NIST, 17 de Julio de 2023

3. Cualquier ataque que rompa la seguridad debe requerir un coste computacional comparable o mayor a los necesarios para la búsqueda de claves en un bloque cifrado con una clave de 192 bits (por ejemplo, AES192)

4. Cualquier ataque que rompa la seguridad debe requerir un coste computacional comparable o mayor a los necesarios para la búsqueda de una colisión en una función hash de 384 bits (por ejemplo, SHA384/SHA3-384)

5. Cualquier ataque que rompa la seguridad debe requerir un coste computacional comparable o mayor a los necesarios para la búsqueda de claves en un bloque cifrado con una clave de 256 bits (por ejemplo, AES256)

2.3.2. Algoritmos candidatos

Candidatos de KEM

Algunos candidatos que llegaron hasta las rondas finales fueron:

- ***CRYSTALS-Kyber:*** Algoritmo basado en retículos y ganador en la categoría de KEM del concurso de estandarización del NIST. Existen tres variantes distintas que ofrecen distintos niveles de seguridad: Kyber-512, que entra dentro del nivel 1 de seguridad, Kyber-768, dentro del nivel 3 de seguridad y Kyber-1024, dentro del nivel 5 de seguridad [34].

- ***Classic McEliece:*** Algoritmo basado en una aplicación dual del criptosistema de Niederreiter (basado a su vez en el criptosistema de McEliece) utilizando códigos Goppa binarios que permiten corregir t errores. [35]. La clave privada de un individuo se encuentra definida por tres matrices: S, una matriz aleatoria invertible, H, la matriz generatriz, y P, una matriz de permutación aleatoria; mientras que la clave pública se encuentra definida por una matriz H' = SHP, y t.

  Para codificar un mensaje, Bob multiplica H' por el mensaje. Para decodificar el mensaje, Alice multiplica el mensaje codificado por S^{-1} y aplica un algoritmo de decodificación, quedando solo mP^T. En ultima instancia, para obtener el mensaje se multiplica el miembro restante por la inversa de la transpuesta de P, es decir $(P^{-1})^T$ [36].

  Classic McEliece acepta 5 tamaños distintos de clave pública y de clave privada: mceliece348864 (nivel 1), mceliece460896 (nivel 3), mceliece6688128 (nivel 5), mceliece6960119 (nivel 5), mceliece8192128 (nivel 5).

- ***SABER:*** Algoritmo basado en problemas tipo *Module Learning With Rounding* que son una variante de los problemas LWE. En vez de crear los errores mediante un error al azar, estos se crean mediante redondeo determinista

[37]. En el problema LWR original, se da como salida una versión de $\lceil \alpha \cdot A \cdot s \rfloor$, siendo $\alpha \in \mathbb{Q}$ redondeada de forma determinista [38].

SABER ofrece tres variantes que buscan cubrir distintos niveles de seguridad: LightSaber, dentro del nivel 1; SABER, dentro del nivel 3; y FireSaber, dentro del nivel 5.

- ***NTRU:*** Versión resistente a la cuántica del algoritmo homónimo presentado en 1998. Se basa en retículos, concretamente en álgebra de polinomios y en el módulo de reducción de dos números primos entre sí [39].

Otros algoritmos a destacar son Frodo [40] basado en retículos, además de BIKE [41] y HQC [42] basados en códigos, que se encontraban en la categoría de candidatos alternativos [43].

Candidatos de Firma Digital

Los ganadores de firma digital fueron:

- ***CRYSTALS-Dilithium:*** Creado por las mismas personas que *CRYSTALS-Kyber*, es uno de los tres ganadores en el apartado de firma digital. Al igual que *CRYSTALS-Kyber* está basado en retículos. Ofrece tres variantes: Dilithium2 (nivel 2), Dilithium3 (nivel 3) y Dilithium5 (nivel 5) [44].

- ***Falcon:*** Ganador en el apartado de firma digital. Se basa en el marco teórico de Gentry, Peikert and Vaikuntanathan. Este marco propone el uso de retículos con unos elementos denominados *trapdoors*. De forma muy simplificada, estas *trapdoors* se tratan de un conjunto de funciones sobreyectivas y de muchas a uno, es decir, cada elemento del codominio tiene múltiples preimagenes [45]. Gracias a estas propiedades, estas funciones son útiles en criptografía ya que, como las funciones hash, se denominan *funciones de un solo sentido*.

  Utilizando este marco, se hace uso de NTRU y una *trapdoor* denominada *Fast Fourier Sampling* para resolver un problema tipo: *Short Integer Problem* [46]. Estos problemas se basan en la siguiente premisa: sea A una matriz consistente en m vectores uniformes, encontrar un vector distinto de cero tal que $A * z = 0 \in Z_q^n$, siendo Z_q^n un conjunto de vectores n-dimensionales módulo q [47].

- ***SPHINCS⁺:*** Último algoritmo ganador. En la fase tres se encontraba en reserva. *SPHINCS⁺* se basa en hashes sin estado. Tiene tres variedades: SPHINCS⁺-SHAKE256, SPHINCS⁺-SHA-256 y SPHINCS⁺-Haraka. Cada variedad abarca los niveles 1, 3, y 4 de seguridad. [48].

Otro algoritmo de firma que no consiguió pasar a la final es Rainbow [49].

3

Ejecución y resultados

3.1. Ejecución y código

En este capitulo estudiamos la eficiencia sobre las implementaciones oficiales de los diferentes criptosistemas. Nuestro objetivo es hacer una comparación del rendimiento computacional de forma repetible según los estándares de «reproducible research».

3.1.1. Librerías utilizadas

En primer lugar se introduce la librería principal, *liboqs*, que ofrece una API para el uso de algoritmos criptográficos post-cuánticos. También se introduce la librería PQClean, que permite la integración con *liboqs*, y por último se introduce la librería x86intrin, que es utilizada en el código ya que contiene una función para el recuento del tiempo de ejecución.

liboqs

La librería liboqs es una librería de *Open Quantum Safe* (OQS de ahora en adelante). OQS se trata de un proyecto de código abierto cuyo objetivo es dar apoyo al desarrollo y a la creación de prototipos en la criptografía resistente a la cuántica [50]. En particular, liboqs es una librería de código abierto escrita en C para algoritmos criptográficos resistentes a la cuántica disponible en Github

[51]. Ha sido escogida para utilizarse en este trabajo debido a que ofrece una API sencilla que permite utilizar los esquemas criptográficos que se han introducido anteriormente. Actualmente ofrece soporte para los siguientes algoritmos de encapsulado: BIKE, Classic McEliece, Frodo KEM, HQC, CRYSTALS-Kyber y NTRU-Prime. También ofrece soporte para los siguientes algoritmos de firma: CRYSTALS-Dilithium, Falcon, SPHINCS$^+$-Haraka, SPHINCS$^+$-SHA256 y SPHINCS$^+$-SHAKE256.

Liboqs ofrece, además, scripts para implementar algoritmos de otras librerías, integrándolos así en la API. De esta forma se han podido obtener los algoritmos para SABER en KEM y los algoritmos de Rainbow en firma digital de la librería *PQClean*, explicada en el siguiente subapartado.

De la API de liboqs, para los algoritmos de KEM se han utilizado los siguientes tipos de datos:

```
1     OQS_KEM kem* = NULL
2     OQS_STATUS rv
```

Por su lado, OQS_KEM se trata de un struct con los siguientes campos:

- *method_name:* String que contiene el nombre del algoritmo
- *alg_version:* String que contiene la version del algoritmo
- *claimed_nist_level*: Entero que contiene el nivel de seguridad del algoritmo (1, 2, 3, 4 o 5)
- *ind_cca: True* si el algoritmo ofrece seguridad IND-CCA o *False* si ofrece segurdad IND-CPA
- *length_public_key:* Longitud máxima de la clave pública
- *length_secret_key:* Longitud máxima de la clave privada
- *length_ciphertext:* Longitud máxima del ciphertext
- *length_shared_secret:* Longitud máxima del secreto compartido

Mientras que OQS_STATUS representa los distintos valores de retorno que las funciones de liboqs dan. Estos pueden ser:

- *OQS_ERROR:* Devuelto cuando la función ha fallado
- *OQS_SUCCESS*: Devuelto cuando la función ha retornado con exito

- *OQS_EXTERNAL_LIB_ERROR_[LIB]:* Devuelto cuando alguna librería externa ha devuelto algún fallo. La librería se especifica en el campo [LIB].

Y también se han utilizado las siguientes funciones:

```
1     OQS_KEM_new(const char *method_name)
2     OQS_KEM_keypair(const OQS_KEM *kem, uint8_T *public_key,
         uint8_t *secret_key)
3     OQS_KEM_encaps(const OQS_KEM *kem, uint8_t *ciphertext,
         uint8_t *shared_secret, const uint8_t *public_key)
4     OQS_KEM_decaps(const OQS_KEM *kem, uint8_t *shared_secret,
          uint8_t *ciphertext, const uint8_t *secret_key)
```

Con la primera función se genera el objeto KEM; con la segunda se genera el par de claves; con la tercera se produce el encapsulado; y con la última se produce el desencapsulado.

Para los algoritmos de firma digital, los tipos de datos son los mismos, adaptados para la firma digital. Las funciones son:

```
1     OQS_SIG_new(const char *method_name)
2     OQS_SIG_keypair(const OQS_SIG *kem, uint8_T *public_key,
         uint8_t *secret_key)
3     OQS_SIG_sign(const OQS_SIG *sig, uint8_t *signature,
         size_t *signature_len, const uint8_t *message, size_t
         message_len, const uint8_t *secret_key)
4     OQS_SIG_verify(const OQS_SIG *sig, const uint8_t *message,
          size_t message_len, const uint8_t *signature, size_t
         signature_len, const uint8_t *public_key)
```

Con la primera función se genera el objeto SIG; con la segunda se genera el par de claves; con la tercera se produce la firma; y con la última se verifica la firma.

PQClean

PQClean se trata de una librería disponible en Github [52].

El objetivo de esta librería es recolectar implementaciones limpias de los algoritmos enviados al concurso del NIST. Además, ofrece soluciones para la integración con librerías como liboqs. Gracias a esta librería se ha podido integrar SABER y Rainbow, algoritmos de encapsulado y de firma, respectivamente.

x86intrin

La librería x86intrin.h es una librería incluida en GCC. Esta librería se compone de intrínsecos: funciones que permiten realizar operaciones de ensamblador. En concreto, de esta librería se utiliza la función *__rdtsc()*. Esta función devuelve el timestamp del procesador.

Se ha utilizado esta función en vez de *clock()* de la librería time.h ya que es más precisa para calcular el número de ciclos [53]. Más información sobre la función en: Intel Intrinsics Guide

3.1.2. Código

El código utilizado se encuentra disponible en el anexo C. En concreto se compone de dos archivos: kem.c (ver Anexo C) y sig.c (ver Anexo C) que dan lugar a los resultados que se estudian en la siguiente sección.

La estructura de ambos archivos es similar:

- Se inicializan las variables iniciales necesarias: el puntero al algoritmo instanciado, «vaults» de datos donde guardar información producida por los algoritmos, punteros a archivos utilizados para guardar los resultados, etc.
- Se realiza un bucle con *iteraciones* repeticiones para generar las claves
- Se realiza un bucle con *iteraciones* repeticiones para cifrar/firmar la información
- Se realiza un bucle con *iteraciones* repeticiones para descifrar/verificar la información

3.1.3. Algoritmos KEM

Los algoritmos KEM seleccionados se pueden encontrar en la tabla A. A modo de recordatorio: El único esquema elegido para la estandarización ha sido Kyber (con sus 6 variantes). Classic McEliece y Saber formaban parte de los finalistas, mientras que BIKE, HQC y FrodoKEM formaban parte de los candidatos alternativos.

3.1.4. Algoritmos de Firma digital

Los algoritmos de firma digital que se han seleccionado se pueden encontrar en las tablas A.2 y A.3. A modo de recordatorio: Los algoritmos seleccionados para la estandarización fueron Dilithium, Falcon y SPHINCS$^+$. Rainbow formaba parte de los candidatos alternativos.

3.2. Resultados

En esta sección se presenta una comparación de los distintos algoritmos en cuanto a características y a media de ciclos que se han tardado en ejecutar las distintas fases que los componen. La ejecución se ha llevado a cabo en Ubuntu 22.04 gestionado por *Windows Subsystem for Linux* (WSL2) con un Intel(R) Core(TM) i7-9750H 2.60GHz.

3.2.1. Algoritmos KEM

En esta sección se analizarán los resultados de ejecutar los distintos algoritmos de encapsulado disponibles en la librería, más los agregados.

Se comparan los distintos tamaños máximos para la clave pública, la clave privada y el texto cifrado, así como una media de los ciclos que tardan en ejecutarse la generación de claves, la encapsulación y la desencapsulación.

Se ha excluido NTRU-Prime de la comparación ya que la librería solo ofrece la versión ntruprime-sntrup761, que entra dentro del nivel de seguridad 2, por lo que no se tendrían otros candidatos con los que comparar de forma justa.

Así mismo, se ha omitido la representación gráfica de Kyber y Saber en cuanto a la media de ciclos, ya que había tanta diferencia en los valores que no aparecían reflejados. También se ha separado el esquema de McEliece a la subsección 3.2.1 por la diferencia de escala en las figuras, que hacia la presentación de resultados menos clara.

Nivel de seguridad 1

Las características de los algoritmos en nivel 1 son las siguientes:

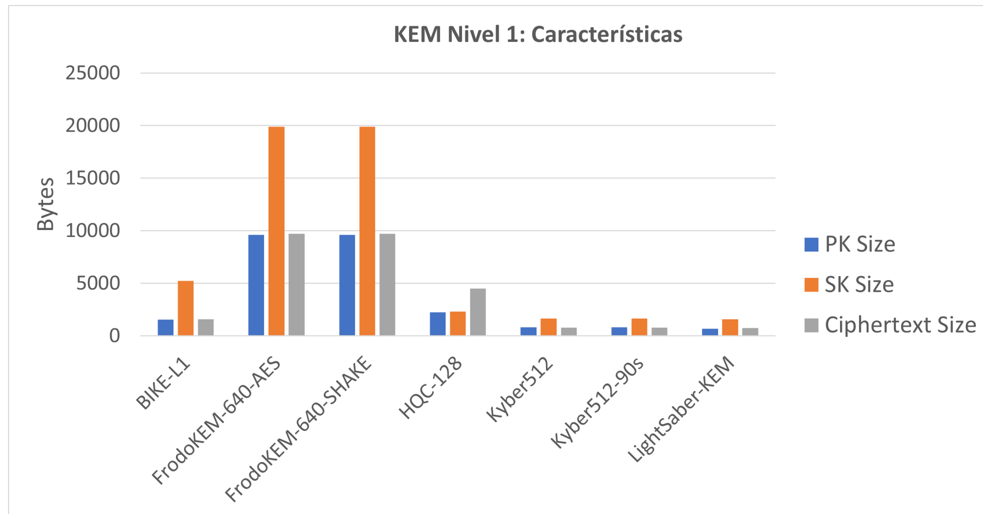

Figura 3.1: Gráfico características de algoritmos KEM de nivel 3

Algoritmo	PK Size (Bytes)	SK Size (Bytes)	Ciphertext (Bytes)
BIKE-L1	1541	5223	1573
FrodoKEM-640-AES	9616	19888	9720
FrodoKEM-640-SHAKE	9616	19888	9720
HQC-128	2249	2289	4481
Kyber512	800	1632	768
Kyber512-90s	800	1632	768
LightSaber-KEM	672	1568	736

Tabla 3.1: Características de algoritmos KEM de nivel 1

En cuanto a las características, llama la atención el caso de Frodo, que utiliza claves y texto cifrado muy grandes en comparación con los demás algoritmos. Esto se debe a que de forma interna utiliza matrices para las operaciones, en vez de vectores, por lo que requiere más capacidad de memoria [54].

El resto de algoritmos tienen unas características similares entre sí.

En cuanto al rendimiento, los datos son los siguientes:

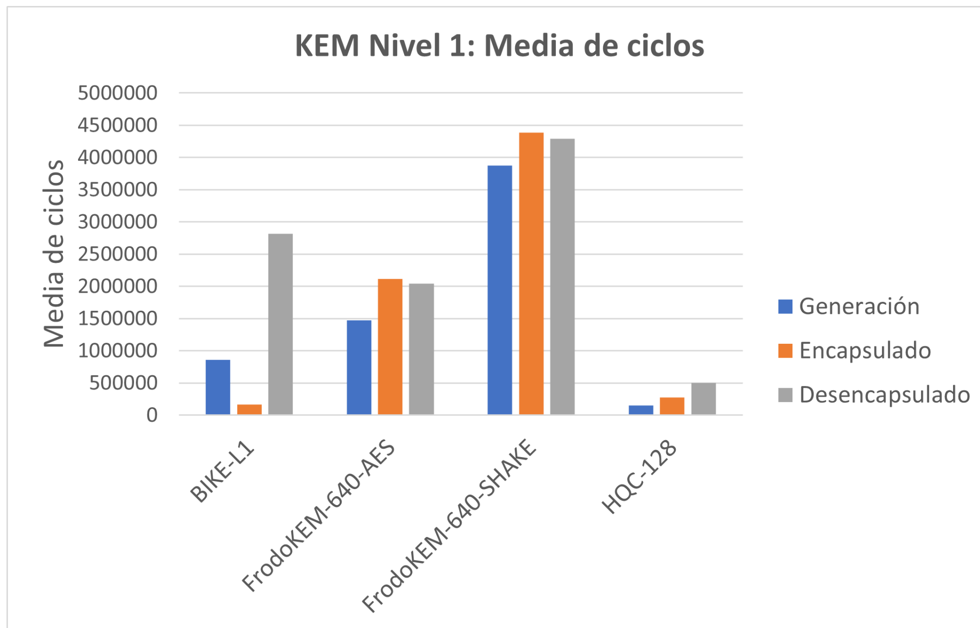

Figura 3.2: Gráfico características de algoritmos KEM de nivel 1

Algoritmo	Generación	Encapsulado	Desencapsulado
BIKE-L1	856499	161874	2816238
FrodoKEM-640-AES	1471691	2117455	2043166
FrodoKEM-640-SHAKE	3873341	4387440	4290312
HQC-128	148078	277880	497634
Kyber512	41736	47477	34467
Kyber512-90s	32167	41531	27449
LightSaber-KEM	58107	63177	56760

Tabla 3.2: Media de ciclos de ejecución algoritmos KEM de nivel 1.

Es en la ejecución donde se observa la mayor diferencia entre algoritmos. Se pueden distinguir tres grupos:

- Los algoritmos más rápidos, entre los que están Kyber y Lightsaber, ambos basados en retículos.
- HQC, basado en códigos , con un rendimiento también aceptable.

- Dentro de los algoritmos con peor rendimientos, BIKE-L1 basado en códigos lineales es el que menos ciclos requiere y Frodo, que pese a estar basado en retículos, por su funcionamiento interno, es el que más tarda de todos.

Llama la atención también, como el hecho de utilizar como función Hash SHAKE [1] internamente en lugar de AES como generador de elementos pseudoaleatorios, hace que el número de ciclos que tarda en ejecutarse cada fase prácticamente se duplique en el caso de Frodo.

Aunque ambos estén basados en código, HQC y BIKE siguen una filosofía diferente. Mientras que BIKE utiliza *Quasi-Cyclic Moderate Density Parity-Check Codes* para corregir los errores, HQC utiliza *codewords* y distancia de Hamming, lo que parece hacer al segundo más veloz en cuanto a tiempo de ejecución en las tres fases, siendo la fase de encapsulado donde menos diferencias hay.

Dentro de los algoritmos basados en retículos, y excluyendo a Frodo, no hay diferencias notables: los tres algoritmos tardan una media de ciclos similar, siendo LightSaber el que más tarda, seguido por Kyber512 y Kyber512-90's. La diferencia entre estos dos últimos es que Kyber-90s utiliza AES-256 y SHA2 en vez de SHAKE [34].

Nivel de seguridad 3

Las características de los algoritmos en nivel 3 son las siguientes:

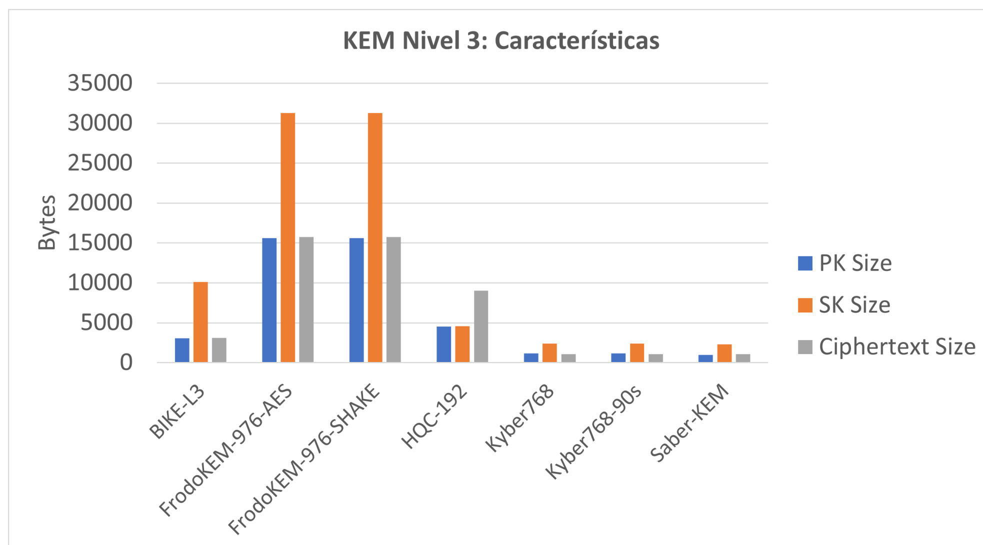

Figura 3.3: Gráfico características de algoritmos KEM de nivel 3.

[1]Secure Hash Algorithm and KECCAK

Algoritmo	PK Size (Bytes)	SK Size (Bytes)	Ciphertext (Bytes)
BIKE-L3	3083	10105	3115
FrodoKEM-976-AES	15632	31296	15744
FrodoKEM-976-SHAKE	15632	31296	15744
HQC-192	4522	4562	9026
Kyber768	1184	2400	1088
Kyber768-90s	1184	2400	1088
Saber-KEM	992	2304	1088

Tabla 3.3: Características de algoritmos KEM de nivel 3.

Si se observa, la figura 3.3 es muy similar a la figura 3.1. Esto se debe a que la diferencia entre estos algoritmos reside en un aumento en el número de bytes que componen la clave pública, la clave privada y el ciphertext. Para una mejor comparación, se generan la figura y tabla 3.4 y 3.4, respectivamente.

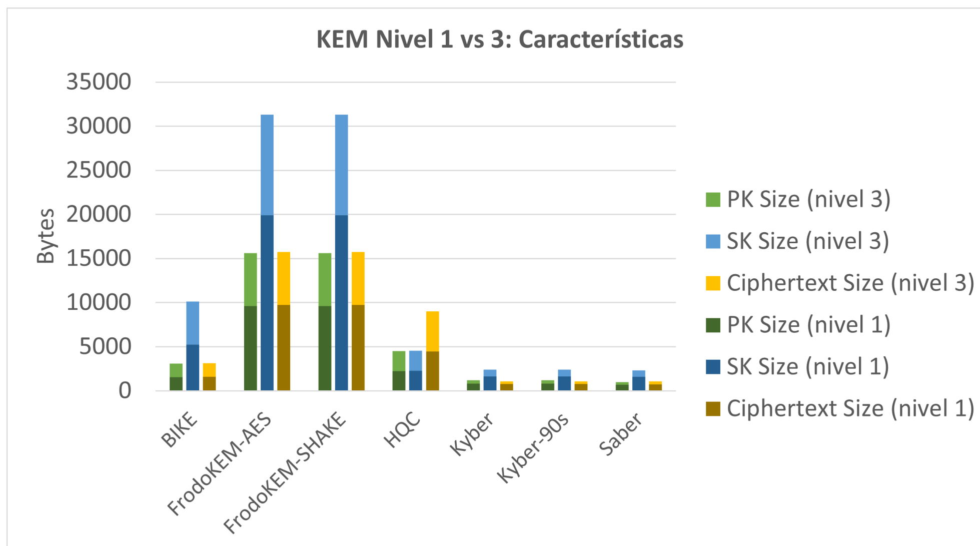

Figura 3.4: Gráfico características de algoritmos KEM de nivel 1 (color oscuro) vs nivel 3 (color claro).

En la figura 3.4 se observa la diferencia entre en nivel 1 y el nivel 3 en cada caso. En la tabla 3.4 se reflejan los coeficientes de crecimiento. Mientras que algoritmos como BIKE o HQC duplican el tamaño de sus componentes, el resto de algoritmos tienen un crecimiento menor.

Algoritmo	Coeficiente PK	Coeficiente SK	Coeficiente Texto cifrado
BIKE	2.01	1.93	1.98
Frodo	1.63	1.57	1.62
HQC	2.01	1.99	2.01
Kyber	1.48	1.47	1.42
Saber	1.48	1.47	1.48

Tabla 3.4: Coeficientes de crecimiento (cociente entre el nivel 3 y nivel 1 KEM: características.

La siguiente figura representa el rendimiento de los algoritmos KEM de nivel 3 más costosos, estando todos los algoritmos reflejados en la tabla para su comparación:

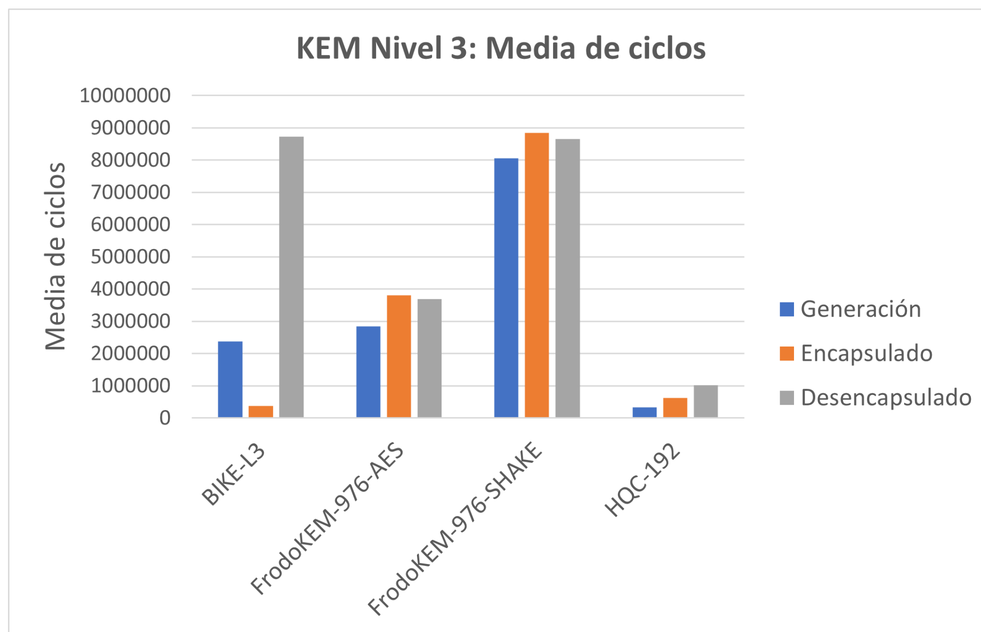

Figura 3.5: Gráfico media de ciclos de ejecución algoritmos KEM de nivel 3.

Algoritmo	Generación	Encapsulado	Desencapsulado
BIKE-L3	2379099	372475	8725927
FrodoKEM-976-AES	2844581	3811944	3692599
FrodoKEM-976-SHAKE	8063816	8844841	8655748
HQC-192	334942	628901	1023148
Kyber768	59215	71000	52490
Kyber768-90s	45378	55692	36855
Saber-KEM	87596	99283	92818

Tabla 3.5: Media de ciclos de ejecución algoritmos KEM de nivel 3.

BIKE y Frodo son los que más ciclos consumen. En el caso de Frodo, el uso de AES mejora el rendimiento, pero sigue sin poder rivalizar con HQC, Kyber o Saber. Llama la atención la diferencia entre las tres fases en el caso de BIKE-L3: el número de ciclos de la generación y el desencapsulado es muy alto en comparación con el encapsulado.

La siguiente figura 3.6 y la siguiente tabla 3.6 ilustran las diferencias entre el nivel 1 y el nivel 3.

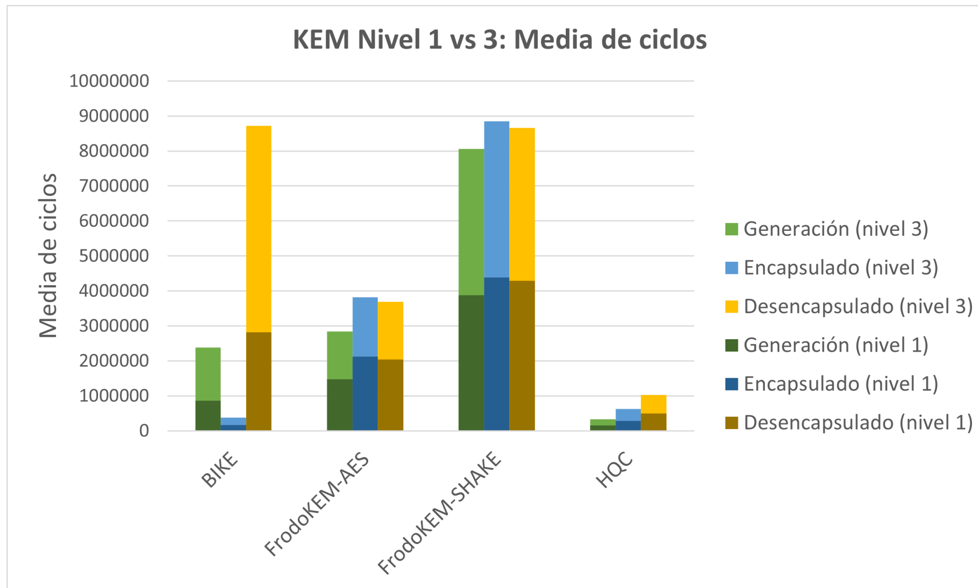

Figura 3.6: Gráfico media de ciclos de algoritmos KEM de nivel 1 (color oscuro) vs nivel 3 (color claro).

Algoritmo/Coeficiente	Generación	Encapsulado	Desencapsulado
BIKE	2.78	2.3	3.1
FrodoKEM-AES	1.93	1.8	1.81
FrodoKEM-SHAKE	2.08	2.02	2.02
HQC	2.26	2.26	2.06
Kyber	1.42	1.5	1.52
Kyber90s	1.417	1.34	1.35
Saber-KEM	1.51	1.57	1.64

Tabla 3.6: Coeficientes de crecimiento en media de ciclos nivel 1 - nivel 3 KEM: Cociente entre el nivel 3 y el nivel 1.

El crecimiento de BIKE no es lineal: un aumento del doble de tamaño en sus componentes implica un crecimiento de más del doble en el número de ciclos, especialmente en el desencapsulado, donde hay un crecimiento del 200 %. No obstante, llama la atención los pocos ciclos que tarda en realizar el encapsulado en comparación con la generación y el desencapsulado. En el caso de Frodo el

número de ciclos se duplica al aumentar un poco el tamaño de sus componentes. Estos dos algoritmos son los que peor rinden.

En el caso de HQC, Kyber y Saber, lo más aproximado sería un crecimiento lineal (al menos con los datos observados hasta ahora).

Nivel de seguridad 5

Las características de los algoritmos en nivel 5 son las siguientes:

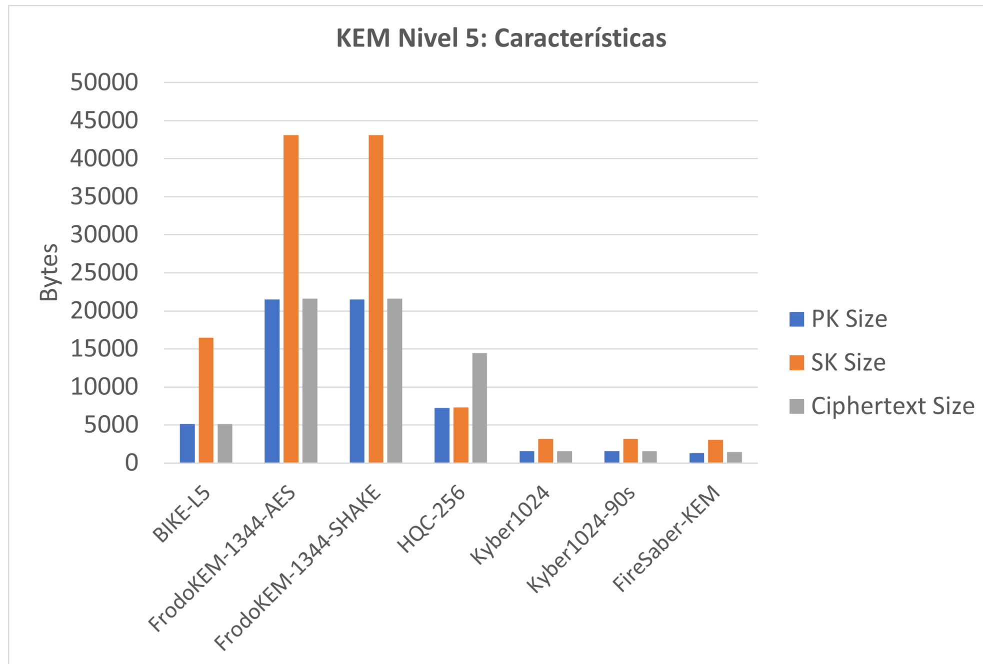

Figura 3.7: Tamaño en bytes para cada fase de los algoritmos KEM de nivel 5.

Algoritmo	PK Size (Bytes)	SK Size (Bytes)	Ciphertext Size (Bytes)
BIKE-L5	5122	16494	5154
FrodoKEM-1344-AES	21520	43088	21632
FrodoKEM-1344-SHAKE	21520	43088	21632
HQC-256	7245	7285	14469
Kyber1024	1568	3168	1568
Kyber1024-90s	1568	3168	1568
FireSaber-KEM	1312	3040	1472

Tabla 3.7: Características de algoritmos KEM de nivel 5.

Igual que en la anterior ocasión, no se aprecian diferencias notables entre la figura 3.3 y la figura 3.7 a simple vista, mas allá del aumento del eje Y: ambas figuras mantienen la misma relación entre los esquemas propuestos.

El crecimiento con respecto al nivel 3 se encuentra reflejado en la figura 3.8 y la tabla 3.8

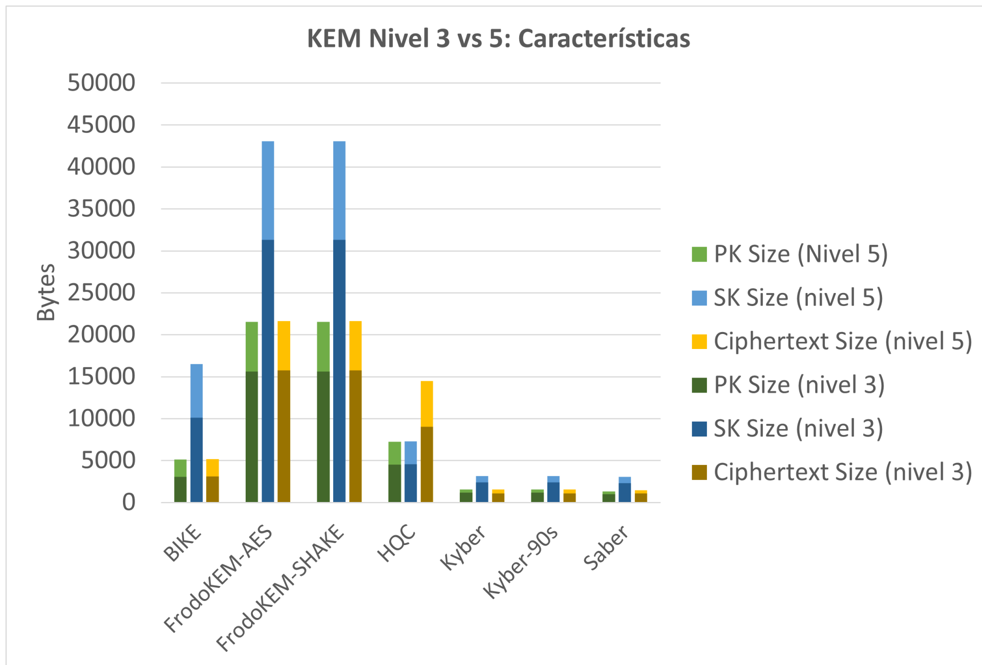

Figura 3.8: Tamaño de clave de algoritmos KEM de nivel 3 (color oscuro) vs nivel 5 (color claro)

Algoritmo	Crecimiento PK	Crecimiento SK	Crecimiento Ciphertext
BIKE	1.66	1.63	1.65
Frodo	1.38	1.38	1.37
HQC	1.60	1.6	1.60
Kyber	1.32	1.32	1.44
Saber	1.32	1.32	1.35

Tabla 3.8: Coeficientes de crecimiento nivel 3 - nivel 5 KEM: características. Cociente entre nivel 5 y nivel 3

En esta ocasión, como se observa en la tabla 3.8, ningún algoritmo aumenta el tamaño de sus componentes al doble de la anterior: todos crecen en menor medida. Frodo, Kyber y Saber son los que menos crecimiento experimentan entre ambos niveles.

Los resultados respecto a la ejecución son los siguientes:

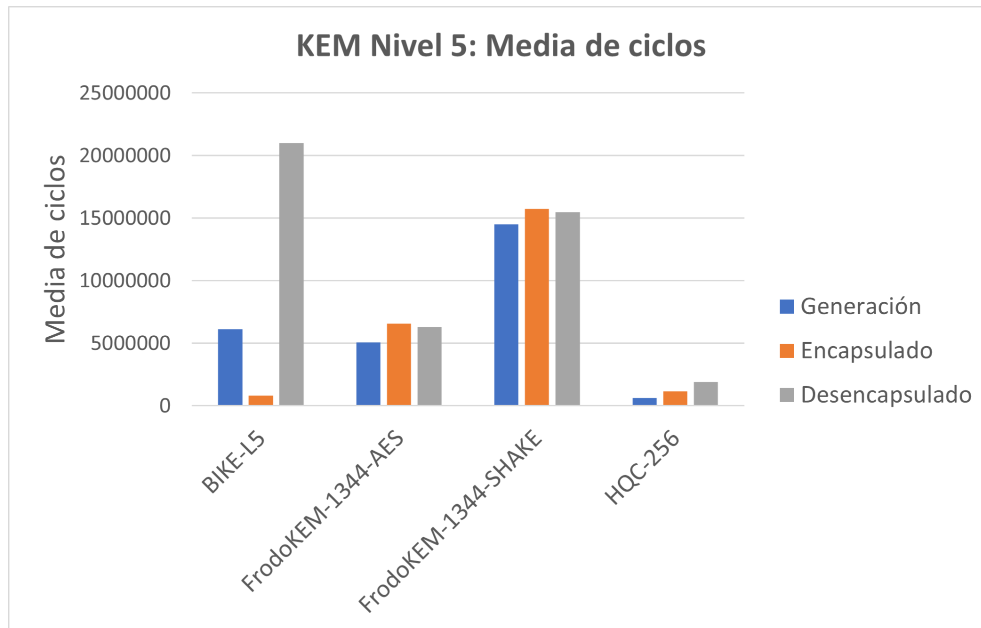

Figura 3.9: Gráfico media de ciclos de algoritmos KEM de nivel 5.

Algoritmo	Generación	Encapsulado	Desencapsulado
BIKE-L5	6128013	810399	21016567
FrodoKEM-1344-AES	5090832	6562956	6294773
FrodoKEM-1344-SHAKE	14509182	15749206	15486416
HQC-256	620912	1175907	1912313
Kyber-1024	80593	96706	76248
Kyber-1024-90s	56749	72891	51569
FireSaber-KEM	127746	145264	138926

Tabla 3.9: Media de ciclos de ejecución algoritmos KEM de nivel 5.

Igual que en los otros dos niveles, tanto HQC como Kyber y Saber se diferencian de BIKE y Frodo. BIKE, por su lado, tarda muchos ciclos en la generación y en el desencapsulado, siendo en este último el que más tarda de todos los algoritmos, incluso por encima de Frodo-SHAKE.

La diferencia entre Frodo-AES y Frodo-SHAKE no parece acentuarse más a medida que se aumenta el tamaño de sus componentes.

La siguiente figura ilustra la diferencia de ciclos entre los algoritmos en nivel 3 y nivel 5:

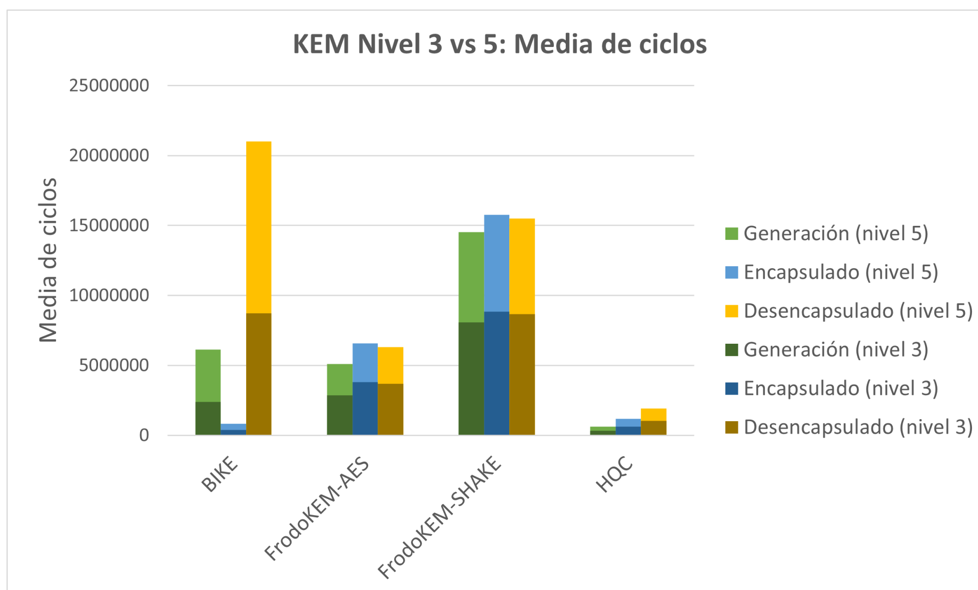

Figura 3.10: Gráfico media de ciclos de algoritmos KEM de nivel 3 (color claro) vs nivel 5 (color oscuro)

Algoritmo	Generación	Encapsulado	Desencapsulado
BIKE	2.58	2.18	2.4
FrodoKEM-AES	1.79	1.72	1.7
FrodoKEM-SHAKE	1.8	1.78	1.79
HQC	1.85	1.87	1.87
Kyber	1.36	1.36	1.45
Kyber-90s	1.25	1.3	1.4
Saber	1.46	1.46	1.5

Tabla 3.10: Coeficientes de crecimiento nivel 3 - nivel 5 KEM: cociente entre media de ciclos nivel 5 y nivel 3.

En el caso de BIKE y Frodo, si bien el crecimiento en el número de ciclos es menor, esto se debe a que de la misma forma, el crecimiento del tamaño de sus atributos es significativamente menor (hasta un 30 % menos en los tres atributos).

En el caso de HQC, sus atributos aumentan un 60 % (a diferencia del paso del nivel 1 al nivel 3, que aumentaban un 100 %), que se traduce en un crecimiento del 85 % en el número de ciclos (a diferencia del paso del nivel 1 al nivel 3, que aumentaban entre un 100 % y un 126 %).

Por su lado, Kyber y Saber siguen siendo los que mejor escalan. Kyber siguiendo la aproximación lineal mencionada anteriormente (sobre todo en el algoritmo principal), y Saber empeorando un poco respecto del paso del nivel 1 al 3.

El caso de Classic McEliece

Aunque alcanzó la ronda 3 del concurso, al final no ha sido elegido como un algoritmo a estandarizar. No obstante, ha avanzado a una cuarta ronda, por lo que aún puede ser un candidato en la estandarización. Siendo el esquema más antiguo además de haber sido ampliamente estudiado, tiene un tamaño de texto cifrado mucho menor que el resto de algoritmos, pero su clave pública es mucho mayor en tamaño y además el tiempo de generación es alto. Debido a este problema, se ha decido separarlo de la comparación. CM ofrece 5 algoritmos distintos:

- **Classic-McEliece-348864** con un nivel de seguridad 1
- **Classic-McEliece-460896** con un nivel de seguridad 3
- **Classic-McEliece-6688128** con un nivel de seguridad 5
- **Classic-McEliece-6960119** con un nivel de seguridad 5
- **Classic-McEliece-8192128** con un nivel de seguridad 5

Las características de los algoritmos son las siguientes:

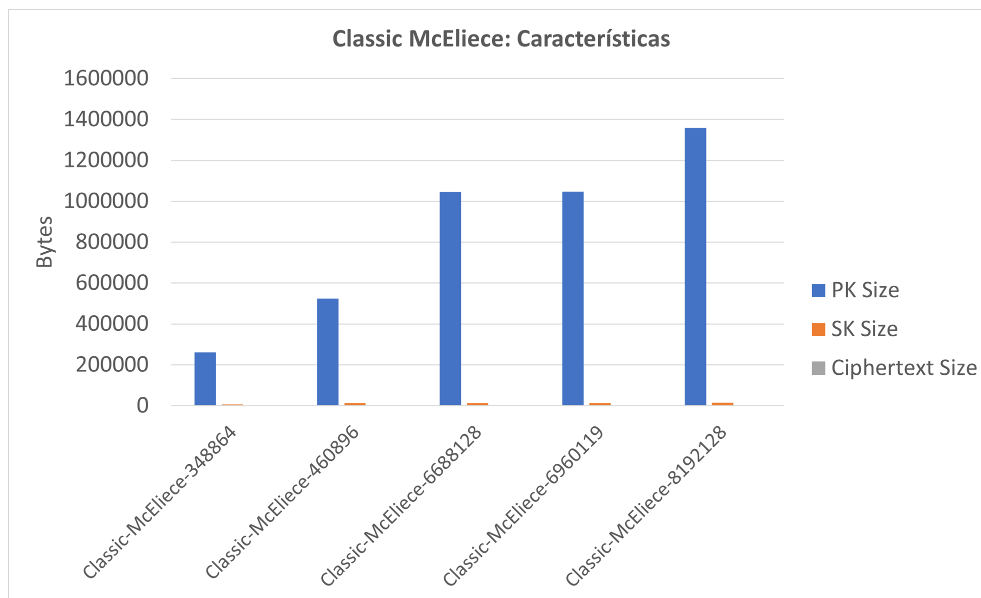

Figura 3.11: Tamaño en bytes de las etapas del algoritmo Classic McEliece.

En la figura se puede observar muy fácilmente lo que se ha mencionado anteriormente: la clave pública ocupa mucho tamaño en comparación con el resto de atributos. Así mismo, texto cifrado es tan pequeño que no aparece en la figura.

Algoritmo CM	PK Size (Bytes)	SK Size (Bytes)	Ciphertext Size (Bytes)
348864	261120	6452	128
460896	524160	13568	188
6688128	1044992	13892	240
6960119	1047319	13908	226
8192128	1357824	14080	240

Tabla 3.11: Tamaños en bytes del algoritmo Classic McEliece.

Los ciclos de generación se presentan por separado debido a su magnitud:

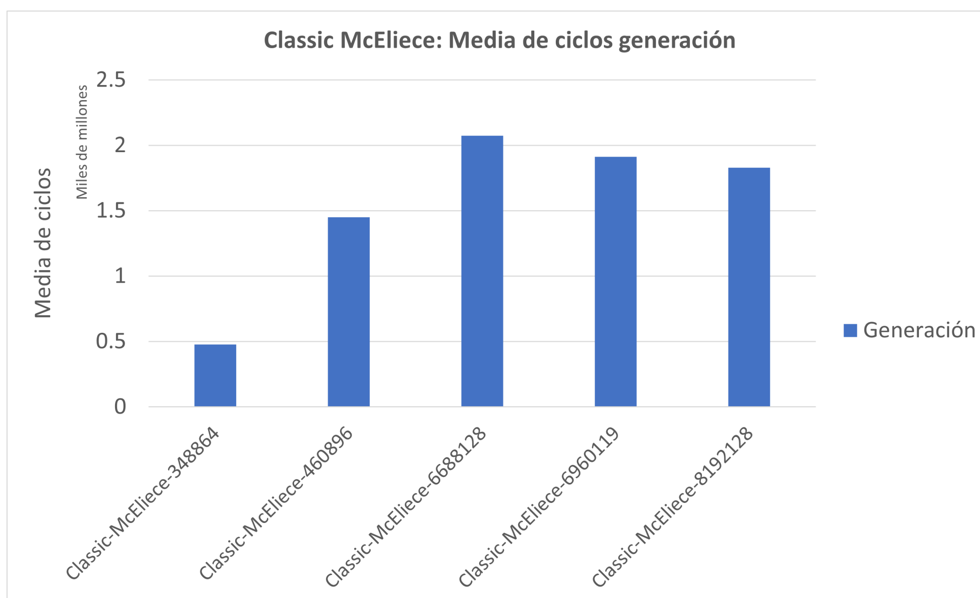

Figura 3.12: Gráfico media de ciclos generación algoritmo Classic McEliece.

La figura 3.12 se encuentra expresada en miles de millones. Por realizar una pequeña comparación, CM-348864, de nivel 1, tarda en generar las claves 478,6 millones de ciclos, mientras que FrodoKEM-1344-SHAKE, de nivel 5, que era el que más tardaba en generar las claves, tarda 14,5 millones de ciclos.

Llama la atención como a pesar de tener un tamaño de clave pública mayor, CM-8192128 tarda 246,4 millones de ciclos menos que CM-6688128, con un tamaño de clave pública mayor, lo cual en cuanto a cuestiones de rendimiento, lo hace el más viable de entre los algoritmos de nivel 5.

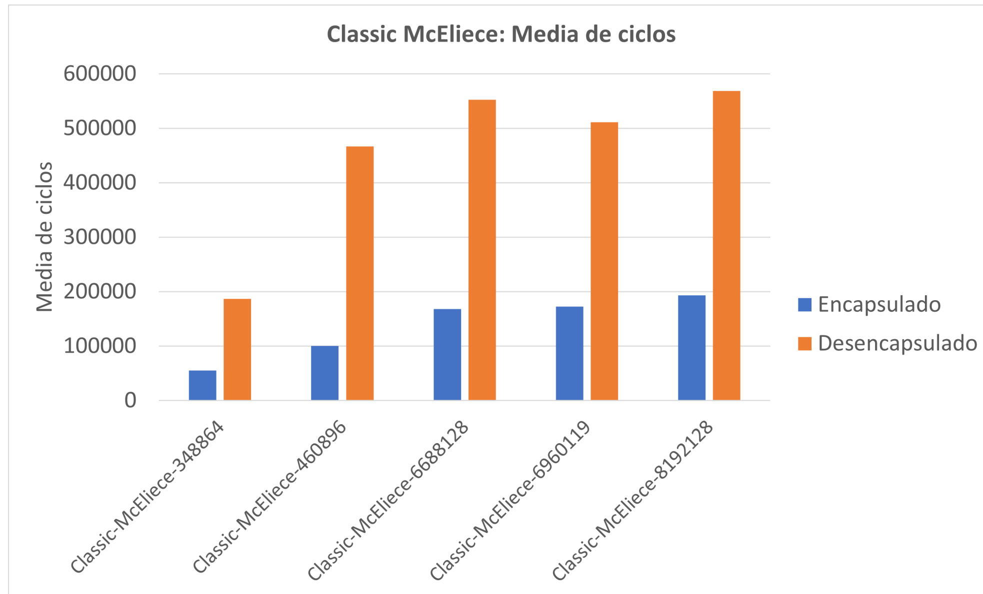

Figura 3.13: Gráfico media de ciclos algoritmo Classic McEliece.

Algoritmo CM	Generación	Encapsulado	Desencapsulado
348864	478640522	55422	186897
460896	1449194413	100218	467077
6688128	2075271100	167843	552320
6960119	1913772298	172866	511400
8192128	1828803082	193255	568568

Tabla 3.12: Media de ciclos algoritmo Classic McEliece.

En cuanto al encapsulado, parece que el pequeño tamaño del texto cifrado hace que sea uno de los más rápidos es esta fase. No llega a la rapidez de Kyber o Saber, pero tarda menos que HQC. Esta característica, sin embargo, no parece ayudar al desencapsulado de la misma manera. No obstante, sigue estando entre Kyber/Saber y HQC en cuanto a rendimiento en esta fase.

Estudio de consistencia

En la sección anterior se realiza un estudio del rendimiento de los algoritmos de encapsulado de claves. En esta sección se hace un breve repaso sobre todas las ejecuciones que han llevado a los resultados anteriores.

Esto sólo se realiza con los algoritmos de nivel 1, ya que con el resto de niveles se espera obtener resultados similares por algunos experimentos iniciales. Estos experimentos dieron las mismas diferencias pero de distinta magnitud.

Los resultados de BIKE son los siguientes:

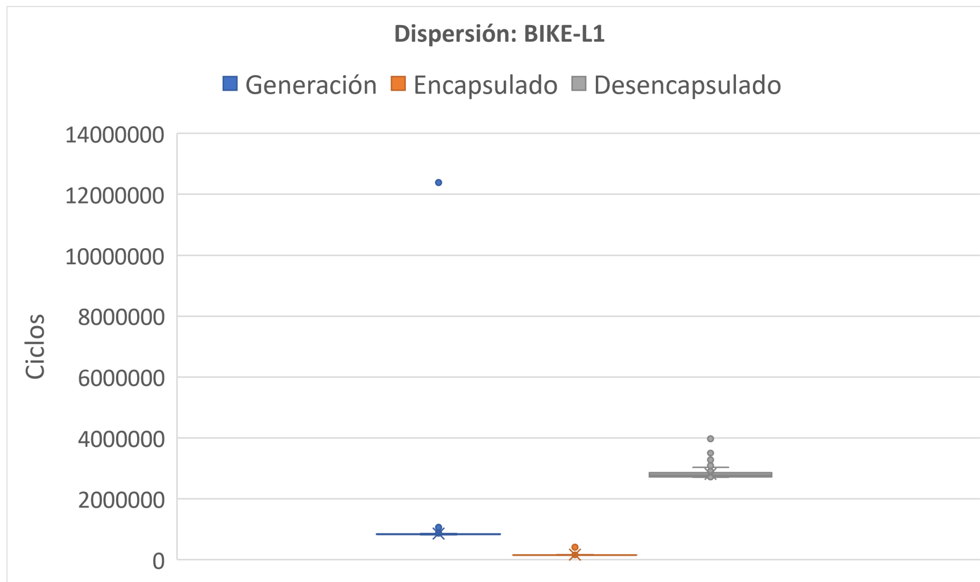

Figura 3.14: Gráfico consistencia algoritmo BIKE.

En la figura no se observa una gran dispersión de los valores en ninguna de las tres fases. El valor atípico de la generación se trata de la primera ejecución, que se ha incluido ya que forma parte de los resultados de la sección anterior. Este valor atípico se encuentra presente en la primera ejecución de la generación de la mayoría de los algoritmos.

En el caso de ambas versiones de Frodo (figuras 3.15 y 3.16) se observa una dispersión mayor. Llama la atención que aunque de media FrodoKEM-AES tiene un mejor rendimiento, FrodoKEM-SHAKE tiene una dispersión menor entre todas las ejecuciones.

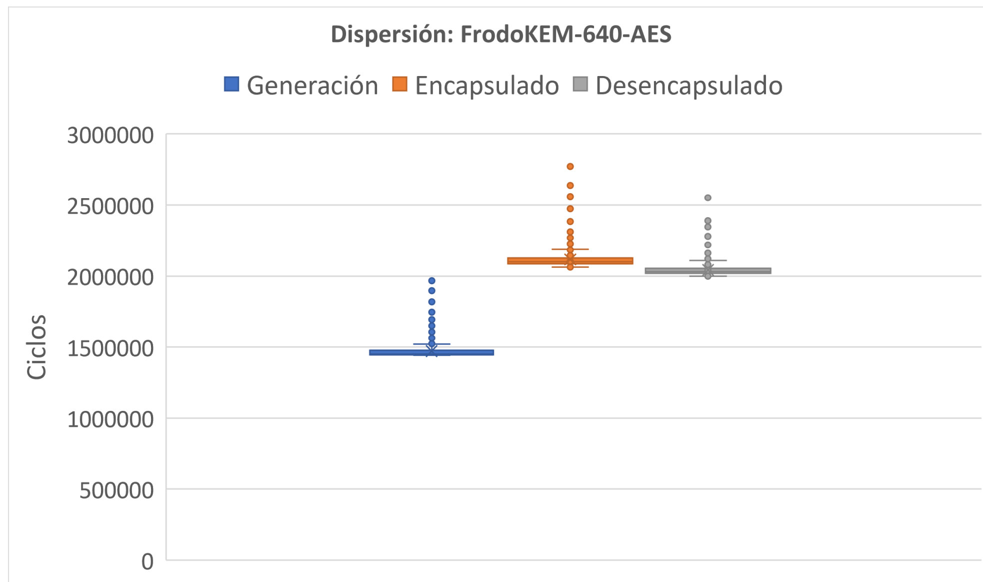

Figura 3.15: Gráfico consistencia algoritmo FrodoKEM-AES.

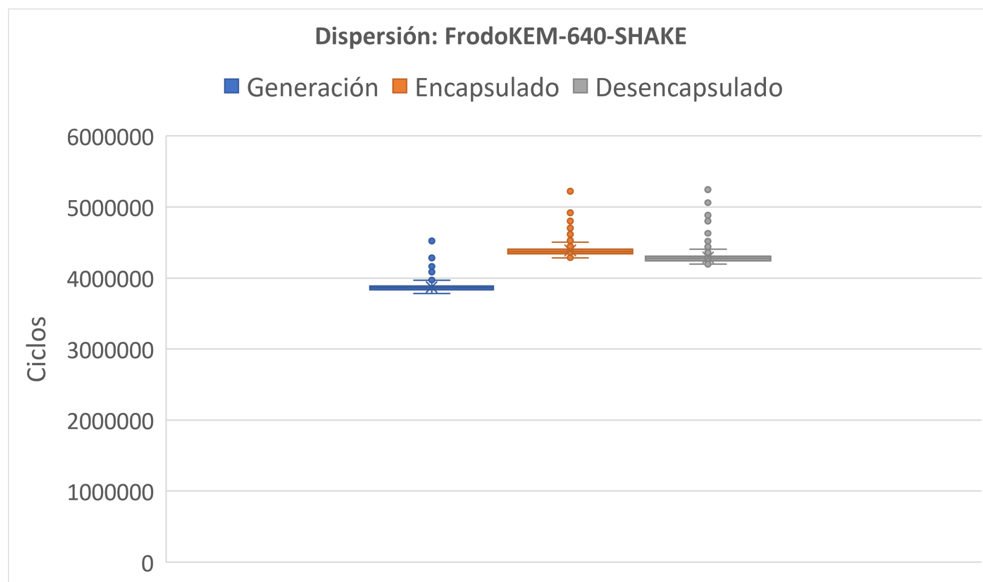

Figura 3.16: Gráfico consistencia algoritmo FrodoKEM-SHAKE.

La figura 3.17 corresponde al algoritmo HQC. De nuevo, se observa el valor atípico en la generación, pero en este caso, se observa un valor atípico en el encapsulado. En el caso del encapsulado corresponde, de nuevo, con la primera ejecución de la fase de encapsulado. Al igual que en BIKE, no se observa una gran dispersión en la mayoría de ejecuciones.

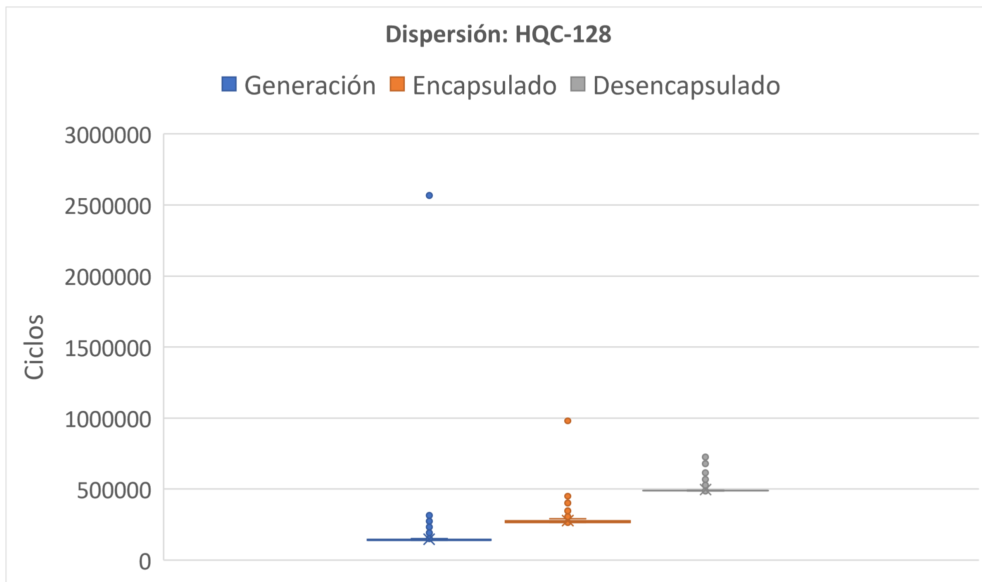

Figura 3.17: Gráfico consistencia algoritmo HQC.

Las siguientes figuras corresponden a los algoritmos de Kyber y Kyber-90's. En el caso de Kyber (figura 3.18) se observa una dispersión mínima en todas sus fases, mientras que Kyber-90's (figura 3.19) presenta una gran dispersión en sus valores, a pesar de que en media, tardaba menos en su ejecución.

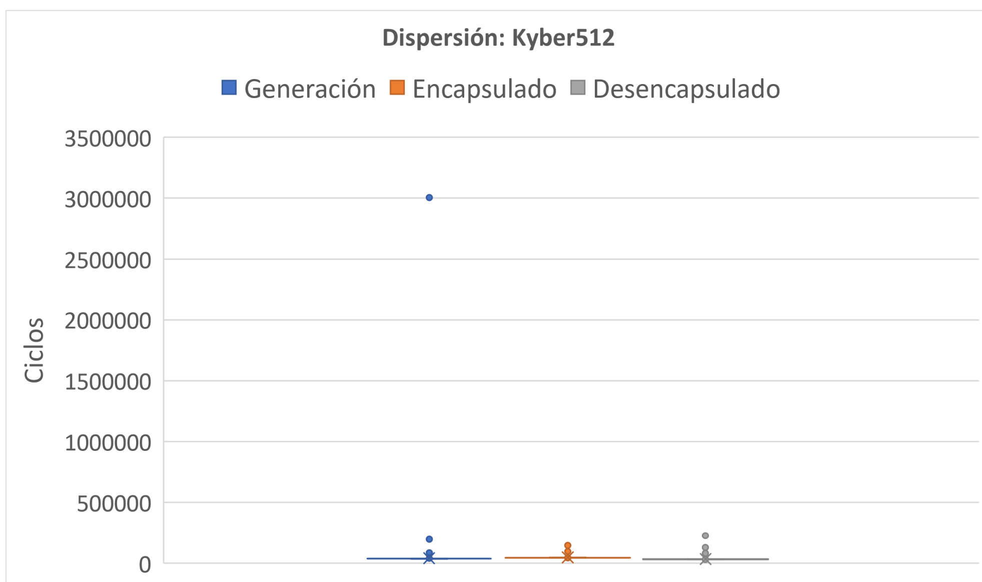

Figura 3.18: Gráfico consistencia algoritmo Kyber.

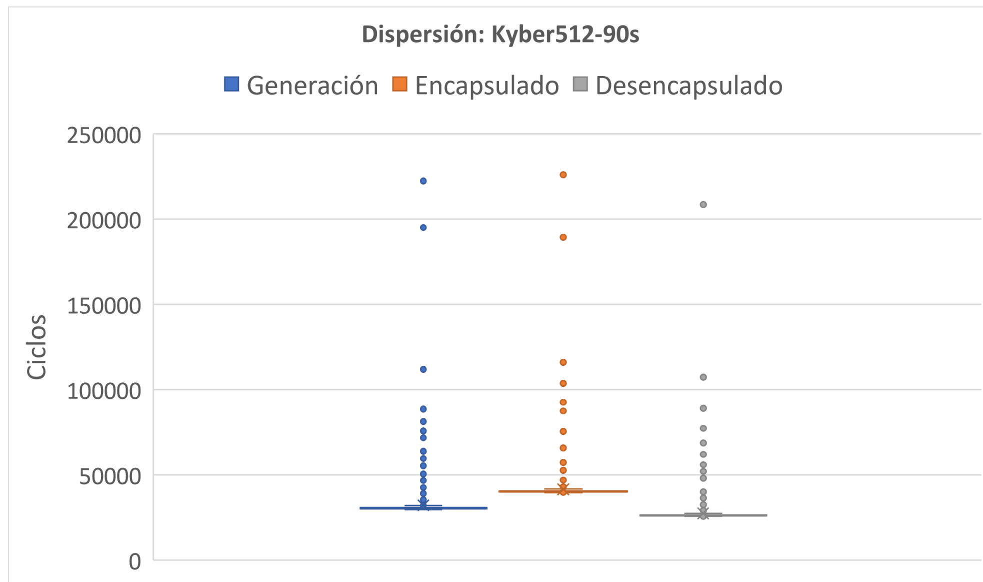

Figura 3.19: Gráfico consistencia algoritmo Kyber-90s.

Por último, la figura 3.20 corresponde a Classic McEliece. De nuevo, en el caso de este algoritmo los ciclos se encuentran en mil millones de ciclos debido a la magnitud de sus números. En la generación es donde mayor dispersión se muestra, además de ser la fase en la que más tarda. En el caso del encapsulado y el desencapsulado no se observa prácticamente dispersión, probablemente debido a la escala en la que están representados los datos.

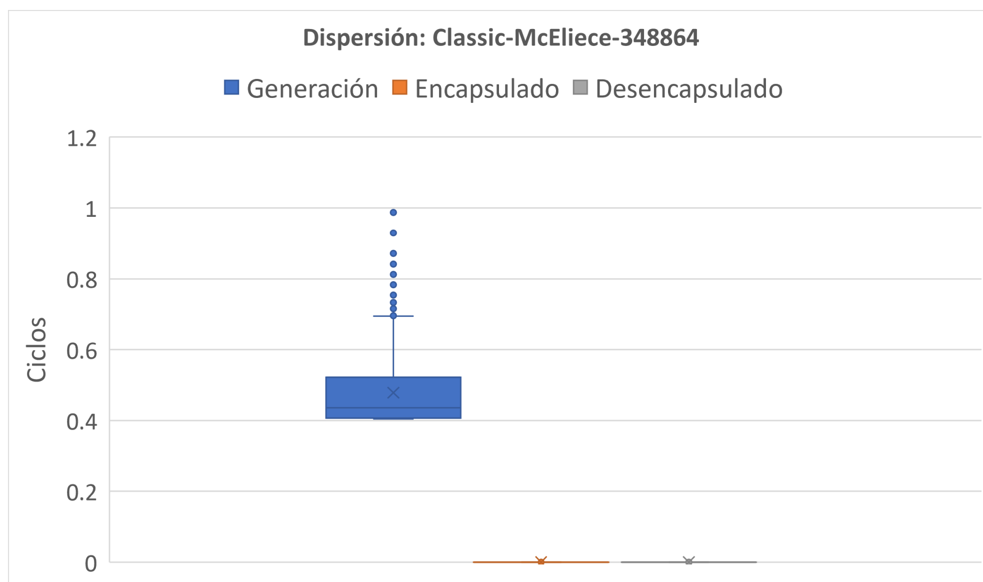

Figura 3.20: Gráfico consistencia algoritmo Classic McEliece.

En conclusión: la mayoría de algoritmos son consistentes en cuanto a las ejecuciones realizadas, tardando más o menos lo mismo en ejecutarse todas las veces,

siendo Classic McEliece en su fase de generación de claves el que más dispersión presenta.

3.2.2. Algoritmos de firma digital

En esta sección se analizarán los resultados de ejecutar los distintos algoritmos de firma digital disponibles en la librería, más los agregados. Se seguirán las mismas pautas que en los algoritmos KEM. Debido al gran volumen de algoritmos, el caso de SPHINCS se tratará por separado.

Asi mismo, Dilithium2 queda fuera de la comparación, al ser el único algoritmo con nivel de seguridad 2. También, debido a la diferencia de algoritmos entre un nivel y otro, no se realizará una comparación del crecimiento de los algoritmos en los distintos niveles.

Nivel de seguridad 1

Las características de los algoritmos de nivel 1 son las siguientes:

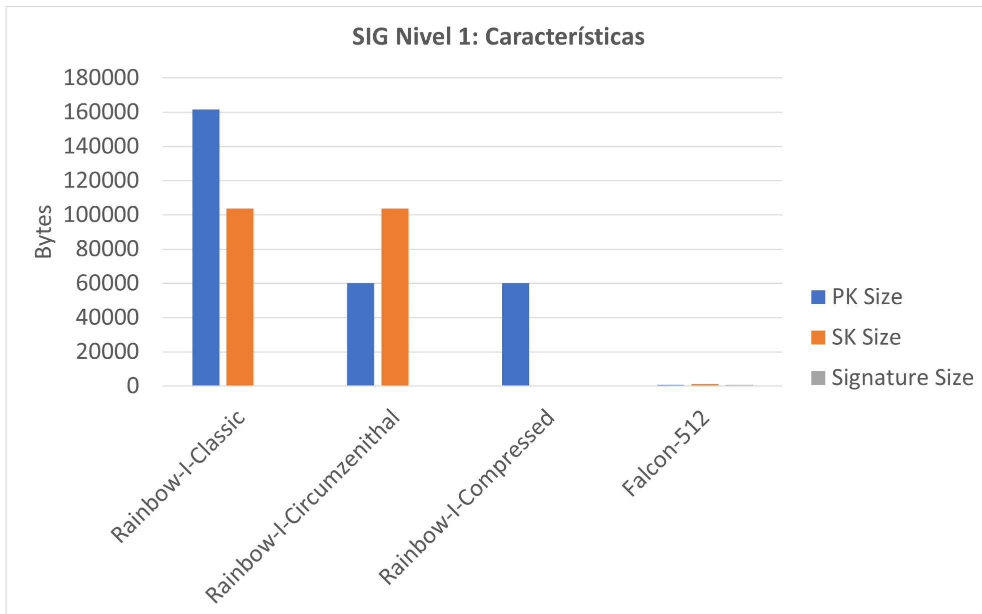

Figura 3.21: Tamaño en bytes de algoritmos de firma de nivel 1

Algoritmo	PK Size	SK Size	Signature Size
Rainbow-I-Classic	161600	103648	66
Rainbow-I-Circumzenithal	60192	103648	66
Rainbow-I-Compressed	60192	64	66
Falcon-512	897	1281	690

Tabla 3.13: Tamaño en bytes de algoritmos de firma de nivel 1

En todos los algoritmos el tamaño de la firma es mínimo. En el caso de Rainbow, es el tamaño de clave pública y privada lo que varía de una versión a otra. Los resultados de la ejecución son los siguientes (en cien millones de ciclos):

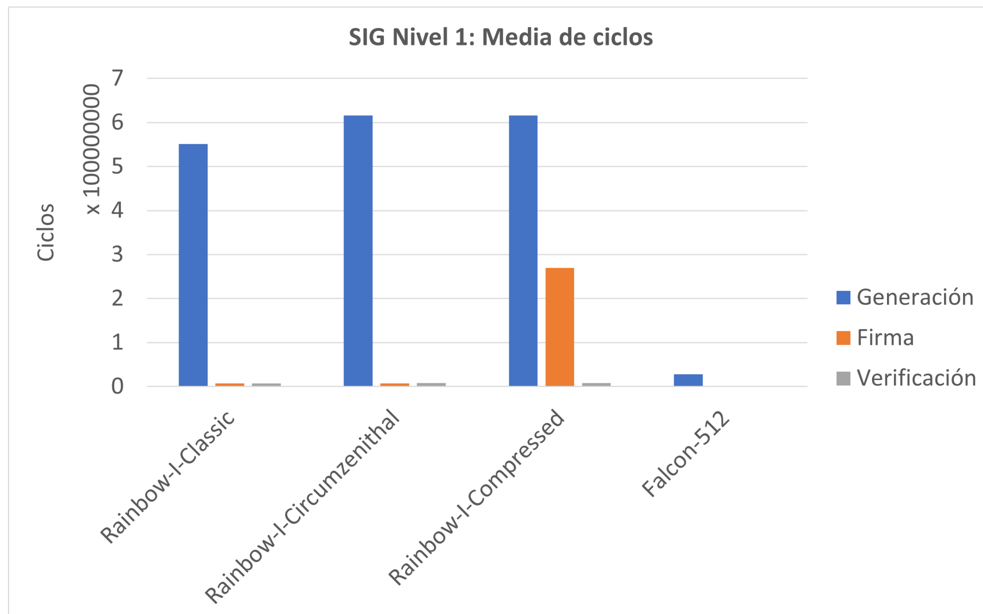

Figura 3.22: Gráfico media de ciclos de algoritmos de firma de nivel 1

Algoritmo	Generación	Firma	Verificación
Rainbow-I-Classic	551594377	7055017	7430319
Rainbow-I-Circumzenithal	615943180	7063991	8066440
Rainbow-I-Compressed	615853020	269689926	8061464
Falcon-512	28146175	1026286	180950

Tabla 3.14: Media de ciclos de algoritmos de firma de nivel 1

Rainbow tiene un tiempo de generación de claves muy alto en comparación con Falcon en cualquiera de sus tres variables. Además, en su versión *Compressed*, tarda mucho en realizar la firma, en comparación con sus otras dos versiones. Por su lado, Falcon requiere menos ciclos en cualquiera de las tres fases.

Nivel de seguridad 3

Las características de los algoritmos de nivel 3 son las siguientes:

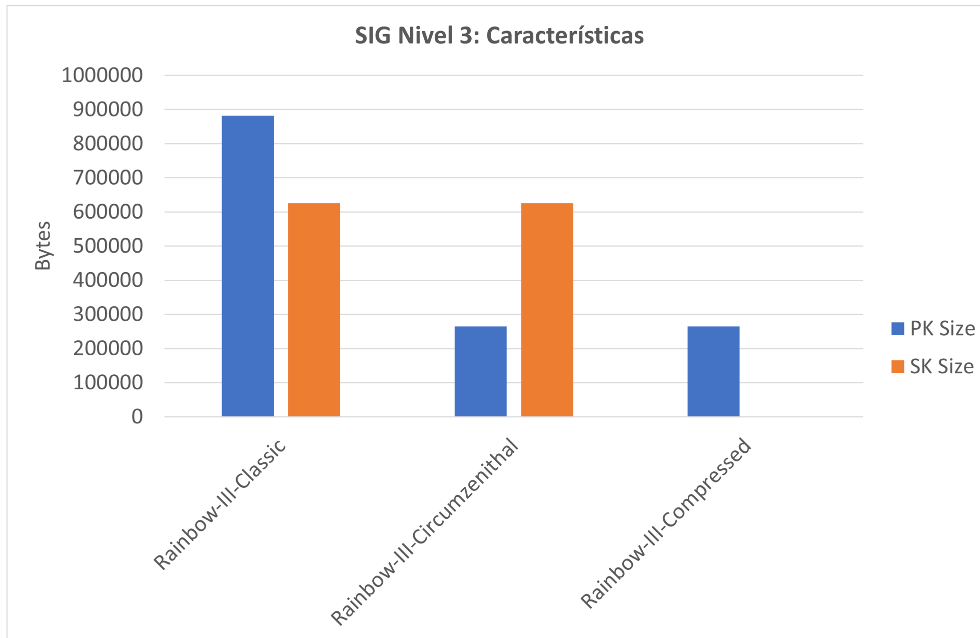

Figura 3.23: Tamaño en bytes de algoritmos de firma de nivel 3

Algoritmo	PK Size	SK Size	Signature Size
Rainbow-III-Classic	882080	626048	164
Rainbow-III-Circumzenithal	264608	626048	164
Rainbow-III-Compressed	264608	64	164
Dilithium3	1952	4000	3293
Dilithium3-AES	1952	4000	3293

Tabla 3.15: Tamaño en bytes de firma de nivel 3

Rainbow aumenta el tamaño de todos sus componentes, menos de la clave privada en su versión *Compressed*, que se mantiene respecto del algoritmo de nivel 1. Dilithium tiene un tamaño muy pequeño en todos sus atributos en comparación con Rainbow , exceptuando el tamaño de clave privada de *Compressed*. Los resultados de la ejecución se pueden ver en la figura 3.24.

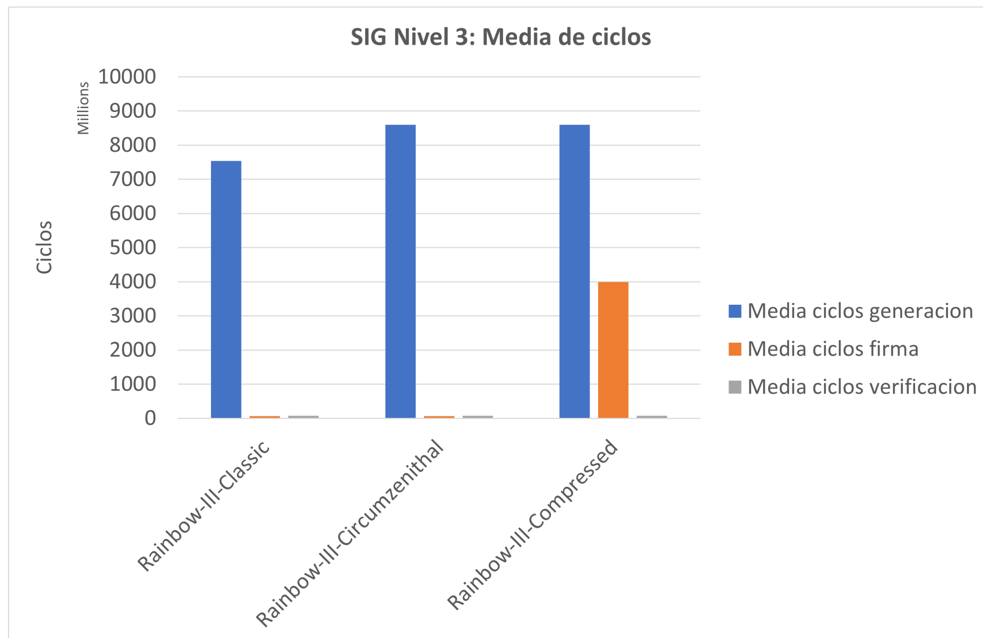

Figura 3.24: Gráfico media de ciclos de algoritmos de firma de nivel 3

Algoritmo	Generación	Firma	Verificación
Rainbow-III-Classic	7535079342	67563568	69781899
Rainbow-III-Circumzenithal	8595320294	67586792	73422835
Rainbow-III-Compressed	8599177377	3995382320	73452455
Dilithium3	195890	522641	181619
Dilithium3-AES	110096	329674	108653

Tabla 3.16: Media de ciclos de algoritmos de firma de nivel 3

Al igual que pasa con Kyber en los algoritmos KEM, Dilithium tarda muy poco en ejecutarse en comparación con los demás. Mientras, Rainbow aumenta demasiado el número de ciclos.

Nivel de seguridad 5

Las características de los algoritmos de nivel 5 son las siguientes:

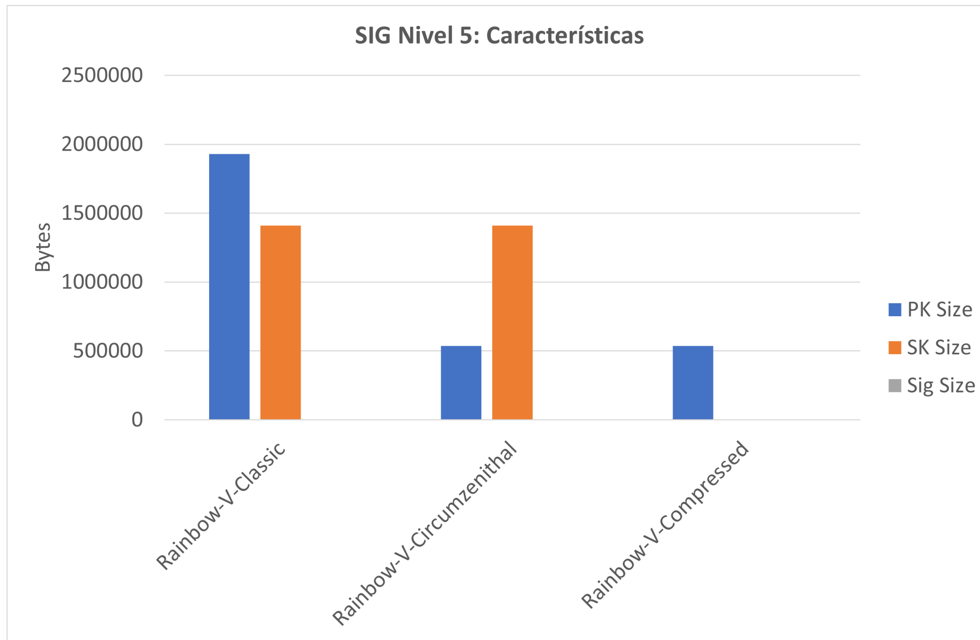

Figura 3.25: Gráfico características de algoritmos de firma de nivel 3

Algoritmo	PK Size	SK Size	Signature Size
Rainbow-IV-Classic	1930600	1408736	212
Rainbow-IV-Circumzenithal	536136	1408736	212
Rainbow-IV-Compressed	536136	64	212
Dilithium5	2592	4864	4595
Dilithium5-AES	2592	4864	4595
Falcon-1024	1793	2305	1330

Tabla 3.17: Características de algoritmos de firma de nivel 5

Tanto Dilithium como Falcon tienen tamaños de clave muy pequeños en comparación con Rainbow (excluyendo la versión *Compressed*). No obstante, el tamaño de la firma es mucho menor en Rainbow en comparación con Dilithium y Falcon.

La tabla 3.18 muestra los resultados de la ejecución. De nuevo y como en los niveles anteriores, Dilithium y Falcon tardan muy poco en las tres fases, mientras que Rainbow tarda demasiado en generar las claves.

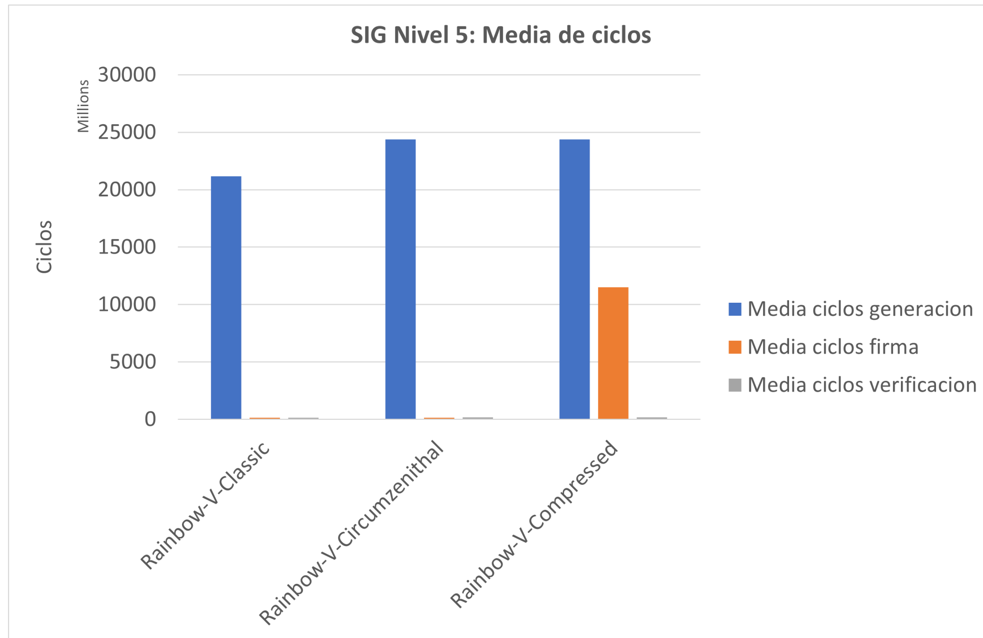

Figura 3.26: Gráfico media de ciclos de algoritmos de firma de nivel 5

Algoritmo	Generación	Firma	Verificación
Rainbow-IV-Classic	21166582680	148941889	152162128
Rainbow-IV-Circumzenithal	24366122583	148798833	160442673
Rainbow-IV-Compressed	24369767186	11511037035	160329869
Dilithium5	306077	580400	289547
Dilithium5-AES	168424	386924	170355
Falcon-1024	82268715	2071864	344583

Tabla 3.18: Media de ciclos de algoritmos de firma de nivel 5

Algoritmo de firma SPHINCS$^+$

El algoritmo de SPHINCS$^+$ ofrece múltiples variantes. Por un lado las distinciones entre *robust* y *simple*. La primera se corresponde con la versión enviada en la primera ronda del concurso y todas las mejoras en seguridad posteriores, mientras que la segunda se deriva de un estudio teórico. Por el otro lado, las versiones difieren en la función hash utilizada: Haraka, SHA256 y SHAKE256. Las versiones s y f difieren en el valor de sus parámetros internos [48] [55].

Las características de los distintos algoritmos se encuentran en la figura 3.27 y en la tabla 3.19. Se ha excluido de la figura los tamaños de clave ya que, a parte de no variar entre versiones f/s, son muy pequeños en comparación con el tamaño de firma. La variación está en el tamaño de firma, con la versión s ofreciendo un tamaño de firma menor. Debido a que las simulaciones ofrecen resultados similares, las figuras y tablas correspondientes a las distintas variaciones se encuentran

en el Apéndice B para su consulta ofreciéndose aquí un resumen.

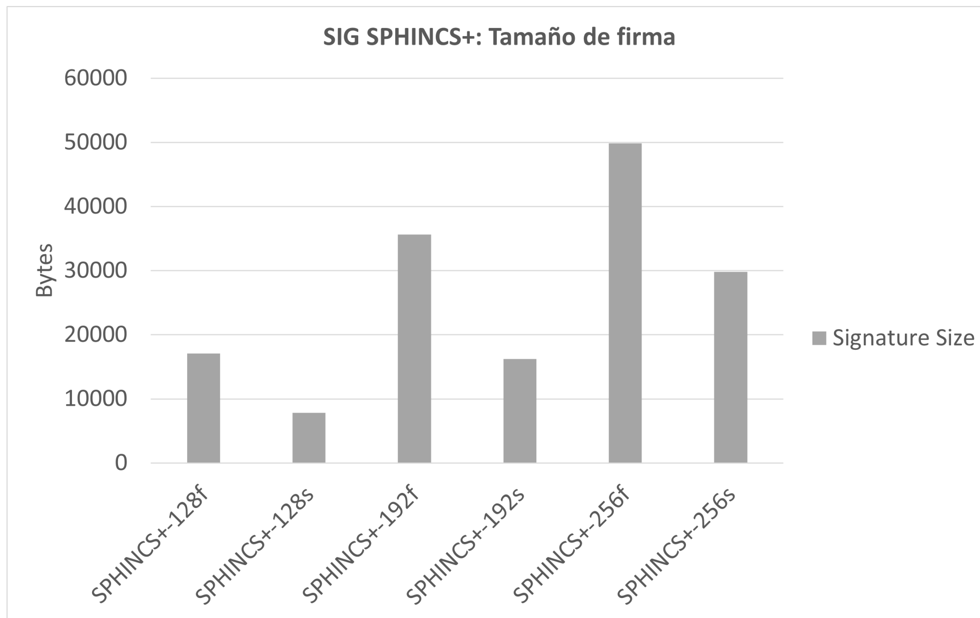

Figura 3.27: Gráfico tamaño de firma de algoritmos de SPHINCS$^+$

Algoritmo	PK Size	SK Size	Signature Size
SPHINCS+-128f	32	64	17088
SPHINCS+-128s	32	64	7856
SPHINCS+-192f	48	96	35664
SPHINCS+-192s	48	96	16224
SPHINCS+-256f	64	128	49856
SPHINCS+-256s	64	128	29792

Tabla 3.19: Tamaño en bytes de algoritmo de firma SPHINCS$^+$

Debido a las múltiples variantes, la comparación de rendimiento se ha hecho en dos fases: La primera fase compara los algoritmos de un mismo nivel de seguridad y la misma función hash. La segunda fase compara los que mejor rinden de la fase anterior y las distintas función hash para ver cuál es el que mejor rinde de todos.

Comenzando con la primera fase, la ejecuciones de SPHINCS$^+$-128-Haraka se puede observar que ambas versiones f tardan tan poco en la generación y en la verificación que no aparecen en la figura, las versiones s tardan demasiado en la firma y en la generación (ver tabla B.1). Dentro de esta comparación, *la mejor parada es f-simple*. Se puede ampliar esta conclusión en la figura B.2 para SPHINCS$^+$-192-Haraka y SPHINCS$^+$-256-Haraka (ver figura B.3). Respecto a de SHA256 y SHAKE256[2] se considera que el mejor rendimiento lo ofrece la versión

[2] Algoritmo de Hash SHAKE dando un hash de 256 bits

con SHA256 y algoritmo f-simple para cualquier versión (para más información ver tabla B.4).

Comenzando con la segunda fase dentro de los algoritmos de nivel 1, se puede observar como Haraka es el que mejor rendimiento ofrece en general. Es más notorio en el proceso de firma, con una diferencia de 40 millones de ciclos entre Haraka y SHA256, y de 80 millones de ciclos entre Haraka y SHAKE256. Comparando entre SPHINCS$^+$-192 y SPHINCS$^+$-256, respectivamente la versión con Haraka es más rápida y escalable y crecer en menor medida en comparación. Haraka es una función hash creada específicamente para escenarios post-cuánticos [56], lo que explica su mejor rendimiento.

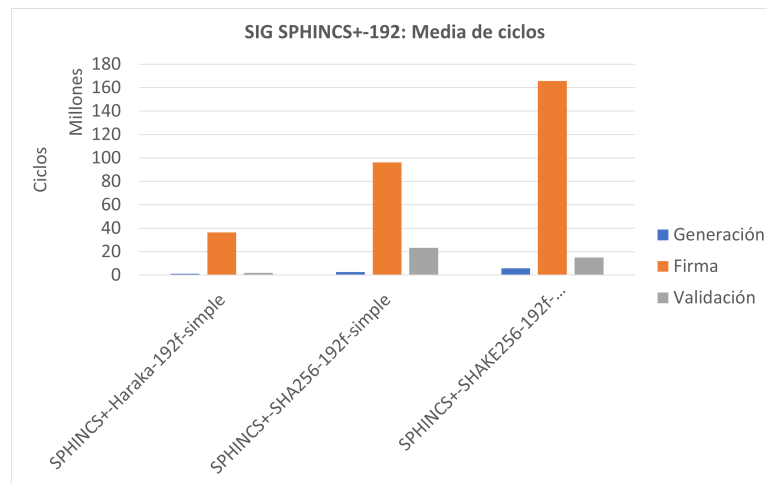

Figura 3.28: Gráfico media de ciclos de algoritmos de SPHINCS$^+$-192

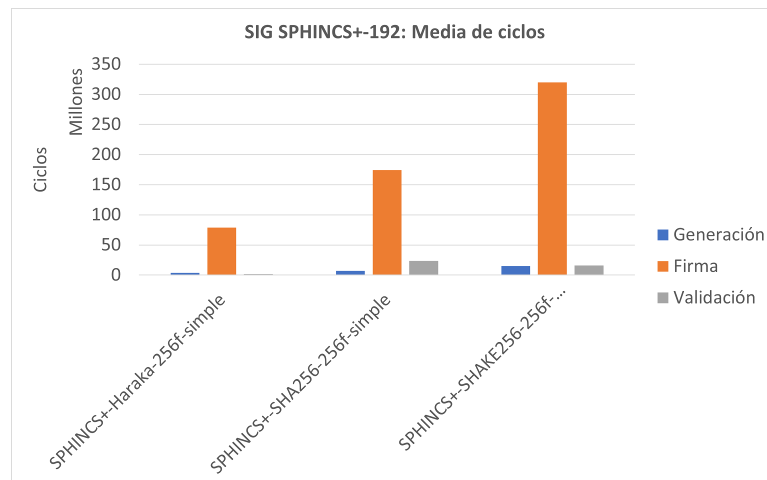

Figura 3.29: Gráfico media de ciclos de algoritmos de SPHINCS$^+$-256

4 Conclusiones y trabajos futuros

Respecto a los algoritmos de KEM, Kyber y Saber son los claros ganadores. Quitando los aspectos a nivel de hardware, en cuestiones de velocidad y de memoria, estos algoritmos son utilizables hasta por dispositivos con prestaciones más bajas, como pueden ser los dispositivos móviles. Aunque son parecidos, el NIST escogió a Kyber por encima de Saber, debido a que no tenía sentido estandarizar dos algoritmos que se basaban en lo mismo [57].

BIKE tiene un rendimiento aceptable en cuanto al encapsulado se refiere, mientras que a medida que escala, el rendimiento de la generación y el desencapsulado empeora notablemente. HQC no está a la altura de Kyber o Saber en cuanto a rendimiento se refiere, pero aun así tiene un rendimiento aceptable. En cuanto a la escalabilidad, aunque tarde el doble en el paso de un nivel a otro, sigue siendo un número de ciclos dentro de un rango aceptable. Ambos algoritmos, Bike y HQC, no tienen tampoco un gran costo en cuanto a recursos se refiere. Frodo es el algoritmo que peor rendimiento demuestra respecto a los demás, sobre todo en su variante con SHAKE. No obstante, en cuanto a escalabilidad, demuestra ser consistente al ir aumentando el tamaño de sus componentes, siempre y cuando se utilice la versión con AES.

Si bien Classic McEliece es un candidato a ser estandarizado por el NIST como alternativa a Kyber una vez termine la ronda 4, como se explica en [33], Classic McEliece no es un buen candidato para ser implementado ampliamente. Esto se refleja en cuestiones como la cantidad de memoria que ocupa su clave pública, así como el tiempo que tarda en generar las claves. No obstante, en situaciones donde se pueda reutilizar la clave o donde no haya que transmitirla, Classic McEliece

podría ser mejor candidato que Kyber debido a su robusta seguridad.

En el caso de esquemas para firmas, Rainbow es el algoritmo más rápido a la hora de firmar, pero tanto el tamaño de claves como el tiempo que tarda en generarlas es demasiado alto en comparación con los demás. Además, en mitad del concurso se encontraron ataques que afectaban directamente a la seguridad de este algoritmo, por lo que no pasó de la tercera ronda [57].

Dilithium fue uno de los algoritmos escogidos para su estandarización. Basado en retículos, al igual que Kyber, es de los algoritmos más rápidos, sobretodo al usar AES internamente. Falcon es un algoritmo basado en retículos que utiliza un paradigma distinto a Dilithium. Esto le hace muy rápido, pero no llega a alcanzar a Dilithium.

SPHINCS$^+$ es el único algoritmo basado en hash que ha sido estandarizado. Esto se realizó para tener alternativas a los algoritmos basados en retículos. El algoritmo parte con la ventaja de que, a parte de ser bastante rápido, se puede utilizar con diversos algoritmos de hash: esto lo vuelve un candidato versátil.

Como trabajos futuros, se plantean las siguientes opciones:

- Profundizar en el lado matemático del trabajo, estudiando con detalle el por qué los problemas matemáticos planteados son elegibles frente a un adversario cuántico.
- Estudiar las implicaciones de algunos algoritmos a nivel de hardware: ver si alguna modificación en circuitos lógicos podría implicar una mejora en el rendimiento de los mismos.
- Estudio de otros algoritmos propuestos. Que no hayan sido elegidos no implica que no puedan plantear nuevas ideas o ser utilizados en otros ámbitos. Por ejemplo, NTRU, que no ha sido estudiado en este TFG, fue elegido para utilizarse junto con OpenSSL. Más información del proyecto en OpenSSLNTRU
- Implementación de algún tipo de proyecto cuyo cifrado se base en el uso de alguno de estos algoritmos. Debido a que se encuentra fuera del alcance de este trabajo, no he podido realizar una implementación práctica del uso de alguno de estos algoritmos. La librería liboqs ofrece una versión de OpenSSL que permite este tipo de operaciones con algoritmos de criptografía post-cuántica en OQS OpenSSL. También podemos encontrar lo mismo con OpenSSH: OQS OpenSSH.

Bibliografía

[1] Y. F. Utf-8, a transformation format of iso 10646. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3629

[2] R. L. Rivest, M. J. Robshaw, R. Sidney, and Y. L. Yin, "The rc6tm block cipher," in *First advanced encryption standard (AES) conference*, 1998, p. 16.

[3] D. J. Bernstein, "The salsa20 family of stream ciphers," *New stream cipher designs: the eSTREAM finalists*, pp. 84–97, 2008.

[4] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," National Institute of Standards and Technology, Tech. Rep., March 2019.

[5] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray Jr, "Advanced encryption standard (aes)," 2001.

[6] J.-P. Aumasson, *Serious cryptography: a practical introduction to modern encryption.* No Starch Press, 2017.

[7] R. Rivest, "Rfc1321: The md5 message-digest algorithm," 1992.

[8] H. Dobbertin, "Cryptanalysis of md5 compress," *rump session of Eurocrypt*, vol. 96, pp. 71–82, 1996.

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

[10] G. Tamvada and S. Celi, "Deep dive into a post-quantum key encapsulation algorithm," 2022. [Online]. Available: https://blog.cloudflare.com/post-quantum-key-encapsulation/

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[12] M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[13] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *Journal of cryptology*, vol. 14, pp. 255–293, 2001.

[14] A. K. Lenstra, "Key lengths," 2004.

[15] D. Giry and P. Bullens. Keylenght - cryptographic key length recommendation. [Online]. Available: https://www.keylength.com/en/2/

[16] R. Laboratories, "Announcement of rsa factoring challenge"," 1991.

[17] ——, "Rsa challenge list," 1994.

[18] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[19] C. F. Kerry and P. D. Gallagher, "Digital signature standard (dss)," *FIPS PUB*, pp. 186–4, 2013.

[20] C. Soto Anaya, "Computación cuántica aplicada en criptografía," 2018.

[21] R. Carpio López and A. López Montes, "Introducción a la computación cuántica," 2022.

[22] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[23] C. Gidney, "Factoring with n+ 2 clean qubits and n-1 dirty qubits," *arXiv preprint arXiv:1706.07884*, 2017.

[24] O. Regev, "Lattice-based cryptography," in *Annual International Cryptology Conference*. Springer, 2006, pp. 131–141.

[25] J. H. Silverman, "An introduction to lattices, lattice reduction, and lattice-based cryptography," 2020.

[26] O. Regev, "The learning with errors problem," *Invited survey in CCC*, vol. 7, no. 30, p. 11, 2010.

[27] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[28] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques.* Springer, 1987, pp. 369–378.

[29] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standarization process," 2016.

[30] ——. Post-quantum cryptography standardization. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[31] ——. Post-quantum cryptography standardization: Round 1 submissions. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions

[32] ——. Post-quantum cryptography standardization: Round 2 submissions. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions

[33] Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates. [Online]. Available: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

[34] P. Schwabe. Crystals, cryptographic suite for algebraic lattices: Kyber. [Online]. Available: https://pq-crystals.org/kyber/index.shtml

[35] D. J. Bernstein. Classic mceliece. [Online]. Available: https://classic.mceliece.org/

[36] M. García Luis *et al.*, "Criptosistema de mceliece," 2021.

[37] J.-P. D'Anvers. Saber: Lwr-based kem. [Online]. Available: https://www.esat.kuleuven.be/cosic/pqcrypto/saber/index.html

[38] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with rounding, revisited: New reduction, properties and applications," in *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I.* Springer, 2013, pp. 57–74.

[39] J. Hoffstein, J. Pipher, and J. H. Silverman, "Ntru: A ring-based public key cryptosystem," in *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*. Springer, 2998, pp. 267–288.

[40] E. Alkim, J. W. Bos, L. Ducas, P. Longa, I. Mironov, M. Naehrig, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila, "Frodokem, learning with errors key encapsulation."

[41] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, J. Richter-Brockmann, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor, "Bike: Bit flipping key encapsulation," 2022.

[42] 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)," *NIST PQC Round*, vol. 2, no. 4, p. 13, 2018.

[43] NIST. Post-quantum cryptography standardization: Round 2 submissions. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

[44] P. Schwabe. Crystals, cryptographic suite for algebraic lattices: Dilithium. [Online]. Available: https://pq-crystals.org/dilithium/index.shtml

[45] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," Cryptology ePrint Archive, Paper 2007/432, 2007, https://eprint.iacr.org/2007/432. [Online]. Available: https://eprint.iacr.org/2007/432

[46] T. Prest. Falcon. [Online]. Available: https://falcon-sign.info/

[47] C. Peikert, "Lattice-based cryptography: Short integer solution (sis) and learning with errors (lwe)," *Georgia Institute of Technology*, 2013.

[48] D. J. Bernstein. Sphincs+ stateless hash-based signatures. [Online]. Available: https://sphincs.org/

[49] M.-s. Chen, J. Ding, M. Kannwischer, J. Patarin, D. Petzoldt, Albretch amd Schmidt, and B.-Y. Yang. Pqrainbow. [Online]. Available: https://www.pqcrainbow.org/

[50] Open quantum safe. [Online]. Available: https://openquantumsafe.org/

[51] Github liboqs. [Online]. Available: https://github.com/open-quantum-safe/liboqs

[52] Github pqclean. [Online]. Available: https://github.com/PQClean/PQClean

[53] Fletch. (2021) Rdtsc the only way to benchmark. [Online]. Available: https://medium.com/geekculture/rdtsc-the-only-way-to-benchmark-fc84562ef734

[54] M. Á. González de la Torre, L. Hernández Encinas, and J. I. Sánchez García, "Comparative analysis of lattice-based post-quantum cryptosystems," 2022.

[55] J. P. Aumasson, D. J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdaf, A. Hülsing, P. Kampanakis, S. Köbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, and B. Westerbaan, "Sphincs+: Submission to the nist post-quantum project, v3.1," 2022.

[56] S. Kölbl, M. M. Lauridsen, F. Mendel, and C. Rechberger, "Haraka v2–efficient short-input hashing for post-quantum applications," *IACR Transactions on Symmetric Cryptology*, pp. 1–29, 2016.

[57] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the third round of the nist post-quantum cryptography standardization process," 2022.

Apéndices

Tablas de algoritmos

Se adjuntan un resumen de los algoritmos utilizados con su nivel de seguridad

Nombre del algoritmo	Nivel de seguridad
BIKE-L1	1
BIKE-L3	3
BIKE-L5	5
Classic-McEliece-348864	1
Classic-McEliece-460896	3
Classic-McEliece-6688128	5
Classic-McEliece-6960119	5
Classic-McEliece-8192128	5
FrodoKEM-640-AES	1
FrodoKEM-640-SHAKE	1
FrodoKEM-976-AES	3
FrodoKEM-976-SHAKE	3
FrodoKEM-1344-AES	5
FrodoKEM-1344-SHAKE	5
HQC-128	1
HQC-192	3
HQC-256	5
Kyber512	1
Kyber512-90s	1
Kyber768	3
Kyber768-90s	3
Kyber1024	5
Kyber1024-90s	5
LightSaber-KEM	1
Saber-KEM	3
FireSaber-KEM	5

Tabla A.1: Tabla de algoritmos KEM

Nombre del algoritmo	Nivel de seguridad
Rainbow-I-Classic	1
Rainbow-I-Circumzenithal	1
Rainbow-I-Compressed	1
Rainbow-III-Classic	3
Rainbow-III-Circumzenithal	3
Rainbow-III-Compressed	3
Rainbow-V-Classic	5
Rainbow-V-Circumzenithal	5
Rainbow-V-Compressed	5
Dilithium3	3
Dilithium5	5
Dilithium3-AES	3
Dilithium5-AES	5
Falcon-512	1
Falcon-1024	5
SPHINCS+-Haraka-128f-robust	1
SPHINCS+-Haraka-128f-simple	1
SPHINCS+-Haraka-128s-robust	1
SPHINCS+-Haraka-128s-simple	1
SPHINCS+-Haraka-192f-robust	3
SPHINCS+-Haraka-192f-simple	3
SPHINCS+-Haraka-192s-robust	3
SPHINCS+-Haraka-192s-simple	3
SPHINCS+-Haraka-256f-robust	5
SPHINCS+-Haraka-256f-simple	5
SPHINCS+-Haraka-256s-robust	5
SPHINCS+-Haraka-256s-simple	5
SPHINCS+-SHA256-128f-robust	1
SPHINCS+-SHA256-128f-simple	1
SPHINCS+-SHA256-128s-robust	1
SPHINCS+-SHA256-128s-simple	1
SPHINCS+-SHA256-192f-robust	3
SPHINCS+-SHA256-192f-simple	3
SPHINCS+-SHA256-192s-robust	3
SPHINCS+-SHA256-192s-simple	3
SPHINCS+-SHA256-256f-robust	5
SPHINCS+-SHA256-256f-simple	5
SPHINCS+-SHA256-256s-robust	5
SPHINCS+-SHA256-256s-simple	5

Tabla A.2: Tabla de algoritmos de firma digital 1

Nombre del algoritmo	Nivel de seguridad
SPHINCS+-SHAKE256-128f-robust	1
SPHINCS+-SHAKE256-128f-simple	1
SPHINCS+-SHAKE256-128s-robust	1
SPHINCS+-SHAKE256-128s-simple	1
SPHINCS+-SHAKE256-192f-robust	3
SPHINCS+-SHAKE256-192f-simple	3
SPHINCS+-SHAKE256-192s-robust	3
SPHINCS+-SHAKE256-192s-simple	3
SPHINCS+-SHAKE256-256f-robust	5
SPHINCS+-SHAKE256-256f-simple	5
SPHINCS+-SHAKE256-256s-robust	5
SPHINCS+-SHAKE256-256s-simple	5

Tabla A.3: Tabla de algoritmos de firma digital 2

B

figuras y tablas de SPHINCS$^+$

En este apéndice se encuentran las figuras y tablas que se referencian en 3.2.2

Algoritmo	Generación	Firma	Validación
SPHINCS+-Haraka-128f-robust	1100064	27654627	1896144
SPHINCS+-Haraka-128f-simple	849939	21088262	1275717
SPHINCS+-Haraka-128s-robust	64889588	515490693	679822
SPHINCS+-Haraka-128s-simple	60823634	475687971	465404

Tabla B.1: Media de ciclos de algoritmos de SPHINCS$^+$-128-Haraka

Algoritmo	Generación	Firma	Validación
SPHINCS+-Haraka-192f-robust	1541536	45316894	2799921
SPHINCS+-Haraka-192f-simple	1286772	36435795	1931588
SPHINCS+-Haraka-192s-robust	87661000	922318126	1039598
SPHINCS+-Haraka-192s-simple	90440012	895833826	705984

Tabla B.2: Media de ciclos de algoritmos de SPHINCS$^+$-192-Haraka

Algoritmo	Generación	Firma	Validación
SPHINCS+-Haraka-256f-robust	4409238	96513140	2809965
SPHINCS+-Haraka-256f-simple	3660904	78990321	1969127
SPHINCS+-Haraka-256s-robust	59872773	864075257	1455396
SPHINCS+-Haraka-256s-simple	60256765	818625925	1081963

Tabla B.3: Media de ciclos de algoritmos de SPHINCS$^+$-256-Haraka

Función Hash	Algoritmo	Generación	Firma	Validación
SHA256	SPHINCS+-128f-simple	2058739	63076436	15904689
	SPHINCS+-128f-robust	3651367	113409704	31823449
	SPHINCS+-128s-simple	123308738	941351606	6151810
	SPHINCS+-128s-robust	228508169	1733317479	10162916
	SPHINCS+-192f-simple	2714709	96242210	23397196
	SPHINCS+-192f-robust	5309377	179300352	46927169
	SPHINCS+-192s-simple	174899090	1744294000	8180178
	SPHINCS+-192s-robust	351813134	3329457381	15951486
	SPHINCS+-256f-simple	7274307	174428986	23625633
	SPHINCS+-256f-robust	25298535	549658979	52988721
	SPHINCS+-256s-simple	113977630	1443182655	11222481
	SPHINCS+-256s-robust	408097614	4657281254	25935221
SHAKE256	SPHINCS+-128f-simple	3964704	104199798	11186432
	SPHINCS+-128f-robust	7091912	185290362	21227126
	SPHINCS+-128s-simple	246460420	1884365092	3519205
	SPHINCS+-128s-robust	451160286	3425367428	7461730
	SPHINCS+-192f-simple	5779434	165818713	15028093
	SPHINCS+-192f-robust	10024688	284027331	28550030
	SPHINCS+-192s-simple	363758750	3390466257	5374637
	SPHINCS+-192s-robust	634697219	5737470380	10084555
	SPHINCS+6-256f-simple	14923122	319773009	15871188
	SPHINCS+6-256f-robust	26701783	556978355	30652988
	SPHINCS+-256s-simple	243208194	2977691962	8132098
	SPHINCS+-256s-robust	421398849	4930128270	15014806

Tabla B.4: Media de ciclos de algoritmos SPHINCS^{+}-SHA265/SHAKE256

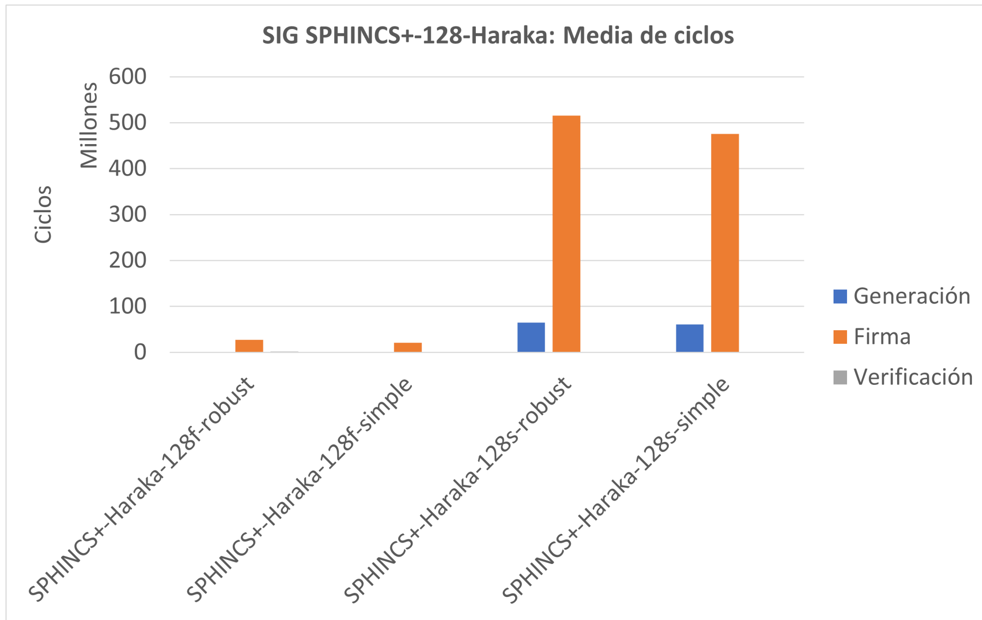

Figura B.1: Gráfico media de ciclos de algoritmos de SPHINCS^{+}-128-Haraka

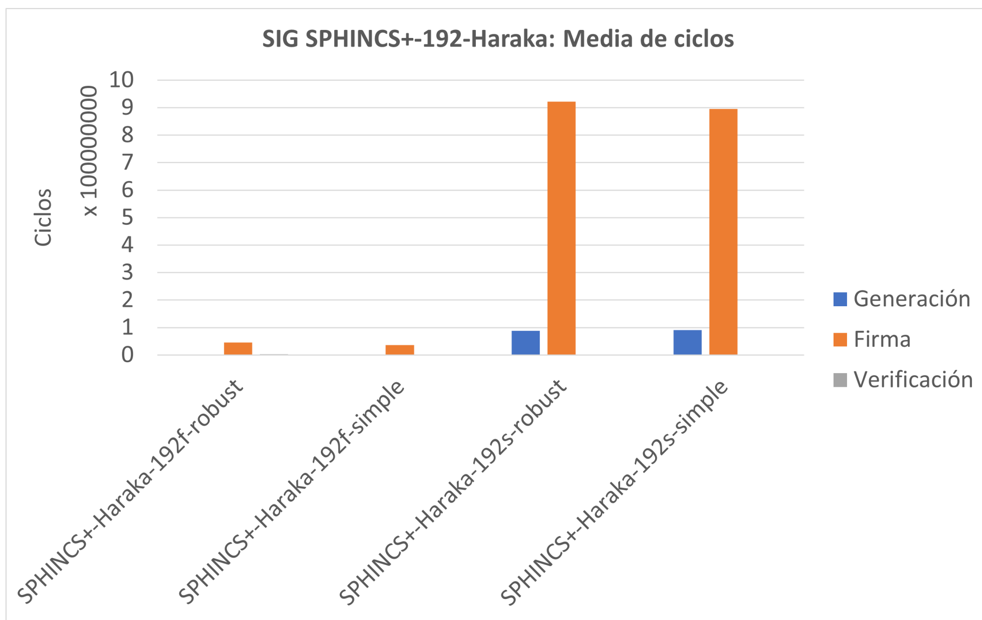

Figura B.2: Gráfico media de ciclos de algoritmos de SPHINCS^{+}-192-Haraka

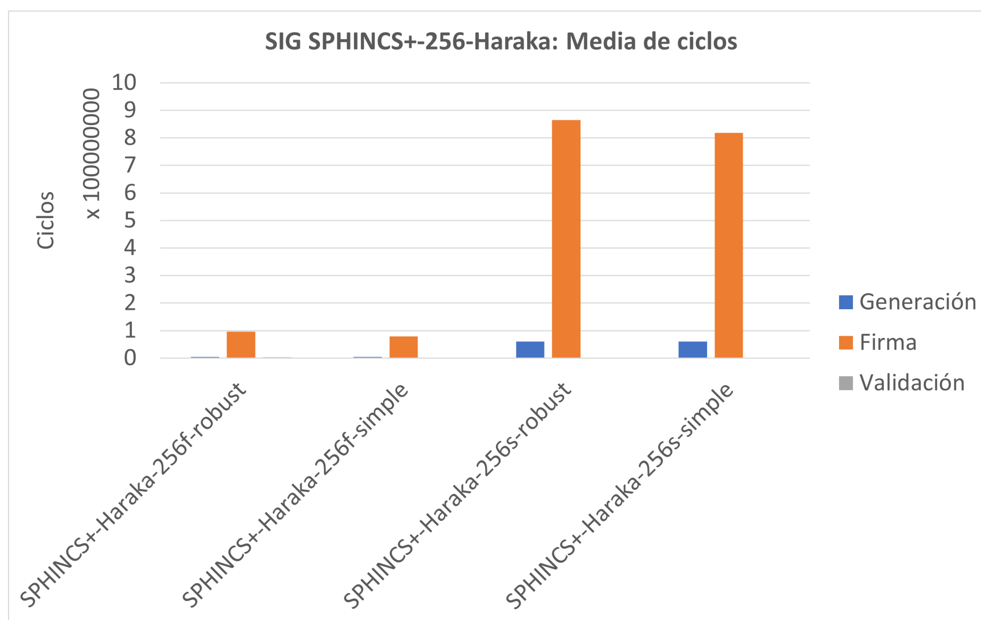

Figura B.3: Gráfico media de ciclos de algoritmos de SPHINCS+-256-Haraka

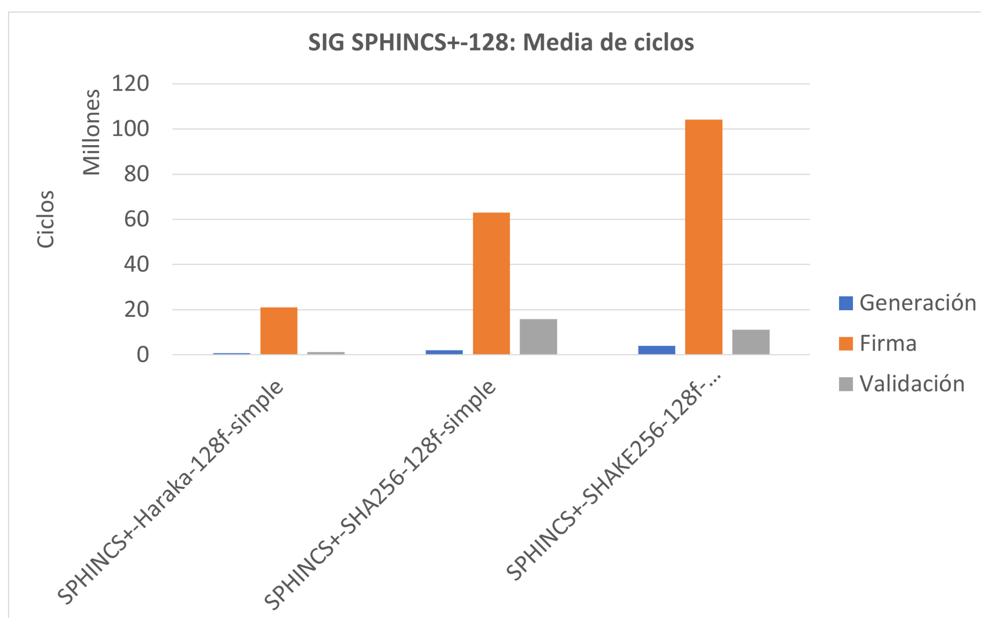

Figura B.4: Gráfico media de ciclos de algoritmos de SPHINCS+-128

C

Código

En las siguientes páginas se encuentra el código creado que ha dado lugar a los resultados: kem.c y kem.h

Código de KEM: kem.c

```
#include <oqs/oqs.h>
#include <stdio.h>
#include <string.h>
#include <x86intrin.h>

int main(){
    // Inicializacion de variables

    // Variable donde se guarda el objeto tipo KEM y el
       status de las funciones
    OQS_KEM *kem = NULL;
    OQS_STATUS rv;

    // Variable donde se guarda el n  de ciclos total y
       variables para calcular los ciclos
    uint64_t *ciclos = malloc(3*sizeof(uint64_t));
        const int generacion = 0;
        const int encapsulado = 1;
        const int desencapsulado = 2;
    uint64_t comienzo, fin;
    if(ciclos == NULL){
```

```
20              fprintf(stderr, "Error reservando memoria.
                    Abortando");
21          }
22
23
24          /*Lugar donde se guardan las claves y los secretos del
                usuario. En el siguiente orden:
25              public_key      —> 0
26              secret_key      —> 1
27              ciphertext      —> 2
28              shared_secret_e —> 3
29              shared_secret_d —> 4
30          */
31          uint8_t **user_vault = malloc(5 * sizeof(uint8_t*));
32              const int public_key = 0;
33              const int secret_key = 1;
34              const int ciphertext = 2;
35              const int shared_secret_e = 3;
36              const int shared_secret_d = 4;
37
38          // Punteros de ficheros y variables utilizadas
39          FILE *lista, *resultados, *valores = NULL;
40          char *algoritmo = malloc(1024 * sizeof(char));
41          if(algoritmo == NULL){
42              fprintf(stderr, "Error reservando memoria.
                    Abortando");
43              free(user_vault);
44              free(ciclos);
45              return OQS_ERROR;
46          }
47
48          // Apertura de ficheros de la lista
49          lista = fopen("kem_list", "r");
50          if(lista == NULL){
51              fprintf(stderr, "Error abriendo lista. Abortando")
                    ;
52              free(user_vault);
53              free(algoritmo);
54              free(ciclos);
55              return OQS_ERROR;
56          }
57
58          // Apertura de fichero de resultados
59          resultados = fopen("kem.csv", "w+");
60          if(resultados == NULL){
61              fprintf(stderr, "Error abriendo csv. Abortando");
62              free(algoritmo);
```

```
63                  free(ciclos);
64                  fclose(lista);
65                  return OQS_ERROR;
66              }
67              fprintf(resultados, "Algoritmo,PK Size,SK Size,
                    Ciphertext Size,Media ciclos generacion,Media
                    ciclos encapsulado,Media ciclos desencapsulado\n");
68
69              // Apertura de fichero de valores
70              valores = fopen("valores_kem.csv", "w+");
71              if(resultados == NULL){
72                  fprintf(stderr, "Error abriendo csv. Abortando");
73                  free(algoritmo);
74                  free(ciclos);
75                  fclose(lista);
76                  fclose(resultados);
77                  return OQS_ERROR;
78              }
79
80              int iteraciones = 1024;
81              uint64_t media = 0;
82              uint64_t valor = 0;
83              while(!feof(lista)){
84                  // Lectura del algoritmo de la lista
85                  fgets(algoritmo, 1024, lista);
86                  algoritmo[strcspn(algoritmo, "\n")] = 0;
87
88                  printf("%s\n", algoritmo);
89                  // Se inicializa el algoritmo
90                  kem = OQS_KEM_new(algoritmo);
91                  if (kem == NULL){
92                      fprintf(stderr, "Algoritmo inicializado
                            incorrectamente. Abortando\n");
93                      free(algoritmo);
94                      free(user_vault);
95                      free(ciclos);
96                      fclose(lista);
97                      fclose(resultados);
98                      fclose(valores);
99                      return OQS_ERROR;
100                 }
101
102                 // Se reserva memoria para las variables del vault
103                 user_vault[public_key] = malloc(kem->
                        length_public_key);
104                 user_vault[secret_key] = malloc(kem->
                        length_secret_key);
```

```
105                 user_vault[ciphertext] = malloc(kem->
                       length_ciphertext);
106                 user_vault[shared_secret_e] = malloc(kem->
                       length_shared_secret);
107                 user_vault[shared_secret_d] = malloc(kem->
                       length_shared_secret);
108
109                 for(int i = 0; i < 5; i++)
110                 if (user_vault[i] == NULL){
111                     fprintf(stderr, "Error reservando memoria en
                           posicion %d del user_vault. Abortando\n", i
                           );
112                     OQS_KEM_free(kem);
113                     for (int j = 0; j<i; i++)
114                         free(user_vault[j]);
115                     free(user_vault);
116                     free(algoritmo);
117                     free(ciclos);
118                     fclose(lista);
119                     fclose(resultados);
120                     fclose(valores);
121                     return OQS_ERROR;
122                 }
123
124                 fprintf(valores, "%s\n", algoritmo);
125
126
127                 // Se genera el par de claves
128                 printf("\tGenerando claves\n");
129                 fprintf(valores, " ,Generacion,");
130                 for(int i=0; i<iteraciones; i++){
131                     comienzo = __rdtsc();
132                     rv = OQS_KEM_keypair(kem, user_vault[
                           public_key], user_vault[secret_key]);
133                     if(rv != OQS_SUCCESS){
134                         fprintf(stderr, "No se ha podido generar
                               el par de claves. Abortando\n");
135                         OQS_KEM_free(kem);
136                         for (int j = 0; j<i; i++)
137                             free(user_vault[j]);
138                         free(user_vault);
139                         free(algoritmo);
140                         free(ciclos);
141                         fclose(lista);
142                         fclose(resultados);
143                         fclose(valores);
144                         return OQS_ERROR;
```

```
145                 }
146                 fin = __rdtsc();
147                 valor = fin - comienzo;
148                 fprintf(valores, "%'lu,", valor);
149                 media += valor;
150             }
151             fprintf(valores, "\n");
152             ciclos[generacion] = media/iteraciones;
153             media = 0;
154 
155             printf("\tCifrando mensaje\n");
156             // Se genera el ciphertext
157             fprintf(valores, " ,Encapsulado,");
158             for(int i=0; i<iteraciones; i++){
159                 comienzo = __rdtsc();
160                 rv = OQS_KEM_encaps(kem, user_vault[ciphertext
                        ], user_vault[shared_secret_e], user_vault[
                        public_key]);
161                 if(rv != OQS_SUCCESS){
162                     fprintf(stderr, "No se ha podido cifrar el
                            mensaje. Abortando\n");
163                     OQS_KEM_free(kem);
164                     for (int j = 0; j<i; i++)
165                         free(user_vault[j]);
166                     free(user_vault);
167                     free(algoritmo);
168                     free(ciclos);
169                     fclose(lista);
170                     fclose(resultados);
171                     fclose(valores);
172                     return OQS_ERROR;
173                 }
174                 fin = __rdtsc();
175                 valor = fin - comienzo;
176                 fprintf(valores, "%'lu,", valor);
177                 media += valor;
178             }
179             fprintf(valores, "\n");
180             ciclos[encapsulado] = media/iteraciones;
181             media = 0;
182 
183             printf("\tDescifrando el mensaje\n\n");
184             // Se descifra el ciphertext
185             fprintf(valores, " ,Desencapsulado,");
186             for(int i=0; i<iteraciones; i++){
187                 comienzo = __rdtsc();
```

```
188                 rv = OQS_KEM_decaps(kem, user_vault[
                        shared_secret_d], user_vault[ciphertext],
                        user_vault[secret_key]);
189                 if(rv != OQS_SUCCESS){
190                     fprintf(stderr, "No se ha podido cifrar el
                            mensaje. Abortando\n");
191                     OQS_KEM_free(kem);
192                     for (int j = 0; j<i; i++)
193                         free(user_vault[j]);
194                     free(user_vault);
195                     free(algoritmo);
196                     free(ciclos);
197                     fclose(lista);
198                     fclose(resultados);
199                     fclose(valores);
200                     return OQS_ERROR;
201                 }
202                 fin = __rdtsc();
203                 valor = fin - comienzo;
204                 fprintf(valores, "%'lu,", valor);
205                 media += valor;
206             }
207             fprintf(valores, "\n");
208             ciclos[desencapsulado] = media/iteraciones;
209             media = 0;
210 
211             // Se guardan los resultados en un archivo
212             fprintf(resultados, "%s,%lu,%lu,%lu,%'lu,%'lu,%'lu
                    \n", algoritmo, kem->length_public_key, kem->
                    length_secret_key, kem ->length_ciphertext,
                    ciclos[generacion], ciclos[encapsulado], ciclos
                    [desencapsulado]);
213 
214             for (int i = 0; i<5; i++)
215                 free(user_vault[i]);
216 
217         }
218 
219         // Liberacion de recursos
220         fclose(lista);
221         fclose(resultados);
222         fclose(valores);
223         OQS_KEM_free(kem);
224         free(algoritmo);
225         free(user_vault);
226         free(ciclos);
227 
```

```
228         return 0;
229     }
```

Cdigo de firma: sig.c

```
1     #include <oqs/oqs.h>
2     #include <stdio.h>
3     #include <string.h>
4     #include <x86intrin.h>
5
6     #define MESSAGE_LEN 50
7
8     int main(){
9         // Inicializacion de variables
10
11        // Variable donde se guarda el objeto de tipo SIG y el
              status de las funciones
12        OQS_SIG *sig = NULL;
13        OQS_STATUS rv;
14
15        // Variable donde se guarda el n   de ciclos total y
             variables para calcular los ciclos
16        uint64_t *ciclos = malloc(3*sizeof(uint64_t));
17            const int generacion = 0;
18            const int encapsulado = 1;
19            const int desencapsulado = 2;
20        uint64_t comienzo, fin;
21        if(ciclos == NULL){
22            fprintf(stderr, "Error reservando memoria.
                 Abortando");
23        }
24
25        /*Lugar donde se guardan las claves y la firma del
             usuario. En el siguiente orden:
26            public_key      —> 0
27            secret_key      —> 1
28            message     —> 2
29            signature —> 3
30        */
31       uint8_t **user_vault = malloc(5 * sizeof(uint8_t*));
32            const int public_key = 0;
33            const int secret_key = 1;
34            const int message = 2;
35            const int signature = 3;
36        size_t message_len = MESSAGE_LEN;
```

```
37          size_t signature_len;
38
39          FILE *lista, *resultados, *valores = NULL;
40          char *algoritmo = malloc(1024 * sizeof(char));
41          if(algoritmo == NULL){
42              fprintf(stderr, "Error reservando memoria.
                    Abortando");
43              free(user_vault);
44              free(ciclos);
45              return OQS_ERROR;
46          }
47
48          // Apertura de ficheros de la lista
49          lista = fopen("sig_list", "r");
50          if(lista == NULL){
51              fprintf(stderr, "Error abriendo lista. Abortando")
                    ;
52              free(user_vault);
53              free(algoritmo);
54              free(ciclos);
55              return OQS_ERROR;
56          }
57
58          // Apertura de fichero de resultados
59          resultados = fopen("sig.csv", "w+");
60          if(resultados == NULL){
61              fprintf(stderr, "Error abriendo csv. Abortando");
62              free(algoritmo);
63              free(ciclos);
64              fclose(lista);
65              return OQS_ERROR;
66          }
67          fprintf(resultados,"Algoritmo,PK Size,SK Size,Sig Size
                ,Media ciclos generacion,Media ciclos firma,Media
                ciclos verificacion\n");
68
69          // Apertura de fichero de valores
70          valores = fopen("valores_sig.csv", "w+");
71          if(resultados == NULL){
72              fprintf(stderr, "Error abriendo csv. Abortando");
73              free(algoritmo);
74              free(ciclos);
75              fclose(lista);
76              fclose(resultados);
77              return OQS_ERROR;
78          }
79
```

```
80          int iteraciones = 100;
81          uint64_t media = 0;
82          uint64_t valor = 0;
83          while(!feof(lista)){
84              // Lectura del algoritmo en la lista
85              fgets(algoritmo, 1024, lista);
86              algoritmo[strcspn(algoritmo, "\n")] = 0;
87              printf("%s\n", algoritmo);
88
89              // Se inicializa el algoritmo
90              sig = OQS_SIG_new(algoritmo);
91              if(sig == NULL){
92                  fprintf(stderr, "Algoritmo inicializado
                        incorrectamente. Abortando\n");
93                  free(algoritmo);
94                  free(user_vault);
95                  free(ciclos);
96                  fclose(lista);
97                  fclose(resultados);
98                  fclose(valores);
99                  return OQS_ERROR;
100             }
101
102             // Se reserva memoria para las variables del vault
103             user_vault[public_key] = malloc(sig->
                    length_public_key);
104             user_vault[secret_key] = malloc(sig->
                    length_secret_key);
105             user_vault[message] = malloc(message_len);
106             user_vault[signature] = malloc(sig->
                    length_signature);
107
108             for(int i = 0; i < 4; i++)
109             if (user_vault[i] == NULL){
110                 fprintf(stderr, "Error reservando memoria en
                        posicion %d del user_vault. Abortando\n", i
                        );
111                 OQS_SIG_free(sig);
112                 for (int j = 0; j<i; i++)
113                     free(user_vault[j]);
114                 free(user_vault);
115                 free(algoritmo);
116                 free(ciclos);
117                 fclose(lista);
118                 fclose(resultados);
119                 fclose(valores);
120                 return OQS_ERROR;
```

```
121             }
122
123             // Se genera un mensaje aleatorio
124             OQS_randombytes(user_vault[message], message_len);
125
126             fprintf(valores, "%s\n", algoritmo);
127             // Se genera el par de claves
128             printf("\tGenerando claves\n");
129             fprintf(valores, " ,Generacion,");
130             for(int i=0; i<iteraciones; i++){
131                 comienzo = __rdtsc();
132                 rv =  OQS_SIG_keypair(sig, user_vault[
                        public_key], user_vault[secret_key]);
133                 if(rv != OQS_SUCCESS){
134                     fprintf(stderr, "No se ha podido generar
                            el par de claves. Abortando\n");
135                     OQS_SIG_free(sig);
136                     for (int j = 0; j<i; i++)
137                         free(user_vault[j]);
138                     free(user_vault);
139                     free(algoritmo);
140                     free(ciclos);
141                     fclose(lista);
142                     fclose(resultados);
143                     fclose(valores);
144                     return OQS_ERROR;
145                 }
146                 fin = __rdtsc();
147                 valor = fin - comienzo;
148                 fprintf(valores, "%'lu,", valor);
149                 media += valor;
150             }
151             fprintf(valores, "\n");
152             ciclos[generacion] = media/iteraciones;
153             media = 0;
154
155             printf("\tFirmando el mensaje\n");
156             // Se firma el mensaje
157             fprintf(valores, " ,Firma,");
158             for(int i=0; i<iteraciones; i++){
159                 comienzo = __rdtsc();
160                 rv = OQS_SIG_sign(sig, user_vault[signature],
                       &signature_len, user_vault[message],
                       message_len, user_vault[secret_key]);
161                 if(rv != OQS_SUCCESS){
162                     fprintf(stderr, "No se ha podido firmar el
                            mensaje. Abortando\n");
```

```
163                         OQS_SIG_free(sig);
164                         for (int j = 0; j<i; i++)
165                             free(user_vault[j]);
166                         free(user_vault);
167                         free(algoritmo);
168                         free(ciclos);
169                         fclose(lista);
170                         fclose(resultados);
171                         fclose(valores);
172                         return OQS_ERROR;
173                     }
174                     fin = __rdtsc();
175                     valor = fin - comienzo;
176                     fprintf(valores, "%'lu,", valor);
177                     media += valor;
178                 }
179                 fprintf(valores, "\n");
180                 ciclos[encapsulado] = media/iteraciones;
181                 media = 0;
182 
183                 printf("\tVerificando la firma\n\n");
184                 // Se verifica la firma
185                 fprintf(valores, "  ,Verificado,");
186                 for(int i=0; i<iteraciones; i++){
187                     comienzo = __rdtsc();
188                     rv = OQS_SIG_verify(sig, user_vault[message],
                           message_len, user_vault[signature],
                           signature_len, user_vault[public_key]);
189                     if(rv != OQS_SUCCESS){
190                         fprintf(stderr, "No se ha podido verificar
                               la firma. Abortando\n");
191                         OQS_SIG_free(sig);
192                         for (int j = 0; j<i; i++)
193                             free(user_vault[j]);
194                         free(user_vault);
195                         free(algoritmo);
196                         free(ciclos);
197                         fclose(lista);
198                         fclose(resultados);
199                         fclose(valores);
200                         return OQS_ERROR;
201                     }
202                     fin = __rdtsc();
203                     valor = fin - comienzo;
204                     fprintf(valores, "%'lu,", valor);
205                     media += valor;
206                 }
```

```
207                 fprintf(valores, "\n");
208                 ciclos[desencapsulado] = media/iteraciones;
209                 media = 0;
210 
211                 // Se guardan los resultados en un archivo
212                 fprintf(resultados, "%s,%lu,%lu,%lu,%'lu,%'lu,%'lu
                        \n", algoritmo, sig->length_public_key, sig->
                        length_secret_key, sig->length_signature,ciclos
                        [generacion], ciclos[encapsulado], ciclos[
                        desencapsulado]);
213 
214                 for (int i = 0; i<4; i++)
215                     free(user_vault[i]);
216             }
217 
218             // Liberacion de recursos
219             fclose(lista);
220             fclose(resultados);
221             fclose(valores);
222             OQS_SIG_free(sig);
223             free(algoritmo);
224             free(user_vault);
225             free(ciclos);
226 
227             return 0;
228         }
```

D
Licencia

©2023 Daniel Alfonsel García, Ana Isabel Gómez Pérez

Algunos derechos reservados

Este documento se distribuye bajo la licencia "Atribución- CompartirIgual 4.0 Internacional" de Creative Commons, disponible en: https://creativecommons.org/licenses/by-sa/4.0/deed.es