

A todos los que me han acompañado en este largo viaje. . .

“Sé paciente contigo mismo.

Nada en la naturaleza florece todo el año.”

Resumen

En este TFM se ha llevado a cabo la adaptación y ejecución de varios algoritmos criptográficos postcuánticos, entre los cuales se encuentran Dilithium, Kyber y Sphincs en distintos dispositivos relativos a IoT, como son ESP32, RP2040 y STM32.

Posteriormente, se ha llevado a cabo la medición de rendimiento de estas implementaciones y, finalmente, dichas mediciones han sido comparadas.

Palabras clave: Algoritmo, Postcuántico, IoT, ESP32, Dilithium.

Abstract

In this TFM, multiple cryptographic post-quantum algorithms, such as Dilithium, Kyber and Sphincs, have been adapted and executed in various platforms relative to IoT, like ESP32, RP2040 and STM32.

Afterwards, these implementations' performance have been measured and, finally, these measurements have been compared.

Keywords: Algorithm, Post-quantum, IoT, ESP32, Dilithium.

Índice general

Resumen	iii
Abstract	v
Índice general	vii
Índice de figuras	xi
Índice de tablas	xiii
Índice de código fuente	xv
Índice de algoritmos	xvii
Lista de acrónimos	xix
1 Introducción	1
1.1 Motivación	1
1.2 Objetivos del proyecto	1
1.3 Estructura de la memoria	1
2 Estado del arte	3
2.1 Criptografía	3
2.1.1 Características	3
2.1.2 Modelos actuales	4
2.1.2.1 RSA	4
2.1.2.2 DSA	4
2.1.2.3 Curva elíptica	4
2.1.3 Computación cuántica	5
2.1.4 Criptografía postcuántica	5
2.1.4.1 Dilithium	7
2.1.4.2 Kyber	8
2.1.4.3 Sphincs+	8
2.1.4.4 McEliece	9
2.1.4.5 HQC	10
2.1.4.6 Falcon	10
2.2 IoT	10
2.2.1 ESP32	11
2.2.2 RP2040	11
2.2.3 STM32	12
3 Desarrollo	15
3.1 Selección de dispositivos	15
3.2 ESP32	15
3.2.1 ESP-IDF	16
3.2.1.1 IDF FreeRTOS	16
3.2.1.2 Instalación	16
3.2.2 ESP-Prog	16

3.2.3	Dilithium	17
3.2.3.1	Ficheros CMake	18
3.2.3.2	Generación de números aleatorios	19
3.2.3.3	Fichero de comprobación	20
3.2.3.4	Uso de memoria dinámica	21
3.2.3.5	Comprobación del algoritmo	23
3.2.4	Kyber	25
3.2.4.1	Ficheros CMake	26
3.2.4.2	Generación de números aleatorios	26
3.2.4.3	Fichero de comprobación	26
3.2.4.4	Uso de memoria dinámica	28
3.2.4.5	Comprobación del algoritmo	29
3.3	RP2040	30
3.3.1	Pico SDK	30
3.3.2	HQC-128	30
3.3.2.1	Fichero CMake	31
3.3.2.2	Generación de números aleatorios	33
3.3.2.3	Fichero de comprobación	33
3.3.2.4	Comprobación del algoritmo	35
3.3.3	McEliece348864	35
3.3.3.1	Generación de números aleatorios	37
3.3.3.2	Fichero CMake	37
3.3.3.3	Fichero de comprobación	38
3.3.3.4	Comprobación del algoritmo	38
3.3.4	Sphincs	38
3.3.4.1	Fichero CMake	40
3.3.4.2	Generación de números aleatorios	41
3.3.4.3	Fichero de comprobación	41
3.3.4.4	Comprobación del algoritmo	42
3.4	STM32	42
3.4.1	Keil uVision5	43
3.4.2	McEliece348864	43
3.4.2.1	Generación de números aleatorios	43
3.4.2.2	Archivo startup_stm32l4r5xx.s	44
3.4.2.3	Función de encapsulado y desencapsulado	45
3.4.3	PQM4	45
4	Evaluación y resultados	47
4.1	Pruebas unitarias de comprobación	47
4.2	Medidas de ejecución	47
4.2.1	Consumo energético	47
4.2.2	Tiempo de ejecución	47
4.2.2.1	Temporización en dispositivo ESP32	47
4.2.2.2	Temporización en dispositivo RP2040	48
4.2.2.3	Temporización en dispositivo STM32	48
4.2.3	Pruebas en dispositivo ESP32	48
4.2.4	Pruebas en dispositivo RP2040	49
4.2.5	Pruebas en dispositivo STM32	49
5	Conclusiones y líneas futuras	51
5.1	Conclusiones	51
5.2	Líneas futuras	51
	Bibliografía	53
	Apéndice A Temporización y presupuesto	57
A.1	PLanificación temporal	57

A.2 Recursos Hardware	58
A.3 Recursos Humanos	58
A.4 Presupuesto de ejecución material	58

Índice de figuras

2.1	Bit clásico y qubit [1].	5
2.2	Progreso en el número de qubits de un computador cuántico [2].	6
2.3	ESP32-WROOM-32E en Devkitc v4 con <i>pinout</i> [14].	11
2.4	Raspberry Pi Pico [16].	11
2.5	Diagrama del dispositivo NUCLEO-L4R5ZI [18].	12
3.1	Diagrama del dispositivo ESP-Prog [25].	17
3.2	Árbol de ficheros del proyecto Dilithium.	18
3.3	Comprobación de ejecución de algoritmo Dilithium.	24
3.4	Árbol de ficheros del proyecto Kyber.	25
3.5	Comprobación de ejecución de algoritmo Kyber.	29
3.6	Árbol de ficheros del proyecto HQC-128.	31
3.7	Comprobación de ejecución de algoritmo HQC-128.	35
3.8	Árbol de ficheros del proyecto McEliece348864.	36
3.9	Comprobación de ejecución de algoritmo McEliece348864.	39
3.10	Árbol de ficheros del proyecto Sphincs.	39
3.11	Comprobación de ejecución de algoritmo Sphincs.	42
3.12	Errores de manejo de memoria <i>stack</i> al ejecutar McEliece348864.	44
4.1	Tiempo de ejecución medio para el algoritmo Dilithium.	49
4.2	Tiempo de ejecución medio para el algoritmo Kyber.	50
4.3	Tiempo de ejecución medio para el algoritmo HQC.	50
A.1	Diagrama de Gantt	57

Índice de tablas

2.1	Diferencias entre versiones de Dilithium [6].	8
2.2	Diferencias entre versiones de Kyber [7].	8
2.3	Tamaños de las diferentes versiones de Sphincs+ [9].	9
2.4	Diferencias entre versiones de McEliece [10].	10
2.5	Diferencias entre versiones de HQC [11].	10
2.6	Diferencias entre versiones de Falcon [12].	10
A.1	Recursos hardware usados	58
A.2	Recursos humanos	58
A.3	Presupuesto de ejecución material	58

Índice de listados de código fuente

3.1	Instalación de ESP-IDF [23].	16
3.2	Archivo Dilithium/CMakeLists.txt.	18
3.3	Archivo Dilithium/main/CMakeLists.txt.	19
3.4	Archivo Dilithium/main/src/randombytes.c.	19
3.5	Generación de números aleatorios para ESP32.	20
3.6	Archivo Dilithium/main/main.c.	20
3.7	Archivo Dilithium/main/include/main.h.	21
3.8	Modificación de la función crypto_sign_keypair en el archivo Dilithium/main/src/sign.c.	22
3.9	Modificación de la función crypto_sign_signature en el archivo Dilithium/main/src/sign.c.	22
3.10	Modificación de la función crypto_sign_verify en el archivo Dilithium/main/src/sign.c.	23
3.11	Archivo Kyber/CMakeLists.txt.	26
3.12	Archivo Kyber/main/CMakeLists.txt.	26
3.13	Archivo Kyber/main/main.c.	27
3.14	Archivo Kyber/main/include/main.h.	28
3.15	Modificación de la función indcpa_keypair en el archivo Kyber/main/src/indcpa.c.	28
3.16	Modificación de la función indcpa_enc en el archivo Kyber/main/src/indcpa.c.	28
3.17	Modificación de la función indcpa_dec en el archivo Kyber/main/src/indcpa.c.	29
3.18	Ejemplo de CMakeLists.txt [34].	30
3.19	Archivo HQC-128/CMakeLists.txt.	31
3.20	Archivo HQC-128/src/randombytes.c.	33
3.21	Archivo HQC-128/include/randombytes.h.	33
3.22	Archivo HQC-128/main.c.	33
3.23	Archivo HQC-128/main.h.	34
3.24	Archivo McEliece348864/src/rndombytes.c.	37
3.25	Archivo McEliece348864/CMakeLists.txt.	37
3.26	Archivo Sphincs/CMakeLists.txt.	40
3.27	Archivo Sphincs/src/randombytes.c.	41
3.28	Archivo Sphincs/main.c.	41
3.29	Función de generación de números aleatorios para McEliece348864.	43
3.30	Variables en la función pk_gen.	44
3.31	Comando para la obtención de medidas utilizando PQM4.	45
4.1	Medición temporal en el dispositivo ESP32.	48
4.2	Medición temporal en el dispositivo RP2040.	48

Índice de algoritmos

2.1	Generación de claves pública y privadas en Dilithium [6].	7
2.2	Firma en Dilithium [6].	7
2.3	Verificación de firma en Dilithium [6].	7
2.4	Generación de claves en Kyber [7].	8

Lista de acrónimos

CAN	Controller Area Network.
DIY	Do It Yourself.
DMA	Direct Memory Access.
DSS	Digital Signature Standard.
FreeRTOS	Free Real-Time Operating System.
GPIO	General Purpose Input Output.
I2C	Inter-Integrated Circuit.
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
IoT	Internet of Things.
KEM	Key-Encapsulation Mecanism.
LWE	Learning With Errors.
NIST	National Institute of Standards and Technology.
OW-CPA	One Way Chosen Plaintext Attack.
SAI	Serial Audio Interface.
SDIO	Secure Digital Input Output.
SIS	Short Integer Solution.
SMP	Symmetric MultiProcessing.
SoC	System On a Chip.
SPI	Serial Peripheral Interface.
UART	Universal Asynchronous Receiver/Transmitter.
USART	Universal Synchronous and Asynchronous Receiver-Transmitter.
USB	Universal Serial Bus.

Capítulo 1

Introducción

En este capítulo se explica la motivación y necesidad que han llevado a la ejecución de este trabajo, así como los distintos objetivos que se busca conseguir a lo largo de su realización. Adicionalmente, se especifica la estructura por la que se rige este documento.

1.1 Motivación

1.2 Objetivos del proyecto

El objetivo final de este trabajo consiste en la implementación de algoritmos criptográficos post-cuánticos en distintos dispositivos Internet of Things (IoT). Para poder alcanzar este objetivo, se han planteado una serie de pautas cuyo seguimiento conduce al objetivo final:

- **Selección de algoritmos a implementar:** En primer lugar, se deberá elegir que algoritmos se buscará implementar.
- **Selección de plataformas IoT a emplear:** A continuación, se llevará a cabo una selección de los dispositivos IoT en los que se implementarán los algoritmos previamente mencionados.
- **Análisis de compatibilidad:** El siguiente objetivo consistirá en llevar a cabo un análisis de compatibilidad existente entre los algoritmos y los dispositivos seleccionados.
- **Validación, diseño de pruebas y análisis de resultados:** Finalmente, se comprobará el funcionamiento de los algoritmos mediante pruebas y el posterior análisis de los resultados.

1.3 Estructura de la memoria

Después de haber explicado la motivación el proyecto así como los objetivos que se buscan mediante el mismo, se indica la estructura que seguirá este documento:

1. **Introducción:** En este capítulo se realiza una contextualización del proyecto. Además, se indican los objetivos buscados en dicho trabajo y la estructura que sigue este documento explicativo.
2. **Estado del arte:** A lo largo de este capítulo se detallan una serie de conocimientos previos que el lector debe conocer para poder comprender correctamente el funcionamiento y desarrollo del proyecto.
3. **Desarrollo:** En este capítulo se presentan las vías de trabajo llevadas a cabo para conseguir los objetivos mencionados en el capítulo introductorio.

4. **Resultados:** A lo largo de este capítulo se muestran las diferentes pruebas realizadas a los algoritmos en los diferentes dispositivos utilizados.
5. **Conclusiones y trabajo futuro:** Se finaliza esta memoria con un capítulo que comente las conclusiones extraíbles de los resultados explicados en el apartado anterior. También, se indicarán posibles vías de desarrollo para mejorar la implementación final y añadir características a la misma.
6. **Bibliografía:** Se incluyen todos los documentos consultados para la realización de este trabajo, ya sean páginas web, libros o artículos. Dichos documentos estarán citados siguiendo el estilo indicado por el Institute of Electrical and Electronics Engineers (IEEE).
7. **Anexo A:** Finalmente, se añade un anexo en el cual se indica la planificación temporal seguida durante la ejecución de este proyecto y el coste de la realización del mismo.

Capítulo 2

Estado del arte

En este capítulo, se lleva a cabo la explicación de distintos aspectos teóricos esenciales para el entendimiento del desarrollo del proyecto. Para ello, este capítulo se dividirá en dos apartados principales: criptografía e IoT.

2.1 Criptografía

En este primer apartado se van a mostrar distintos aspectos importantes en lo referente a la criptografía, como pueden ser las características principales así como los esquemas utilizados actualmente, el paradigma de la computación cuántica y la criptografía postcuántica.

2.1.1 Características

La criptografía consiste en el cifrado de datos con el objetivo de obtener confidencialidad e integridad. Para ello, a lo largo de la historia se han utilizado distintos esquemas, como pueden ser el cifrado César, el cual se basa en la sustitución de los caracteres utilizados por los caracteres a una determinada cantidad de posiciones dentro del abecedario. Otro esquema utilizado es el sistema Vigenère, el cual se basa en desplazamiento, en el cual cada caracter se desplaza una cantidad de posiciones independiente, de forma que se genera un vector k (número de posiciones desplazadas de cada caracter) que representa la clave para el sistema.

En los sistemas criptográficos se pueden buscar 4 posibles tipos de seguridad:

- **Seguridad incondicional:** El sistema es seguro, sin importar los recursos de los que disponga el atacante.
- **Seguridad computacional:** El sistema es seguro frente a un atacante con recursos computacionales limitados
- **Seguridad heurística:** No se ha demostrado seguro pero, por el momento, no se ha encontrado forma de romper el sistema.
- **Seguridad condicional:** Cualquier sistema que no se incluye en las categorías anteriores.

Los sistemas actuales se basan en la seguridad computacional, por lo que, ante el avance en la capacidad de computación, estos se deben actualizar para mantener la seguridad deseada.

Dentro de estos sistemas, se pueden diferenciar dos tipos: los sistemas de clave simétrica y los sistemas de clave asimétrica. Estos primeros utilizan una única clave en ambos extremos de la comunicación, de forma que ambos usuarios utilizan la misma clave para cifrar y descifrar. Los sistemas de clave asimétrica hacen uso de dos claves distintas: una pública y otra privada. La clave pública es conocida por todos los usuarios y se puede utilizar para cifrar un mensaje cuyo destinatario sea

el propietario de esas claves. La clave privada solamente es conocida por el usuario que la crea y se utiliza para descifrar un mensaje cifrado con la clave pública. Además, se puede utilizar la clave privada para firmar un mensaje y, dicha firma, se puede verificar mediante el uso de la clave pública.

2.1.2 Modelos actuales

En cuanto a los sistemas de clave asimétrica actuales, es necesario mencionar RSA, DSA y sistemas basados en curva elíptica.

2.1.2.1 RSA

El mecanismo RSA se desarrolló en 1979 y su seguridad se basa en la factorización de números enteros.

En cuanto al funcionamiento del mecanismo, en primer lugar, se generan las claves, para lo cual se han de elegir dos números primos distintos (p y q) de forma aleatorio en de longitud en bits similar. Con el producto de estos dos números se obtiene el módulo para ambas claves n . A continuación, se escoge el exponente de la clave pública e y un respectivo exponente de la clave privada d , de forma que cumplan $e \times d \equiv 1$. De esta forma, es posible cifrar un texto m mediante $m^e \pmod{n}$. Para descifrar dicho texto cifrado, se debe ejecutar $c^d \pmod{n}$.

Desde el año 2002, se recomendó que las claves fueran de, al menos, 1024 bits aunque, en 2015, el National Institute of Standards and Technology (NIST) consideró que la longitud mínima de las claves debe ser de, por lo menos, 2048 bits. Por el momento, existe un gran debate acerca de si la computación cuántica será capaz de resolver el problema en el que se basa este algoritmo en poco tiempo de ejecución, inutilizando así este algoritmo incluso con tamaños de clave mayores.

2.1.2.2 DSA

Por otro lado, tenemos el algoritmo de firma DSA, el cuál se desarrolló en el año 1991. Este algoritmo fue propuesto por el NIST en su Digital Signature Standard (DSS).

Este mecanismo dispone de 3 funciones distintas: generación de claves, firma y verificación. Para la generación de claves, se debe elegir un número primo p cuya longitud en bits sea divisible por 64. A continuación, se debe escoger un número primo q de 160 bits tal que $p - 1 = qz$ donde z es un número natural. También, se debe elegir h de forma que $1 < h < p - 1$ y $h^z \pmod{p} > 1$ y x de forma que $1 < x < q - 1$. Por último, se computa $y = g^x \pmod{p}$. Los datos públicos son p , q , g e y . La clave privada es x .

Para la firma, se debe escoger un número aleatorio k de forma que $1 < k < q$. A continuación, se calcula $r = (g^k \pmod{p}) \pmod{q}$ y $s = k^{-1}(H(m) + r \times x) \pmod{q}$, donde $H(m)$ es una función resumen sobre el mensaje m . La firma del mensaje es el par (r,s) .

En cuanto a la verificación, se debe calcular $w = s^{-1} \pmod{q}$, $u_1 = (H(m) \times w) \pmod{q}$, $u_2 = (r \times w) \pmod{q}$ y $v = [g^{u_1} \times y^{u_2} \pmod{p}] \pmod{q}$. Si se cumple $v = r$, la firma es correcta.

Respecto a este algoritmo, el NIST ha recomendado el uso de claves de, como poco, 2048 bits y no recomienda el uso de la función resumen SHA-1.

2.1.2.3 Curva elíptica

Por último, se hace uso de sistemas que emplean curva elíptica ya que estos proporcionan una seguridad equivalente en una longitud de clave menor.

Una curva elíptica se representa mediante la ecuación $y^2 = x^3 + ax + b$. Para su uso en criptografía, se utiliza un punto base G publicado con una curva específica. A continuación, se escoge un número entero aleatorio k , el cual se utiliza como clave privada. La clave pública se genera mediante el producto $P = k \times G$.

2.1.3 Computación cuántica

La computación cuántica supone avance tecnológico en cuanto a capacidad de cálculo respecto a los actuales computadores. Este tipo de computación se basa en el uso de los denominados bits cuánticos (o qubits), sustituyendo así los bits utilizados en la computación clásica. Estos qubits se explotan mediante diversas características de la mecánica cuántica como la superposición, la interferencia o su entrelazamiento [1].

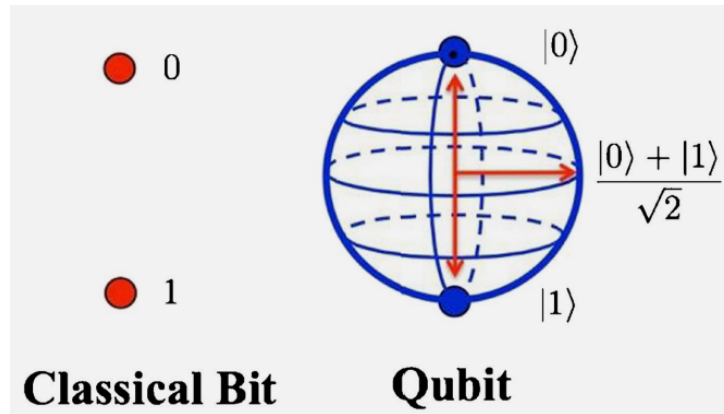


Figura 2.1: Bit clásico y qubit [1].

La forma de trabajar con estos qubits consiste en la consideración de la probabilidad de los bits de encontrarse en un valor o en otro, es decir, siguiendo la Fórmula 2.1.

$$\sum_{b \in \{0,1\}^n} p_b |b\rangle \quad (2.1)$$

A la hora de conocer con exactitud el valor de dicho qubit, es importante mencionar que, al medir su valor, dicho qubit colapsa a uno de los valores tradicionales ("0" o "1"). Teniendo en cuenta la propiedad de superposición de estos qubits, un computador cuántico es capaz de representar 16 números de 4 dígitos cada uno utilizando, únicamente, 4 qubits. Por ello, estos dispositivos pueden realizar una cantidad exponencial de cálculos a la vez. También, gracias al entrelazamiento cuántico, los qubits se encuentran relacionados unos con otros, de forma que el estado de cada qubit puede depender de otro qubit. De esta forma, se consigue una correlación entre qubits no existente en los bits convencionales.

Respecto a la capa *software*, esta debe lidiar con el ruido térmico que puede afectar al funcionamiento de estos computadores. Para evitar el impacto de este ruido, algunos computadores cuánticos deben ser utilizados a menos de una centésima de grado por encima del cero absoluto.

Esta tecnología está siendo ampliamente investigada y desarrollada con el objetivo de conseguir un mayor número de qubits funcionales y, por ende, una mayor capacidad de computación. En la Figura 2.2 se representa el aumento en número de qubits de los computadores cuánticos con el avance de los años.

Gracias a este progreso en el número de los qubits disponibles en los computadores cuánticos, estos consiguen una capacidad de cómputo muy superior a los ordenadores convencionales.

2.1.4 Criptografía postcuántica

Debido a este incremento en las capacidades de los computadores cuánticos y el hecho de que los sistemas criptográficos actuales se basan en seguridad computacional, es necesario asegurar la privacidad de los usuarios. Para ello, se ha optado por sustituir los esquemas actuales de clave pública por otros capaces de asegurar la privacidad de los usuarios frente a este nuevo paradigma.

Los nuevos algoritmos diseñados se pueden dividir en dos tipos:

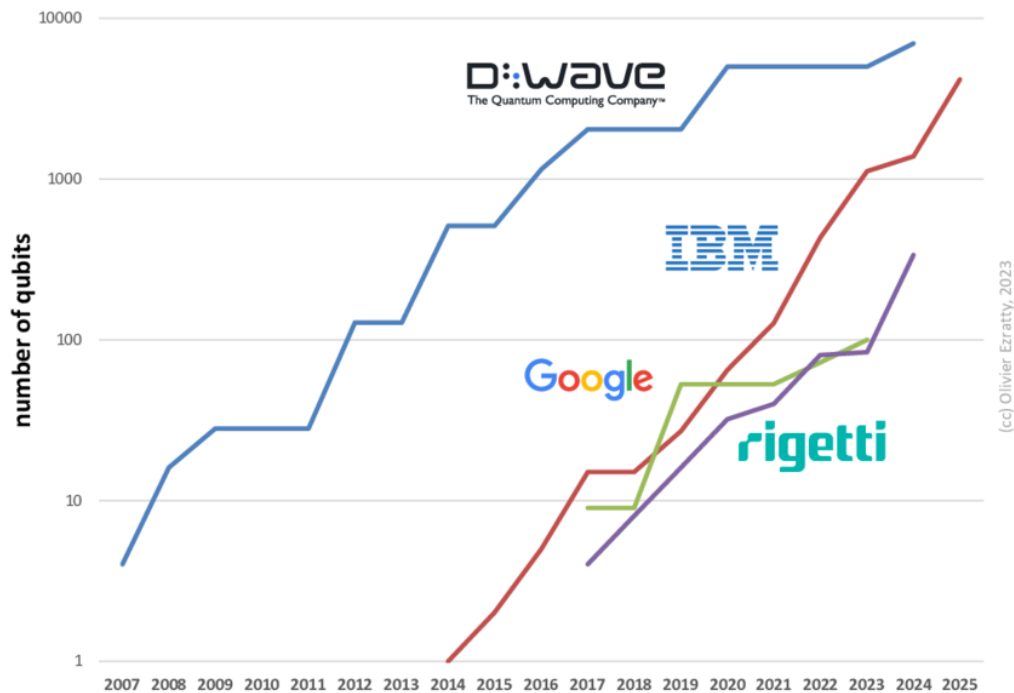


Figura 2.2: Progreso en el número de qubits de un computador cuántico [2].

- **Signature:** Este tipo de algoritmos lleva a cabo la firma de una cadena mediante el uso de claves asimétricas.
- **Key-Encapsulation Mechanism (KEM):** Este tipo de mecanismo lleva a cabo la compartición de un secreto únicamente conocido por los dos extremos de la comunicación.

El mecanismo de firma (*Signature*), hace uso de tres funciones:

- Generación de claves.
- Firma de la cadena.
- Verificación de la firma.

En la generación de las claves, se crean tanto la clave pública como la clave privada. En la firma de la cadena, se utiliza la clave privada para indicar a los receptores del mensaje que este ha sido enviado por el emisor indicado de forma inequívoca. Por último, se utiliza la verificación de la firma para comprobar que el emisor indicado es correcto y la integridad del mensaje.

El mecanismo KEM hace uso de otras tres funciones:

- Generación de claves.
- Encapsulado.
- Desencapsulado.

En la generación de las claves, se crean tanto la clave pública como la clave privada. En la función de encapsulado genera el secreto compartido y un texto cifrado. En la función de desencapsulado, se utiliza el texto cifrado anteriormente creado para obtener el secreto compartido. De esta forma, ambos usuarios disponen del secreto compartido sin haberlo intercambiado directamente.

2.1.4.1 Dilithium

Dilithium [3] es un algoritmo criptográfico de firma digital basado en retículos. Este algoritmo fue seleccionado por el NIST como estándar para un sistema de firma postcuántica [4].

El esquema Dilithium se basa en Fiat-Shamir con abortos [5] y consta de tres funciones principales: una función de generación de claves, una función de firma y una función de comprobación de firma [6]. En cuanto a la generación de las claves, se proporciona el Algoritmo 2.1. En este proceso, se genera una matriz A de dimensiones $k \times l$ y, posteriormente, se obtienen vectores aleatorios de la clave secreta.

Result: Claves pública y privada.

```

 $A \leftarrow R_q^{k \times l}$ 
 $(s_1, s_2) \leftarrow S_\eta^l \times S_\eta^k$ 
 $t := As_1 + s_2$ 
return  $(pk = (A, t), sk = (A, t, s_1, s_2))$ 

```

Algoritmo 2.1: Generación de claves pública y privadas en Dilithium [6].

En lo respectivo a la firma, esta se muestra en el algoritmo 2.2. En él, se genera un vector de enmascaramiento de polinomios y con coeficientes menores que γ_1 . El parámetro γ_1 se escoge de forma que sea lo suficientemente grande como para no revelar la clave secreta pero no demasiado grande como para que la firma sea fácilmente falsificable. A continuación, establece w_1 como los bits de mayor orden del cómputo Ay y obtiene c como la función resumen del mensaje M y w_1 . Para evitar que z filtre la clave privada, se lleva a cabo una comprobación de que los componentes de z son menores que $\gamma_1 - \beta$. En caso contrario, se reinicia el procedimiento.

Data: Clave privada y mensaje a firmar.

Result: Firma digital.

```

 $z := \perp$ 
while  $z = \perp$  do
   $y \leftarrow S_{\gamma_1-1}^l$ 
   $w_1 := HighBits(Ay, 2\gamma_2)$ 
   $c \in B_\tau := H(M || w_1)$ 
   $z := y + cs_1$ 
  if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|LowBits(Ay - cs_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$  then
     $z := \perp$ 
return  $\sigma = (z, c)$ 

```

Algoritmo 2.2: Firma en Dilithium [6].

Finalmente, para el sistema de verificación de firma, se dispone del algoritmo 2.3. En esta función, se computa w'_1 como los bits de mayor orden de Az y acepta si todos los coeficientes de z menores que $\gamma_1 - \beta$ y si c es el resultado de la función resumen del mensaje M y w'_1 .

Data: Clave pública y mensaje firmado.

Result: Mensaje original.

```

 $w'_1 := HighBits(Az, 2\gamma_2)$ 
if  $\|z\|_\infty < \gamma_1 - \beta$  and  $[c = H(M || w'_1)]$  then
  return

```

Algoritmo 2.3: Verificación de firma en Dilithium [6].

Los algoritmos 2.1, 2.2 y 2.3 son la versión simplificada pero menos eficiente del algoritmo. Se ha decidido explicar brevemente estos diseños para otorgar un conocimiento aceptable pero no total ya que, con el entendimiento de dichos algoritmos, es suficiente.

Dilithium consta de tres versiones: Dilithium2, Dilithium3 y Dilithium5. La diferencia entre estas consiste en el tamaño de las claves y firmas generadas. En la tabla 2.1 se muestran los distintos tamaños a tener en cuenta de cada versión del algoritmo. Además, se indica la seguridad ante el ataque Quantum Core-SVP tanto con Short Integer Solution (SIS) como con Learning With Errors (LWE).

	Dilithium2	Dilithium3	Dilithium5
Clave pública (bytes)	1312	1952	2592
Clave privada (bytes)	2528	4000	4864
Firma (bytes)	2420	3293	4595
Seguridad ante LWE (bits)	112	165	229
Seguridad ante SIS (bits)	112	169	241

Tabla 2.1: Diferencias entre versiones de Dilithium [6].

2.1.4.2 Kyber

Result: Claves pública y privada.

```

 $d \leftarrow B^{32}$ 
 $\rho\sigma := G(d)$ 
 $N := 0$ 
for  $i$  from 0 to  $k-1$  do
  for  $j$  from 0 to  $k-1$  do
     $\hat{A}[i][j] := Parse(XOF(\rho, j, i))$ 
for  $i$  from 0 to  $k-1$  do
   $s[i] := CBD_{\eta_1}(PRF(\sigma, N))$ 
   $N := N + 1$ 
for  $i$  from 0 to  $k-1$  do
   $e[i] := CBD_{\eta_1}(PRF(\sigma, N))$ 
   $N := N + 1$ 

```

Algoritmo 2.4: Generación de claves en Kyber [7].

	Kyber512	Kyber768	Kyber1024
Clave pública (bytes)	800	1184	1568
Clave privada (bytes)	1632	2400	3168
Texto cifrado (bytes)	768	1088	1568
Secreto compartido (bytes)	32	32	64
Core-SVP Quantum Hardness (bits)	107	165	232

Tabla 2.2: Diferencias entre versiones de Kyber [7].

2.1.4.3 Sphincs+

Sphincs+ es un algoritmo de firma basado en un esquema de firma única diseñado por Lamport [8] en la década de 1970, el cual fue mejorado para desarrollar el sistema XMSS. Este algoritmo fue elegido por el NIST como estándar en algoritmos de firma.

Sphincs+ dispone de gran cantidad de variables, a diferencia del resto de algoritmos. En este algoritmo, ofrece una elección entre optimización para tamaño (terminación “s”) u optimización para velocidad (terminación “f”). Además, permite elegir entre una implementación simple (sin considerar máscaras) o robusta (considerando máscaras). Por último, permite la elección entre el uso de distintas funciones resumen: SHA2, SHAKE o Haraka. En la tabla 2.3 se puede comprobar los tamaños de los distintos elementos que componen este algoritmo.

	Clave pública (bytes)	Clave privada (bytes)	Firma (bytes)
Sphincs-128s	32	64	7856
Sphincs-128f	32	64	17088
Sphincs-192s	48	96	16224
Sphincs-192f	48	96	35664
Sphincs-256s	64	128	29792
Sphincs-256f	64	128	49856

Tabla 2.3: Tamaños de las diferentes versiones de Sphincs+ [9].

2.1.4.4 McEliece

El algoritmo McEliece se trata de un algoritmo KEM que fue diseñado para resistir One Way Chosen Plaintext Attack (OW-CPA), de forma que un atacante no puede obtener de forma eficiente la clave privada partiendo de un texto cifrado y la clave pública. Este algoritmo ha mantenido una seguridad considerablemente estable pese a los muchos ataques intentados. Inicialmente, se diseñó para obtener 64 bits de seguridad, aunque este factor escala fácilmente para estar al día con los avances en la computación.

Algunos de los parámetros importantes en este algoritmo son n como la longitud de código y k como dimensión de código, ambos especificados de acuerdo a la versión del algoritmo.

Para la generación de las claves pública y privada se lleva a cabo el siguiente proceso [10]:

1. Se genera una cadena aleatoria y uniforme s con longitud n bits.
2. Se genera un polinomio aleatorio, uniforme e irreducible $g(x) \in F_q[x]$ de grado t .
3. Se genera una secuencia aleatoria y uniforme $(\alpha_1, \alpha_2, \dots, \alpha_n)$ con n elementos distintos de F_q .
4. Se define $\Gamma = (g, \alpha_1, \alpha_2, \dots, \alpha_n)$.
5. Se ejecuta $MatGen(\Gamma) = (T, c_{n-k-\mu+1}, \dots, c_{n-k}, \Gamma')$ donde MatGen es una función propia del algoritmo para la generación de una matriz.
6. La clave pública está representada por T mientras que la clave privada se compone por (Γ', s) .

En cuanto a la generación del encapsulado, se deben llevar a cabo los siguientes pasos [10]:

1. Se genera un vector aleatorio y uniforme $e \in F_2^n$ de peso t .
2. Se define $H = (I_{n-k}|T)$ y se computa $C_0 = He \in F_2^{n-k}$.
3. Se calcula $C_1 = H(2, e)$, donde H representa una función resumen.
4. Se computa $K = H(1, e, C)$.
5. Se obtiene el texto cifrado C y el secreto compartido K .

Respecto al desencapsulado, se llevan a cabo las siguientes operaciones [10]:

1. Se separa el texto cifrado C en (C_0, C_1) con $C_0 \in F_2^{n-k}$ y $C_1 \in F_2^l$.
2. Se asigna $b \leftarrow 1$.
3. Se extraen $s \in F_2^n$ y $\Gamma' = (g, \alpha'_1, \alpha'_2, \dots, \alpha'_n)$ de la clave privada.
4. Se extiende C_0 a $v = (C_0, 0, \dots, 0) \in F_2^n$ y se busca la única clave c definida por Γ' que se encuentra a menor distancia t de v .
5. Se asigna $e = v + c$ y, si no se cumple que $wt(e) = t$ y $C_0 = He$, $e \leftarrow \perp$.

6. Si $e = \perp$, $e \leftarrow s$ y $b \leftarrow 0$.
7. Se calcula $C'_1 = H(2, e)$ donde H es una función resumen.
8. Si $C'_1 \neq C_1$, $e \leftarrow s$ y $b \leftarrow 0$.
9. Se obtiene $K = H(b, e, C)$, donde K es el secreto compartido.

Este algoritmo emplea varias versiones, cada una con distintas longitudes de clave y textos cifrados. Estas diferencias se muestran en la Tabla 2.4.

Versiones	348864	460896	6688128	6960119	8192128
Clave pública (bytes)	261120	524160	1044992	1047319	1357824
Clave privada (bytes)	6492	13068	13932	13948	14120
Texto cifrado (bytes)	96	156	208	194	208
Secreto compartido (bytes)	32	32	32	32	32

Tabla 2.4: Diferencias entre versiones de McEliece [10].

2.1.4.5 HQC

El algoritmo HQC es un KEM que provee indistinguibilidad del texto cifrado. En la tabla 2.5 se muestran las características de las distintas versiones de este algoritmo.

	HQC-128	HQC-192	HQC-256
Clave pública (bytes)	2249	4522	7245
Clave privada (bytes)	2305	4586	7137
Texto cifrado (bytes)	4433	8978	14421
Secreto compartido (bytes)	64	64	64

Tabla 2.5: Diferencias entre versiones de HQC [11].

2.1.4.6 Falcon

Falcon (FAst Fourier Lattice-based COmpact over NTRU) es un algoritmo de firma basado en retículos. Este algoritmo fue escogido por el NIST en su estándar de algoritmos postcuánticos, en la categoría de esquemas de firma digital. En la tabla 2.6 se muestran los distintos tamaños de los componentes del algoritmo así como la seguridad de las distintas versiones del mismo.

	Falcon-512	Falcon-1024
Clave pública (bytes)	897	1783
Clave privada (bytes)	1281	2305
Firma (bytes)	752	1462
Core-SVP Quantum Hardness (bits)	108	252

Tabla 2.6: Diferencias entre versiones de Falcon [12].

2.2 IoT

Una vez comentados los algoritmos empleados a lo largo de este proyecto, se continúa explicando los distintos dispositivos con los que se ha llevado a cabo este trabajo.

2.2.1 ESP32

El primero de estos dispositivos es el System On a Chip (SoC) ESP32, desarrollado por Espressif Systems. Más concretamente, se utilizará la versión ESP32-WROOM-32E [13]. En la Figura 2.3, se muestra el sistema Devkitc v4 que hace uso del dispositivo ESP32 anteriormente mencionado.

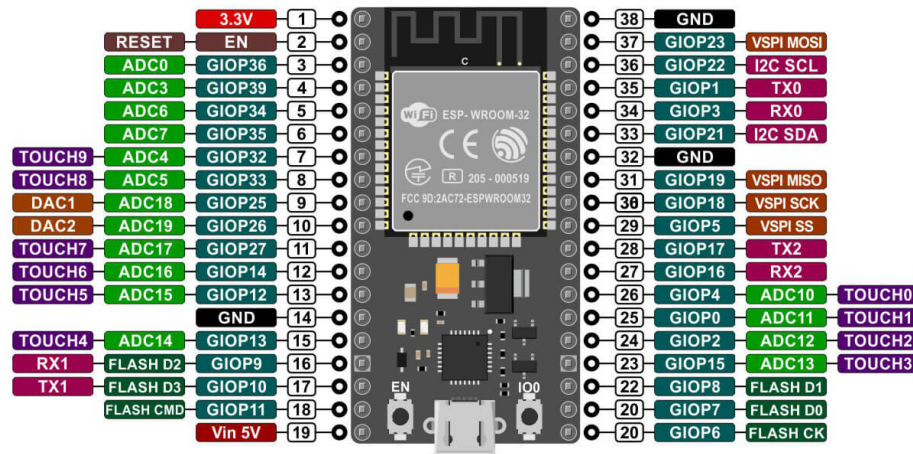


Figura 2.3: ESP32-WROOM-32E en Devkitc v4 con *pinout* [14].

Esta versión del SoC dispone de un microprocesador de 32-bit Xtensa LX6 con 2 núcleos a una frecuencia máxima de 240 MHz. En cuanto a memoria, incluye 520 kB de memoria SRAM, 448 kB de memoria ROM y 4, 8 o 16 MB de memoria *flash*.

Respecto a las comunicaciones, este dispositivo puede hacer uso del estándar 802.11 b/g/n al igual que Bluetooth v4.2 y Bluetooth LE. Para estos estándares, incluye una antena en la parte superior, como se puede apreciar en la Figura 2.3.

Por otro lado, este dispositivo incluye una serie de periféricos como Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), Secure Digital Input Output (SDIO), Inter-Integrated Circuit (I2C) y General Purpose Input Output (GPIO).

2.2.2 RP2040

Por otro lado, disponemos del microcontrolador RP2040 [15]. Este microcontrolador se puede apreciar en varios dispositivos como Raspberry Pi Pico y Raspberry Pi Pico W. La primera de ellas se puede apreciar en la Figura 2.4.



Figura 2.4: Raspberry Pi Pico [16].

Este microcontrolador consta de dos procesadores ARM Cortex-M0+ a una frecuencia de 133 MHz.

En cuanto a memoria, dispone de 264 kB de SRAM en seis bancos independientes y soporta el uso de una memoria *flash* de 16 MB a través de un bus QSPI. Además, cuenta con un controlador Direct Memory Access (DMA).

En cuanto a comunicaciones, este dispositivo cuenta 2 módulos UART, 2 controladores SPI y 2 controladores I2C.

En lo que respecta a periféricos, este microcontrolador dispone de 30 pines GPIO, de los cuales 4 pueden ser utilizados como entradas analógicas.

2.2.3 STM32

Por último, tenemos la familia de dispositivos STM32 basados en los procesadores ARM Cortex-M. Esta familia está compuesta por un gran número de dispositivos. De todos ellos, se va a proceder a explicar las características de STM32L4R5ZIT6U [17]. Esto se debe a que se utilizará el dispositivo NUCLEO-L4R5ZI, el cual hace uso de microcontrolador STM32L4R5ZIT6U. En la Figura 2.5 se puede apreciar el dispositivo NUCLEO-L4R5ZI.

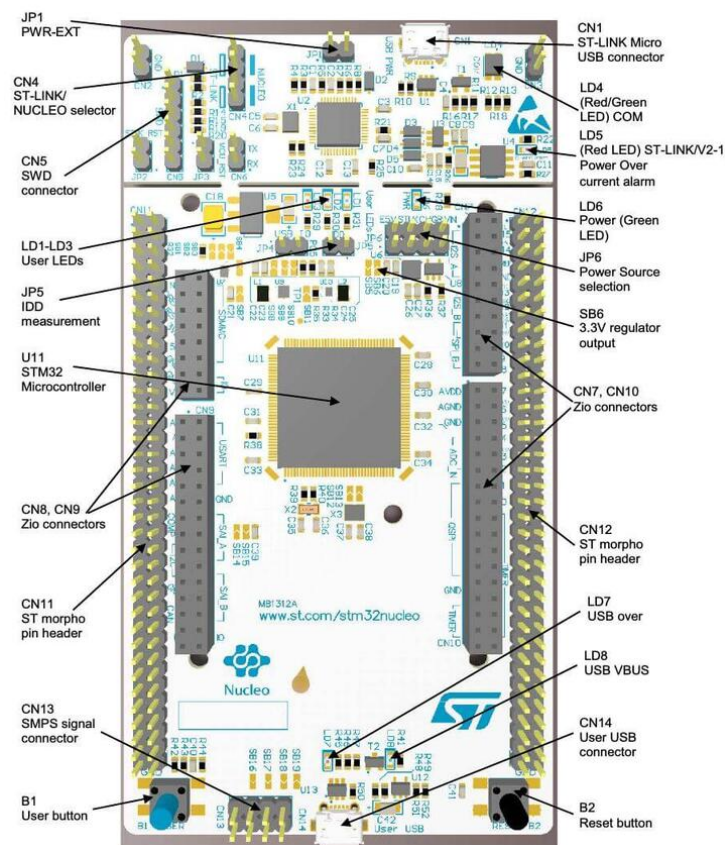


Figura 2.5: Diagrama del dispositivo NUCLEO-L4R5ZI [18].

El microcontrolador STM32L4R5ZIT6U consta de un procesador Arm Cortex-M4 de 32 bits con una frecuencia de hasta 120 MHz.

En lo referente a la memoria, este dispositivo dispone de una memoria *flash* de 2 MB, 640 kB de memoria SRAM y una interfaz para memoria externa. Además, dispone de un controlador DMA con 14 canales.

Respecto a las comunicaciones, este dispositivo contiene una interfaz Universal Serial Bus (USB), dos interfaces Serial Audio Interface (SAI), cuatro interfaces I2C, seis interfaces Universal Synchro-

nous and Asynchronous Receiver-Transmitter (USART), tres interfaces SPI y un bus Controller Area Network (CAN).

Capítulo 3

Desarrollo

En este capítulo se va a proceder a detallar el proceso llevado a cabo para alcanzar el objetivo de este trabajo.

En primer lugar, se va a tratar el proceso para seleccionar que dispositivos serán los utilizados a lo largo del proyecto. A continuación, se explica como se ha trabajado con cada uno de ellos y como se han implementado los algoritmos en cada dispositivo.

3.1 Selección de dispositivos

Para llevar a cabo una selección de los dispositivos a utilizar a lo largo de este proyecto, se ha realizado una búsqueda de los dispositivos en los que pueda existir un interés del estudio de compatibilidad y rendimiento de estos algoritmos. El criterio principal que se ha utilizado para la selección ha sido el uso que reciben los dispositivos, ya que, al disponer de una utilización más extensiva, conllevará su inclusión en funciones y operaciones sensibles y, por ende, sea necesario mantener actualizado en cuanto a esquemas de cifrado empleado.

Por ello, los dispositivos seleccionados han sido:

- **RP2040:** Este dispositivo se ha seleccionado debido a su pertenencia a la familia Raspberry Pi, su popularidad al ser un dispositivo de Raspberry y su limitada capacidad.
- **ESP32:** Este dispositivo ha sido escogido por su inmensa popularidad, su inclusión en gran cantidad de proyectos Do It Yourself (DIY) y su capacidad de conectarse a redes Wi-Fi.
- **STM32:** Este dispositivo supone unas mayores capacidades que los anteriores y una amplio uso en el ámbito industrial, lo cuál genera la necesidad de tratar datos de forma segura.

3.2 ESP32

El primer dispositivo con el que se ha trabajado ha sido el dispositivo ESP32. Para poder trabajar con él, primero se debe seleccionar la herramienta que se va a emplear, entre las cuales se debe tener en cuenta tres principales posibilidades: Arduino [19], ESP-IDF [20] y PlatformIO [21]. De las tres opciones mencionadas anteriormente, tanto Arduino como PlatformIO hacen uso de un gran número de librerías, abstrayendo aspectos de menor nivel con el objetivo de hacer más sencilla la experiencia de usuario. En cambio, ESP-IDF no incluye estas librerías, lo cual permite un mejor rendimiento del *software* desarrollado. Por ello, se ha decidido hacer uso de la herramienta ESP-IDF para trabajar con el dispositivo ESP32.

3.2.1 ESP-IDF

Teniendo en cuenta que se va a utilizar este entorno, es importante conocer en que se basa para lograr su funcionamiento y como se ha de trabajar con él.

3.2.1.1 IDF FreeRTOS

Esta herramienta se basa en el sistema operativo Free Real-Time Operating System (FreeRTOS), el cual es un sistema operativo de tiempo real de código abierto. Para dar soporte a los dispositivos basados en ESP32, Espressif ha desarrollado una versión de este sistema operativo con el objetivo de soportar los dos núcleos que incluyen los dispositivos. Esta versión se denomina IDF FreeRTOS [22]. Más específicamente, esta versión del sistema operativo está basada en la versión Vanilla FreeRTOS v10.5.1.

La versión IDF FreeRTOS soporta Symmetric MultiProcessing (SMP). Esta se basa en múltiples núcleos conectados a una memoria compartida y orquestrados por un sistema operativo. Estos núcleos cuentan con sus registros e interrupciones de forma independiente al resto de núcleos. Debido a que todos los núcleos pueden ejecutar el mismo código al compartir memoria, se obtienen múltiples hilos, consiguiendo una mayor capacidad computacional del dispositivo.

3.2.1.2 Instalación

El primer paso para utilizar esta herramienta es la instalación de la misma. Para ello, se ha seguido la guía oficial en la cuál se indican los comandos especificados en el Código 3.1.

Código 3.1: Instalación de ESP-IDF [23].

```
1  #Prerrequisitos
2  sudo apt-get install git wget flex bison gperf python3 python3-pip python3-venv cmake ninja-build ccache
   libffi-dev libssl-dev dfu-util libusb-1.0-0
3
4  #Descarga de la herramienta
5  mkdir -p ~/esp
6  cd ~/esp
7  git clone -b v5.3 --recursive https://github.com/espressif/esp-idf.git
8
9  #Instalación de la herramienta
10 cd ~/esp/esp-idf
11 ./install.sh esp32
12
13 #Variables de entorno
14 . $HOME/esp/esp-idf/export.sh
15 alias get_idf='. $HOME/esp/esp-idf/export.sh'
```

En el Código 3.1, se muestra en primer lugar el comando necesario para llevar a cabo la instalación de los paquetes necesarios para ejecutar correctamente esta herramienta. A continuación, se lleva a cabo la descarga de la herramienta desde el repositorio oficial de GitHub. En este caso, se descarga la versión 5.3 del *software*. Posteriormente, se instalan estas herramientas mediante la ejecución del *script* `install.sh`. Por último, se definen las variables de entorno necesarias para la ejecución de la herramienta mediante el *script* `export.sh`. En el caso de utilizar varias veces esta herramienta, Espressif recomienda establecer el alias `get_idf` para mayor rapidez.

3.2.2 ESP-Prog

Con el objetivo de poder depurar los distintos algoritmos implementados, se requiere dispositivo que permita este tipo de operación, ya que el ESP32 no incluye uno por defecto. Por ello, se ha decidido hacer uso del depurador ESP-Prog [24], el depurador oficial para dispositivos ESP32.

Este dispositivo cuenta con dos buses de datos como se puede apreciar en la Figura 3.1.

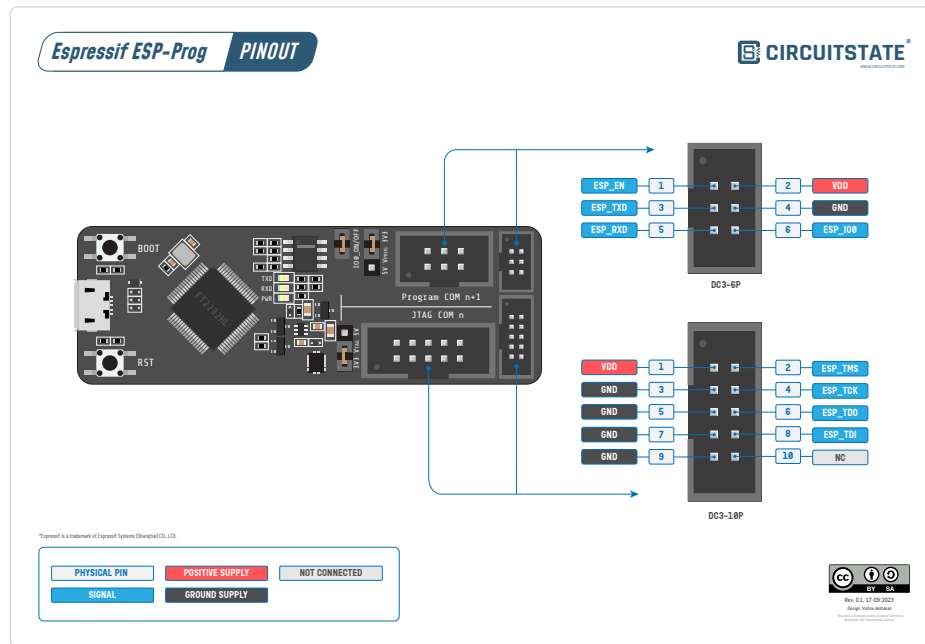


Figura 3.1: Diagrama del dispositivo ESP-Prog [25].

Para poder utilizarlo correctamente, se deberá llevar a cabo la siguiente conexión [26]:

- Pin 1 (V_{DD}) de DC3-10P de ESP-Prog a pin 3V3 del dispositivo ESP32.
- Pin 2 (ESP_TMS) de DC3-10P de ESP-Prog al pin 14 del dispositivo ESP32.
- Pin 3 (GND) de DC3-10P de ESP-Prog a pin GND del dispositivo ESP32.
- Pin 4 (ESP_TCK) de DC3-10P de ESP-Prog al pin 13 del dispositivo ESP32.
- Pin 6 (ESP_TD0) de DC3-10P de ESP-Prog al pin 15 del dispositivo ESP32.
- Pin 8 (ESP_TD1) de DC3-10P de ESP-Prog al pin 12 del dispositivo ESP32.
- Pin 1 (ESP_EN) de DC3-6P de ESP-Prog a pin EN del dispositivo ESP32.
- Pin 3 (ESP_TXD) de DC3-6P de ESP-Prog a pin TX del dispositivo ESP32.
- Pin 5 (ESP_RXD) de DC3-6P de ESP-Prog a pin RX del dispositivo ESP32.
- Pin 6 (ESP_IO0) de DC3-6P de ESP-Prog al pin 0 del dispositivo ESP32.

A la hora de cargar el *software*, se deberá hacer mediante una conexión JTAG. Sin embargo, si no se utilizase el dispositivo ESP-Prog, se debería emplear una conexión UART.

3.2.3 Dilithium

En primer lugar, se debe obtener el código de referencia de este algoritmo sobre el cual trabajar a continuación. Este código se ha obtenido desde el repositorio oficial de CRYSTALS-Dilithium [27]. Dentro de este repositorio, se ha utilizado la implementación de referencia *ref*, ya que la restante, *avx2*, se encuentra optimizada para procesadores Intel con arquitectura x86.

Una vez se ha seleccionado el código a utilizar, se debe crear un proyecto que lo incluya. Para ello, se ha creado un nuevo proyecto utilizando la extensión de Espressif en VSCode. Los ficheros de este repositorio se han organizado tal y como se puede apreciar en la Figura 3.2.

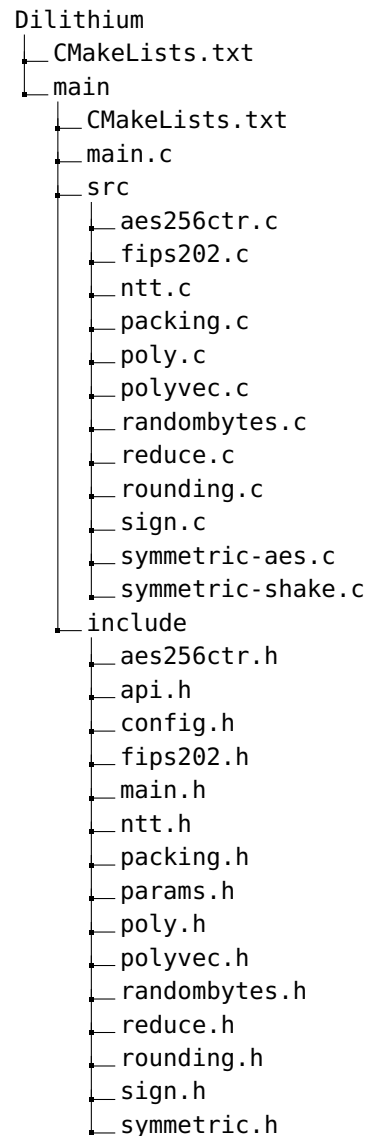


Figura 3.2: Árbol de ficheros del proyecto Dilithium.

3.2.3.1 Ficheros CMake

En este árbol, el archivo con ruta Dilithium/CMakeLists.txt únicamente especifica la versión mínima del módulo CMake para evitar posibles incompatibilidades, la inclusión del archivo `project.cmake` y la definición del nombre del proyecto. Todo esto se refleja en el Código 3.2. El contenido de este fichero se ha creado teniendo en cuenta los ficheros análogos del resto de proyectos que da ESP-IDF como ejemplo.

Código 3.2: Archivo Dilithium/CMakeLists.txt.

```

1 cmake_minimum_required(VERSION 3.16)
2
3 include($ENV{IDF_PATH}/tools/cmake/project.cmake)
4 project(Dilithium)
  
```

Dentro del directorio `main`, se han creado dos subdirectorios: `src` y `include`. En el primero de ellos, se incluyen todos los archivos de código fuente de los que hace uso el algoritmo mientras que en el segundo de ellos se incluyen todas las librerías necesarias para su correcto funcionamiento.

Es necesario incluir, dentro del directorio `main`, otro archivo `CMakeLists.txt` en el cual se especifiquen los archivos a incluir en la compilación del *software*. Por ello, se han incluido todos los archivos del directorio `src` como archivos fuente y el directorio `include` como directorio de librerías, tal y como se muestra en el Código 3.3.

Código 3.3: Archivo `Dilithium/main/CMakeLists.txt`.

```
1 idf_component_register(SRCS "main.c" "./src/sign.c" "./src/packing.c" "./src/polyvec.c" "./src/poly.c" "
  ./src/ntt.c" "./src/reduce.c" "./src/rounding.c" "./src/symmetric-shake.c" "./src/symmetric-aes.c" "
  ./src/fips202.c" "./src/aes256ctr.c" "./src/randombytes.c"
2 INCLUDE_DIRS "include")
```

3.2.3.2 Generación de números aleatorios

Un aspecto esencial consiste en la generación de número aleatorios. Para ello, se dispone del archivo `Dilithium/main/src/randombytes.c`. En él, se incluye la definición de la función `randombytes` (utilizada para la generación de números aleatorios de una determinada longitud) tanto para un dispositivo utilizando el sistema operativo Windows o uno basado en Linux, como se puede apreciar en el Código 3.4. Inicialmente, se incluye la versión del sistema Linux ya que la compilación se lleva a cabo en un dispositivo con el sistema operativo Ubuntu 22.04 LTS.

Código 3.4: Archivo `Dilithium/main/src/randombytes.c`.

```
1 #ifdef _WIN32
2 void randombytes(uint8_t *out, size_t outlen) {
3     HCrypTProv ctx;
4     size_t len;
5
6     if(!CryptAcquireContext(&ctx, NULL, NULL, PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
7         abort();
8
9     while(outlen > 0) {
10         len = (outlen > 1048576) ? 1048576 : outlen;
11         if(!CryptGenRandom(ctx, len, (BYTE *)out))
12             abort();
13
14         out += len;
15         outlen -= len;
16     }
17
18     if(!CryptReleaseContext(ctx, 0))
19         abort();
20 }
21 #elif defined(__linux__) && defined(SYS_getrandom)
22 void randombytes(uint8_t *out, size_t outlen) {
23     ssize_t ret;
24
25     while(outlen > 0) {
26         ret = syscall(SYS_getrandom, out, outlen, 0);
27         if(ret == -1 && errno == EINTR)
28             continue;
29         else if(ret == -1)
30             abort();
31
32         out += ret;
33         outlen -= ret;
34     }
35 }
```

Teniendo esto en cuenta, es necesaria la inclusión de una opción indicada para el dispositivo ESP32 y la compilación de esta en vez de la diseñada para los sistemas basados en Linux. Para ello, se ha añadido la definición especificada en el Código 3.5. En ella, se utiliza la función

`esp_fill_random` [28] la cual genera un número aleatorio de longitud `outlen` y lo almacena en `out`. Estos números aleatorios son generados mediante una fuente *hardware*.

Código 3.5: Generación de números aleatorios para ESP32.

```

1  #elif defined(ESP32)
2  #include "esp_random.h"
3
4  void randombytes(uint8_t *out, size_t outlen) {
5      esp_fill_random(out, outlen);
6  }
7  #endif

```

3.2.3.3 Fichero de comprobación

Una vez se han incluido y referenciado todos los archivos, es necesario crear una prueba que nos permita comprobar la ejecución del algoritmo. Para ello, se ha creado el archivo `main.c`, el cual está formado por el contenido mostrado en el Código 3.6. En este fichero, se ejecuta la generación de claves (`crypto_sign_keypair`) y se comprueba si ha existido algún error en la generación. Una vez se han obtenido las claves pública (`pk`) y privada (`sk`), se lleva a cabo la firma de un mensaje previamente especificado (`m` con longitud `mrlen`) con la función `crypto_sign`. El resultado de esta operación se almacena en la variable `sm` con longitud `smlen` y se comprueba si esta firma se ha realizado correctamente. Finalmente, se lleva a cabo la función de comprobación de firma con la función `crypto_sign_open`. Una vez realizada, se almacena el mensaje recuperado en la variable `m1` con longitud `mrlen1`. Para comprobar la integridad de este mensaje, se compara la longitud y contenido del mensaje recuperado y del mensaje inicial. Si estos dos parámetros coinciden, la prueba ha sido exitosa.

Código 3.6: Archivo Dilithium/main/main.c.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  #include "api.h"
5  #include "main.h"
6
7  void app_main(void)
8  {
9      printf("Inicio con la opción %d\n\r", DILITHIUM_MODE);
10
11      //Generación de claves
12      uint8_t *pk = (uint8_t*) malloc(CRYPTO_PUBLICKEYBYTES * sizeof(uint8_t));
13      uint8_t *sk = (uint8_t*) malloc(CRYPTO_SECRETKEYBYTES * sizeof(uint8_t));
14      if (crypto_sign_keypair(pk, sk) != 0) {
15          printf("Generacion de claves fallida\n\r");
16      } else {
17          printf("Generacion de claves exitosa (%d bytes de clave publica y %d bytes de clave privada)\n\r",
18              CRYPTO_PUBLICKEYBYTES, CRYPTO_SECRETKEYBYTES);
19      }
20
21      //Firma de mensaje
22      uint8_t m[] = "Esto es una prueba de la firma de mensajes utilizando Dilithium.";
23      size_t mlen = sizeof(m), smlen;
24      uint8_t *sm = (uint8_t *)calloc(mlen+CRYPTO_BYTES, sizeof(uint8_t));
25      if (crypto_sign(sm, &smlen, m, mlen, sk) != 0) {
26          printf("Firma de mensaje fallida\n\r");
27      } else {
28          printf("Firma del mensaje exitosa (%d bytes de firma)\n\r", mlen+CRYPTO_BYTES);
29      }
30
31      //Comprobación de la firma
32      size_t mlen1;
33      uint8_t *m1 = (uint8_t *)calloc(mlen+CRYPTO_BYTES, sizeof(uint8_t));
34      if (crypto_sign_open(m1, &mlen1, sm, smlen, pk) != 0) {

```



```

34     printf("Comprobacion de la firma fallida\n\r");
35 } else {
36     if (m1en != m1en1) {
37         printf("Longitud del mensaje original distinta a la longitud del mensaje recuperado\n\r");
38     } else if (memcmp(m, m1, m1en)) {
39         printf("Contenido del mensaje original distinto al contenido del mensaje recuperado\n\r");
40     } else {
41         printf("Comprobacion de la firma del mensaje exitosa (%d bytes de mensaje)\n\r", m1en1);
42     }
43 }
44
45 //Liberación de la memoria reservada
46 free(pk);
47 free(sk);
48 free(m1);
49 free(sm);
50 }

```

Como se puede observar en el Código 3.6, se hace referencia al fichero `main.h`. Este fichero únicamente contiene la especificación de la versión del algoritmo mediante la definición de `DILITHIUM_MODE`, como se muestra en el Código 3.7. También, se elimina la definición previa de `__linux__` y la definición de ESP32, cuyo motivo se trata en la apartado 3.2.3.2.

Código 3.7: Archivo Dilithium/main/include/main.h.

```

1  #ifndef MAIN_H
2  #define MAIN_H
3
4  //Necesario para generación de números aleatorios
5  #ifdef __linux__
6      #undef __linux__
7  #endif
8  #ifndef ESP32
9      #define ESP32
10 #endif
11
12 #ifndef DILITHIUM_MODE
13     #define DILITHIUM_MODE 3 /*{2,3,5}*/
14 #endif
15 #endif

```

3.2.3.4 Uso de memoria dinámica

Una vez realizados todos los pasos anteriores, el siguiente paso consiste en la ejecución de la comprobación del algoritmo. Al ejecutarlo, se comprueba que la ejecución se bloquea en la generación de claves sin mostrar ningún código de error. Llevando a cabo distintas comprobaciones, se encontró que el error sucedía en la función `KeccakF1600_StatePermute` dentro del archivo `fips202.c`, aunque no se pudo concretar la instrucción exacta en la que sucedía el error. Por ello, se ha requerido utilizar el depurador ESP-Prog. A través del depurador, se ha concluido que el error surge del error que se aprecia en la Captura ??.

Según se ha encontrado en distintas entradas del foro oficial de ESP32 [29] [30] [30], este error se debe, sin duda, a un problema en la memoria del dispositivo. En una de estas entradas, se sugiere la posibilidad de un *overflow* del *stack* sea el causante de este problema. Este problema surge del hecho de que, antes de ejecutar el programa de prueba, se realiza una reserva de memoria automáticamente y la cantidad de memoria reservada es inferior a la necesitada posteriormente. Por ello, se debe hacer uso de la memoria *heap* en vez de la memoria *stack*. Con esta idea en mente, se ha llevado a cabo la modificación de la implementación inicial para hacer uso de memoria dinámica y evitar el *overflow* del *stack* al requerir un tamaño mucho menor de esta memoria. De esta forma, la cantidad de memoria reservada inicialmente no será sobrepasada por la cantidad de memoria en *stack* requerida.

Las primeras variables en ser modificadas han sido las incluidas en el archivo `main.c`, las cuales representan las claves pública, privada y firma del mensaje y, por ende, una gran cantidad de datos (1312, 2528 y 2420 bytes respectivamente en Dilithium2). Esto se puede apreciar en el Código 3.6.

Código 3.8: Modificación de la función `crypto_sign_keypair` en el archivo `Dilithium/main/src/sign.c`.

```

1  int crypto_sign_keypair(uint8_t *pk, uint8_t *sk) {
2      uint8_t seedbuf[2*SEEDBYTES + CRHBYTES];
3      uint8_t tr[SEEDBYTES];
4      const uint8_t *rho, *rho_prime, *key;
5      polyvecl *mat = (polyvecl*) malloc(K * sizeof(polyvecl));
6      polyvecl *s1 = (polyvecl*) malloc(sizeof(polyvecl));
7      polyvecl *s1hat = (polyvecl*) malloc(sizeof(polyvecl));
8      polyveck *s2 = (polyveck*) malloc(sizeof(polyveck));
9      polyveck *t1 = (polyveck*) malloc(sizeof(polyveck));
10     polyveck *t0 = (polyveck*) malloc(sizeof(polyveck));
11
12     ...
13
14     free(mat);
15     free(s1);
16     free(s1hat);
17     free(s2);
18     free(t1);
19     free(t0);
20
21     return 0;
22 }

```

En la modificación mostrada en el Código 3.8, se han priorizado las variables de mayor tamaño, como en este caso todas las variables del tipo `polyvecl`, ya que esta estructura consta de un tamaño de 4 kB en Dilithium2, 6 kB en Dilithium3 y 8 kB en Dilithium5. También, las variables de tipo `polyveck` ya que esta estructura requiere de un tamaño de 4 kB en Dilithium2, 5 kB en Dilithium3 y 7 kB en Dilithium5.

Código 3.9: Modificación de la función `crypto_sign_signature` en el archivo `Dilithium/main/src/sign.c`.

```

1  int crypto_sign_signature(uint8_t *sig, size_t *siglen, const uint8_t *m, size_t mlen, const uint8_t *sk)
2  {
3      unsigned int n;
4      uint8_t seedbuf[3*SEEDBYTES + 2*CRHBYTES];
5      uint8_t *rho, *tr, *key, *mu, *rho_prime;
6      uint16_t nonce = 0;
7      polyvecl *mat = (polyvecl*) malloc(K * sizeof(polyvecl));
8      polyvecl *s1 = (polyvecl*) malloc(sizeof(polyvecl));
9      polyvecl *y = (polyvecl*) malloc(sizeof(polyvecl));
10     polyvecl *z = (polyvecl*) malloc(sizeof(polyvecl));
11
12     polyveck *t0 = (polyveck*) malloc(sizeof(polyveck));
13     polyveck *s2 = (polyveck*) malloc(sizeof(polyveck));
14     polyveck *w1 = (polyveck*) malloc(sizeof(polyveck));
15     polyveck *w0 = (polyveck*) malloc(sizeof(polyveck));
16     polyveck *h = (polyveck*) malloc(sizeof(polyveck));
17     poly *cp = (poly*) malloc(sizeof(poly));
18     keccak_state *state = (keccak_state*) malloc(sizeof(keccak_state));
19
20     ...
21
22     free(mat);
23     free(s1);
24     free(y);
25     free(z);
26     free(t0);
27     free(s2);
28     free(w1);

```

```

29 free(w0);
30 free(h);
31 free(cp);
32 free(state);
33
34 return 0;
35 }

```

A continuación, se modificó la función `crypto_sign_signature` mediante el Código 3.9. En este caso, se han alterado las variables de tipo `polyveck` y `polyvecl` por el mismo motivo explicado anteriormente. Además, se han adaptado las variables `cp` y `state` a pesar de no requerir una gran cantidad de memoria.

Código 3.10: Modificación de la función `crypto_sign_verify` en el archivo `Dilithium/main/src/sign.c`.

```

1  int crypto_sign_verify(const uint8_t *sig,
2                        size_t siglen,
3                        const uint8_t *m,
4                        size_t mlen,
5                        const uint8_t *pk)
6  {
7      unsigned int i;
8      uint8_t *buf = (uint8_t *)malloc((K*POLYW1_PACKEDBYTES) * sizeof(uint8_t));
9      uint8_t rho[SEEDBYTES];
10     uint8_t mu[CRHBYTES];
11     uint8_t c[SEEDBYTES];
12     uint8_t c2[SEEDBYTES];
13     poly *cp = (poly*) malloc(sizeof(poly));
14     polyvecl *mat = (polyvecl*) malloc(K * sizeof(polyvecl));
15     polyvecl *z = (polyvecl*) malloc(sizeof(polyvecl));
16
17     polyveck *t1 = (polyveck*) malloc(sizeof(polyveck));
18     polyveck *w1 = (polyveck*) malloc(sizeof(polyveck));
19     polyveck *h = (polyveck*) malloc(sizeof(polyveck));
20     keccak_state *state = (keccak_state*) malloc(sizeof(keccak_state));
21
22     ...
23
24     free(buf);
25     free(cp);
26     free(mat);
27     free(z);
28     free(t1);
29     free(w1);
30     free(h);
31     free(state);
32
33     return 0;
34 }

```

Por último, se ha modificado la función `crypto_sign_verify` tal y como se muestra en el Código 3.10. Para esta modificación se ha seguido la pauta anteriormente especificada, en la cual se modifican las variables de tipo `polyveck` y `polyvecl`. Además, en este caso, se ha modificado la variable `buf` ya que esta necesitaría 768 bytes en Dilithium2 y Dilithium3 y 1 kB en el caso de Dilithium5.

Una vez realizadas todas estas modificaciones, se han adaptado las llamadas a las funciones que componen `crypto_sign_keypair`, `crypto_sign_signature` y `crypto_sign_verify` de forma que se utilicen correctamente los punteros creados en lugar de las variables utilizadas anteriormente.

3.2.3.5 Comprobación del algoritmo

Tras completar todos los pasos indicados previamente, se lleva a cabo la ejecución del algoritmo. Esta ejecución se lleva a cabo mediante el programa mostrado en el Código 3.6. El resultado de esta ejecución se muestra en la Captura 3.3.

```
I (300) main task: Started on CPU0  
I (316) main task: Calling app_main()  
Inicio con la opción 2 de Dilithium  
Generacion de claves exitosa (1312 bytes de clave publica y 2528 bytes de clave privada)  
Firma del mensaje exitosa (2487 bytes de firma)  
Comprobacion de la firma del mensaje exitosa (67 bytes de mensaje)  
Comprobacion de la firma del mensaje = "Esto es una prueba de la firma de mensajes utilizando Dilithium."  
"
```

Figura 3.3: Comprobación de ejecución de algoritmo Dilithium.

3.2.4 Kyber

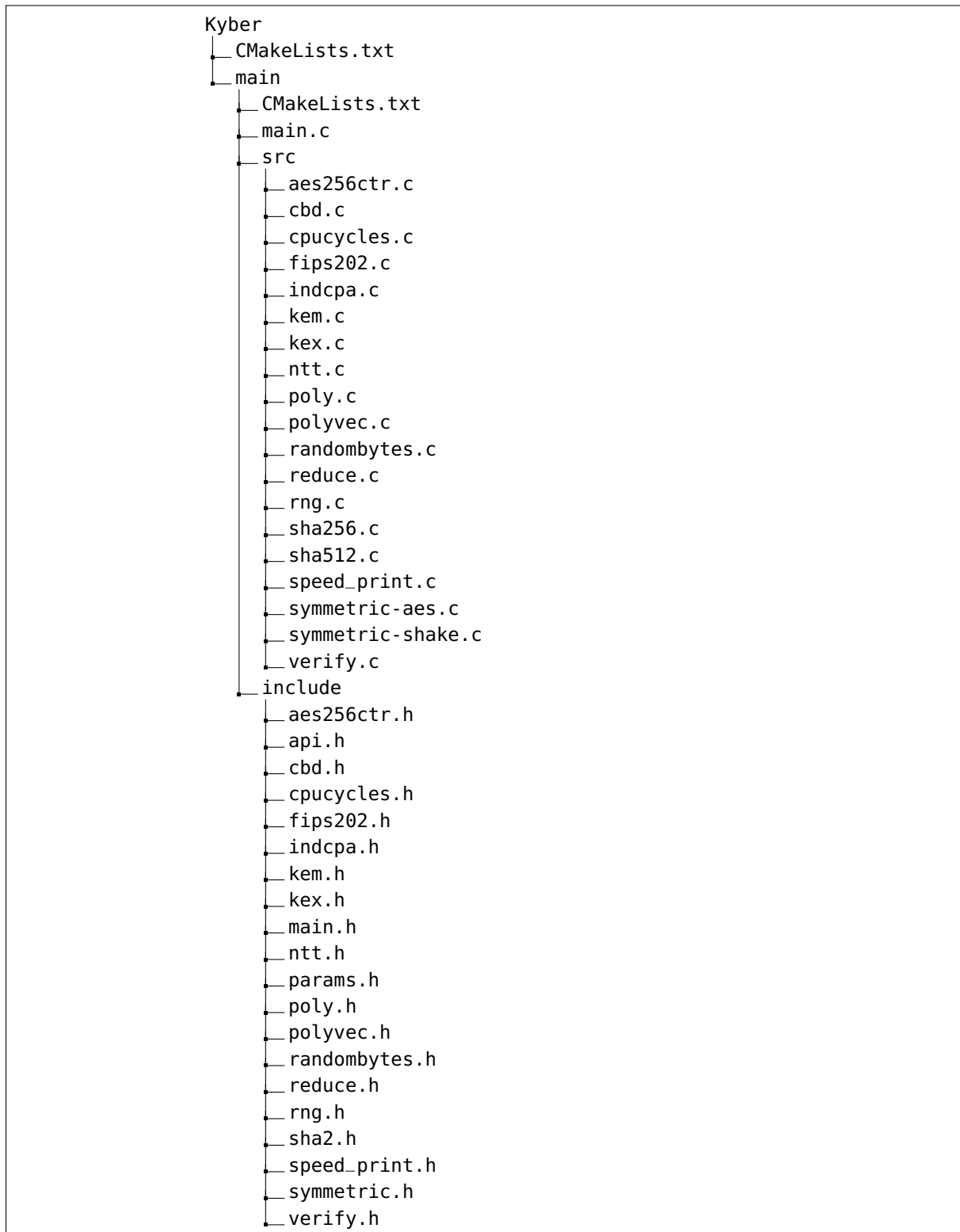


Figura 3.4: Árbol de archivos del proyecto Kyber.

Al igual que para el caso de Dilithium 3.2.3, el primer paso consiste en seleccionar la implementación a tratar. Para ello, se ha utilizado el repositorio oficial de CRYSTALS-Kyber [31]. Dentro de este código, existe tanto una implementación estándar dentro del directorio `ref` y otra optimizada para procesadores Intel x86 bajo el nombre `avx2`.

Repitiendo los pasos descritos para Dilithium, se ha procedido a crear un proyecto mediante la extensión de Espressif en VSCode. El árbol de ficheros utilizado es el mostrado en la Figura 3.4.

3.2.4.1 Ficheros CMake

Dentro de este árbol, el archivo `Kyber/CMakeLists.txt` indica la versión mínima aceptada del módulo CMake para evitar posibles incompatibilidades, el uso del archivo `project.cmake` y el nombre del proyecto. Todo esto se refleja en el Código 3.11. El contenido de este fichero se ha creado teniendo en cuenta los ficheros análogos del resto de proyectos que da ESP-IDF como ejemplo.

Código 3.11: Archivo `Kyber/CMakeLists.txt`.

```
1 cmake_minimum_required(VERSION 3.16)
2
3 include($ENV{IDF_PATH}/tools/cmake/project.cmake)
4 project(Kyber)
```

Dentro del directorio `main`, se han creado dos subdirectorios: `src` y `include`. En el primero de ellos, se incluyen todos los archivos de código fuente de los que hace uso el algoritmo mientras que en el segundo de ellos se incluyen todas las librerías necesarias para su correcto funcionamiento.

Es necesario incluir, dentro del directorio `main`, otro archivo `CMakeLists.txt` en el cual se especifiquen los archivos a incluir en la compilación del *software*. Por ello, se han incluido todos los archivos del directorio `src` como archivos fuente y el directorio `include` como directorio de librerías, tal y como se muestra en el Código 3.12.

Código 3.12: Archivo `Kyber/main/CMakeLists.txt`.

```
1 idf_component_register(SRCS "main.c" "src/kex.c" "src/kem.c" "src/indcpa.c" "src/polyvec.c" "src/poly.c"
   "src/cbd.c" "src/ntt.c" "src/reduce.c" "src/verify.c" "src/symmetric-shake.c" "src/symmetric-aes.c"
   "src/fips202.c" "src/randombytes.c"
2   INCLUDE_DIRS "include")
```

3.2.4.2 Generación de números aleatorios

Teniendo en cuenta que el dispositivo utilizado es el mismo que para el algoritmo Dilithium, la generación de números aleatorios se realiza de la misma manera que la mostrada en el Código 3.5. Además, debido a que tanto Dilithium como Kyber comparten creadores, la generación de números aleatorios es idéntica, por lo que el archivo de generación de números aleatorios `Kyber/main/src/randombytes.c` es el mismo y, por ende, la forma en la que incluir la versión para el ESP32.

3.2.4.3 Fichero de comprobación

Una vez se han incluido y referenciado todos los archivos, es necesario crear una prueba que nos permita comprobar la ejecución del algoritmo. Para ello, se ha creado el archivo `main.c`, el cual está formado por el contenido mostrado en el Código 3.13. En primer lugar, se llama a la función `crypto_kem_keypair`, encargada de generar las claves tanto pública como privada a utilizar. La clave pública se almacenará en la variable `pk` mientras que la clave privada estará representada por `sk`. Si esta generación fuese errónea, la ejecución finalizaría. En caso contrario, se llevaría a cabo la generación del secreto compartido (`ss_pub`) y del texto cifrado (`ct`) haciendo uso de la clave pública mediante la función `crypto_kem_enc`. A continuación, se utiliza la función `crypto_kem_dec`, a la cual se entrega el texto cifrado `ct` generado por la función anterior y la clave privada y devuelve el secreto compartido `ss_priv`, el cual, si la ejecución ha sido correcta, debería coincidir con el secreto generado anteriormente `ss_pub`. Para finalizar, se lleva a cabo esta verificación y se libera la memoria correspondiente a las variables creadas al inicio del programa.

Código 3.13: Archivo Kyber/main/main.c.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #include "main.h"
7  #include "kem.h"
8  #include "api.h"
9
10 #include "randombytes.h"
11
12 void app_main (void)
13 {
14     printf("Inicio con la opción %d\n\r", KYBER_K);
15
16     //Generación de claves
17     unsigned char *pk = (unsigned char*) malloc(CRYPTO_PUBLICKEYBYTES * sizeof(unsigned char));
18     unsigned char *sk = (unsigned char*) malloc(CRYPTO_SECRETKEYBYTES * sizeof(unsigned char));
19     if (crypto_kem_keypair(pk, sk) != 0) {
20         printf("Generacion de claves fallida\n\r");
21         return;
22     } else {
23         printf("Generacion de claves exitosa (%d bytes de clave publica y %d bytes de clave privada)\n\r",
24             CRYPTO_PUBLICKEYBYTES, CRYPTO_SECRETKEYBYTES);
25     }
26
27     //Generación de texto cifrado y secreto compartido
28     unsigned char *ct = (unsigned char*) malloc(CRYPTO_CIPHERTEXTBYTES * sizeof(unsigned char));
29     unsigned char *ss_pub = (unsigned char*) malloc(CRYPTO_BYTES * sizeof(unsigned char));
30     if (crypto_kem_enc(ct, ss_pub, pk) != 0) {
31         printf("Generacion de texto cifrado y secreto compartido fallida\n\r");
32         return;
33     } else {
34         printf("Generacion de texto cifrado y secreto compartido exitosa (%d bytes de texto cifrado y %d
35             bytes de secreto compartido)\n\r", CRYPTO_CIPHERTEXTBYTES, CRYPTO_BYTES);
36     }
37
38     //Generación de secreto compartido
39     unsigned char *ss_priv = (unsigned char*) malloc(CRYPTO_BYTES * sizeof(unsigned char));
40     if (crypto_kem_dec(ss_priv, ct, sk) != 0) {
41         printf("Generacion de secreto compartido fallida\n\r");
42         return;
43     } else {
44         printf("Generacion de secreto compartido exitosa (%d bytes de secreto compartido)\n\r",
45             CRYPTO_BYTES);
46     }
47
48     //Comprobación de resultados
49     if (memcmp(ss_pub, ss_priv, (unsigned int) CRYPTO_BYTES) ) {
50         printf("Comprobación fallida\n\r");
51     } else{
52         printf("¡Comprobación superada!\n\r");
53     }
54
55     //Liberación de la memoria reservada
56     free(pk);
57     free(sk);
58     free(ct);
59     free(ss_pub);
60     free(ss_priv);
61 }

```

Como se puede observar en el Código 3.13, se hace referencia al fichero main.h. Este fichero únicamente contiene la especificación de la versión del algoritmo mediante la definición de KYBER_K, como se muestra en el Código 3.14. En este código, si KYBER_K se hace uso de Kyber512, si vale 3 se utiliza Kyber768 y, si vale 4, se emplea Kyber1024. También, se elimina la definición previa de `__linux__` y la definición de ESP32, cuyo motivo se trata en la apartado 3.2.3.2.

Código 3.14: Archivo Kyber/main/include/main.h.

```

1  #ifndef MAIN_H
2  #define MAIN_H
3
4  //Necesario para generación de números aleatorios
5  #ifdef __linux__
6      #undef __linux__
7  #endif
8  #ifndef ESP32
9      #define ESP32
10 #endif
11
12 #ifndef KYBER_K
13     #define KYBER_K 3
14 #endif
15 #endif

```

3.2.4.4 Uso de memoria dinámica

De la misma manera que Dilithium suponía un *overflow* en el *stack* reservado, Kyber también, como se puede comprobar en la Figura ??.

Por ello, se han seguido los mismos pasos y se ha hecho uso de la *heap* cambiando las variables estáticas que más memoria requerían a ser variables dinámicas. El archivo en ser modificado ha sido el archivo Kyber/main/src/indcpa.c. En él, se ha cambiado la función `indcpa_keypair` tal y como se indica en el Código 3.15.

Código 3.15: Modificación de la función `indcpa_keypair` en el archivo Kyber/main/src/indcpa.c.

```

1  void indcpa_keypair(uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES], uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES])
2  {
3      unsigned int i;
4      uint8_t *buf = (uint8_t*) malloc(2*KYBER_SYMBYTES * sizeof(uint8_t));
5      const uint8_t *publicseed = buf;
6      const uint8_t *noiseseed = buf+KYBER_SYMBYTES;
7      uint8_t nonce = 0;
8
9      polyvec *a = (polyvec*) malloc(KYBER_K * sizeof(polyvec));
10     polyvec *e = (polyvec*) malloc(sizeof(polyvec));
11     polyvec *pkpv = (polyvec*) malloc(sizeof(polyvec));
12     polyvec *skpv = (polyvec*) malloc(sizeof(polyvec));
13
14     ...
15
16     free(buf);
17     free(a);
18     free(e);
19     free(skpv);
20     free(pkpv);
21 }

```

En este caso, las variables de tipo `polyvec` ocupan 1 kB en el caso de Kyber2, 1.5 kB en el caso de Kyber3 y 2 kB en el caso de Kyber4. La variable `a` ocupa un total de 2 kB para Kyber2, 3.5 kB para Kyber3 y 8 kB para Kyber4. Por ello, todas estas variables han sido modificadas para hacer uso de memoria dinámica en vez de memoria estática, siguiendo el propósito indicado en el apartado 3.2.3.4.

Código 3.16: Modificación de la función `indcpa_enc` en el archivo Kyber/main/src/indcpa.c.

```

1  void indcpa_enc(uint8_t c[KYBER_INDCPA_BYTES], const uint8_t m[KYBER_INDCPA_MSGBYTES], const uint8_t pk[
2      KYBER_INDCPA_PUBLICKEYBYTES], const uint8_t coins[KYBER_SYMBYTES])
3  {
4      unsigned int i;
5      uint8_t *seed = (uint8_t*) malloc(KYBER_SYMBYTES * sizeof(uint8_t));

```



```

5     uint8_t nonce = 0;
6
7     polyvec *sp = (polyvec*) malloc(sizeof(polyvec));
8     polyvec *pkpv = (polyvec*) malloc(sizeof(polyvec));
9     polyvec *ep = (polyvec*) malloc(sizeof(polyvec));
10    polyvec *at = (polyvec*) malloc(KYBER_K * sizeof(polyvec));
11    polyvec *b = (polyvec*) malloc(sizeof(polyvec));
12
13    poly *v = (poly*) malloc(sizeof(poly));
14    poly *k = (poly*) malloc(sizeof(poly));
15    poly *epp = (poly*) malloc(sizeof(poly));
16
17    ...
18
19    free(seed);
20    free(sp);
21    free(pkpv);
22    free(ep);
23    free(at);
24    free(b);
25    free(v);
26    free(k);
27    free(epp);
28 }

```

En el Código 3.16 se observa la modificación realizada en la función encargada de generar el secreto compartido y el texto cifrado. Para ello, se han modificado las variables del tipo `polyvec` y del tipo `poly`, aunque estas últimas requieren una cantidad de memoria mucho menor.

Código 3.17: Modificación de la función `indcpa_dec` en el archivo `Kyber/main/src/indcpa.c`.

```

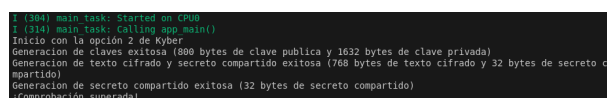
1 void indcpa_dec(uint8_t m[KYBER_INDCPA_MSGBYTES], const uint8_t c[KYBER_INDCPA_BYTES], const uint8_t sk[
  KYBER_INDCPA_SECRETKEYBYTES])
2 {
3     polyvec *b = (polyvec*) malloc(sizeof(polyvec));
4     polyvec *skpv = (polyvec*) malloc(sizeof(polyvec));
5
6     poly *v = (poly*) malloc(sizeof(poly));
7     poly *mp = (poly*) malloc(sizeof(poly));
8
9     ...
10
11    free(b);
12    free(skpv);
13    free(v);
14    free(mp);
15 }

```

La última función modificada ha sido la encargada de obtener el secreto compartido apartir del texto cifrado y la clave privada, `indcpa_dec`. Siguiendo los cambios realizados con anterioridad, se han modificado las variables del tipo `polyvec` y `poly`.

3.2.4.5 Comprobación del algoritmo

Para finalizar con este algoritmo, se ha llevado a cabo la comprobación de su funcionamiento utilizando el Código 3.13. El resultado de esta ejecución se muestra en la Captura 3.5.



```

I (304) main task: Started on CPU0
I (314) main task: Calling app main()
Inicio con la opción 2 de Kyber
Generacion de claves exitosa (800 bytes de clave publica y 1632 bytes de clave privada)
Generacion de texto cifrado y secreto compartido exitosa (768 bytes de texto cifrado y 32 bytes de secreto compartido)
Generacion de secreto compartido exitosa (32 bytes de secreto compartido)
¡Comprobación superada!

```

Figura 3.5: Comprobación de ejecución de algoritmo Kyber.

3.3 RP2040

El siguiente dispositivo a utilizar es el Lytigo T-Display [32], el cual incluye un ESP32 y un RP2040. Este dispositivo utiliza el conector USB-C para programar un elemento u otro, dependiendo si está conectado con una orientación u otra.

Para poder programar el RP2040 se ofrecen varias posibilidades en el repositorio de esta plataforma. Estas posibilidades son: Arduino, Micropython y Pico SDK. Los algoritmos a implementar están desarrollados sobre C, lo cual da un mayor control sobre los recursos del dispositivo. En cambio, Arduino está implementado en una mezcla de C y C++, por lo que puede no ser óptimo a la hora de utilizar aplicaciones exigentes en lo referente a recursos. En este aspecto, Micropython tiene el mismo defecto. Por otro lado, Pico SDK supone directamente una programación en C, lo cual consigue un mejor uso de los recursos disponibles.

3.3.1 Pico SDK

La herramienta Pico SDK [33] está desarrollada para otorgar al usuario el mayor control posible sobre el dispositivo. Este ejecuta un programa con la función `main` convencional y aporta varias librerías que permiten tratar con temporizadores, conexiones USB y sincronización.

Para la construcción de los proyectos, esta herramienta hace uso de CMake ya que es soportado por multitud de IDEs. Para poder utilizar esta herramienta, se deben seguir los pasos indicados en su repositorio oficial [34]. En primer lugar, se deben instalar los paquetes necesarios con el comando `sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-none-eabi libstdc++-arm-none-eabi-newlib`. A continuación, se debe clonar el repositorio de la herramienta con el comando `git clone https://github.com/raspberrypi/pico-sdk.git`. Posteriormente, es necesario copiar el archivo `pico_sdk_import.cmake` en el proyecto y establecer la variable de entorno `PICO_SDK_PATH` con la ruta al repositorio recién clonado. El siguiente paso consiste en crear un archivo `CMakeLists.txt` como el ejemplo mostrado en el Código 3.18 y el programa que se vaya a ejecutar en el dispositivo. Por último, se debe crear un directorio `build` mediante el comando `mkdir build && cd build && cmake ..` y, dentro del directorio `build`, ejecutar el comando `make hello_world` para construir el ejecutable del programa creado. Esto genera un archivo `.elf` para cargar a través de un depurador y un archivo `.elf2` para cargar directamente mediante la conexión USB.

Código 3.18: Ejemplo de `CMakeLists.txt` [34].

```
1 cmake_minimum_required(VERSION 3.13)
2
3 # initialize the SDK based on PICO_SDK_PATH
4 # note: this must happen before project()
5 include(pico_sdk_import.cmake)
6
7 project(my_project)
8
9 # initialize the Raspberry Pi Pico SDK
10 pico_sdk_init()
11
12 # rest of your project
```

3.3.2 HQC-128

El primer algoritmo a implementar en este dispositivo ha sido el algoritmo HQC-128. Para comenzar, se debe obtener una implementación del mismo. La implementación seleccionada ha sido la correspondiente al proyecto PQClean [35]. El código correspondiente a este algoritmo se puede encontrar en su repositorio de GitHub [36].

Una vez extraídos los archivos correspondientes a este algoritmo, se han colocado de la forma indicada en el árbol de ficheros 3.6.

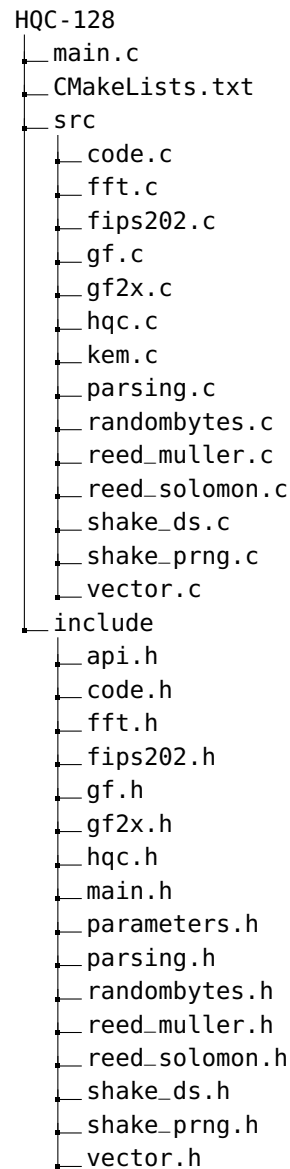


Figura 3.6: Árbol de ficheros del proyecto HQC-128.

Este proyecto incluye, al igual que los proyectos del dispositivo ESP32, todos los ficheros de código fuente en un directorio `src` y todas librerías en un directorio `include`. Además, dispone de un archivo `CMakeLists.txt` que se explica en el apartado 3.3.2.1 y un archivo de comprobación que se detalla en el apartado 3.3.2.3.

3.3.2.1 Fichero CMake

Dentro de este archivo `CMakeLists.txt`, se puede hallar el contenido del Código 3.19.

Código 3.19: Archivo HQC-128/`CMakeLists.txt`.

```

1  cmake_minimum_required(VERSION 3.13)
2
3  if (TARGET tinyusb_device)
4      add_compile_options(-Wall -Wextra -Wpedantic -Wredundant-decls -Wcast-align -Wmissing-prototypes -
5                          DPQCLEAN_NAMESPACE=PQCLEAN_HQC128_CLEAN)
6
7      add_executable(hqc-128)

```

```

7
8     target_sources(hqc-128 PUBLIC
9         ${CMAKE_CURRENT_LIST_DIR}/main.c
10        ${CMAKE_CURRENT_LIST_DIR}/src/code.c
11        ${CMAKE_CURRENT_LIST_DIR}/src/fft.c
12        ${CMAKE_CURRENT_LIST_DIR}/src/gf.c
13        ${CMAKE_CURRENT_LIST_DIR}/src/gf2x.c
14        ${CMAKE_CURRENT_LIST_DIR}/src/hqc.c
15        ${CMAKE_CURRENT_LIST_DIR}/src/kem.c
16        ${CMAKE_CURRENT_LIST_DIR}/src/parsing.c
17        ${CMAKE_CURRENT_LIST_DIR}/src/reed-muller.c
18        ${CMAKE_CURRENT_LIST_DIR}/src/reed-solomon.c
19        ${CMAKE_CURRENT_LIST_DIR}/src/shake_ds.c
20        ${CMAKE_CURRENT_LIST_DIR}/src/shake_prng.c
21        ${CMAKE_CURRENT_LIST_DIR}/src/vector.c
22        ${CMAKE_CURRENT_LIST_DIR}/src/randombytes.c
23        ${CMAKE_CURRENT_LIST_DIR}/src/fips202.c
24    )
25
26    target_include_directories(hqc-128 PUBLIC
27        ${CMAKE_CURRENT_LIST_DIR}/include)
28
29    # pull in common dependencies
30    target_link_libraries(hqc-128 pico_stdlib pico_rand)
31
32    # enable usb output, disable uart output
33    pico_enable_stdio_usb(hqc-128 1)
34    pico_enable_stdio_uart(hqc-128 0)
35
36    # create map/bin/hex/uf2 file etc.
37    pico_add_extra_outputs(hqc-128)
38
39    # add url via pico_set_program_url
40    example_auto_set_url(hqc-128)
41 elseif(PICO_ON_DEVICE)
42     message(WARNING "not building hqc-128 because TinyUSB submodule is not initialized in the SDK")
43 endif()

```

En el Código 3.19 se incluyen, en primer lugar, las opciones de compilación mediante la instrucción `add_compile_options` [37]. Esta opción se implementa para incluir las opciones de compilación que se utilizan en la implementación original del código. Todas las opciones incluidas afectan únicamente a los *warning* mostrados excepto `-DPQCLEAN_NAMESPACE`, que se utiliza para indicar el valor de `PQCLEAN_NAMESPACE` durante la ejecución del programa. Se han eliminado dos opciones de las utilizadas en la compilación de la implementación original: `-O3` y `-std=c99`. El primero de ellos indica un nivel de optimización mientras que el segundo especifica el uso del estándar ISO del lenguaje C de 1999. Estas dos opciones se han eliminado porque causan conflicto con la compilación que realiza la herramienta Pico SDK.

A continuación, se especifica el nombre que tendrá el ejecutable, en este caso `hqc-128`.

Posteriormente, con la opción `target_sources`, se indican todos los archivos de código fuente que se incluirán en la compilación del ejecutable. Como se puede apreciar, se han indicado todos los archivos contenidos en el directorio `src`.

El siguiente paso consiste en indicar el directorio con las librerías mediante la opción `target_include_directories`.

Después, se especifican dependencias comunes del *software*, como son `stdlib` y `pico_rand` para la generación de números aleatorios.

Por último, se habilita el uso de salida a través de USB y se deshabilita la salida a través de UART.

3.3.2.2 Generación de números aleatorios

Al igual que en el caso del dispositivo ESP32, es necesario adaptar la generación de números aleatorios al funcionamiento del dispositivo. En este caso, se debe utilizar la función `get_rand_32` [38] para obtener 32 bits aleatorios. Introduciendo esta función en un bucle, se puede conseguir un número aleatorio del tamaño requerido.

Código 3.20: Archivo HQC-128/src/randombytes.c.

```

1  #ifdef RP2040
2  static int randombytes_rp2040_randombytes(unsigned char *x, unsigned long long xlen){
3      uint32_t aux = 0;
4      for (int i = 0; i < xlen; i+=4){
5          aux = get_rand_32();
6          memcpy(x+i, &aux, min(4, xlen-1));
7      }
8      return 0;
9  }
10 #endif

```

Para poder utilizar correctamente la función anterior, se ha debido definir la función `min` ?? en el archivo `randombytes.h` de la forma indicada en el Código 3.21.

Código 3.21: Archivo HQC-128/include/randombytes.h.

```

1  #define min(a,b) \
2      ({ __typeof__ (a) _a = (a); \
3         __typeof__ (b) _b = (b); \
4         _a < _b ? _a : _b; })

```

3.3.2.3 Fichero de comprobación

Para poder comprobar el funcionamiento del algoritmo, se ha creado el fichero `main.c` con el contenido del Código 3.22. En este fichero, se inicializará la librería de entrada y salida para poder conocer el progreso de la ejecución del algoritmo y la inicialización de las variables a utilizar a lo largo del proceso y se llevará a cabo un bucle en el que se llame a las funciones que componen el algoritmo. En este caso, se ha utilizado los mismos nombres de variables que en el caso de Dilithium 3.2.3.3. La función `crypto_kem_keypair` es la encargada de generar la clave pública (`pk`) y privada (`sk`). Una vez se han generado las claves, se hace uso de la función `crypto_kem_enc` para generar el secreto compartido (`ss_pub`) y el texto cifrado (`ct`) con la clave pública. A continuación, se lleva a cabo la generación del secreto compartido `ss_priv` con el texto cifrado y la clave privada. Finalmente, se lleva a cabo una comprobación entre ambos secretos compartidos y, si coinciden, la ejecución ha sido exitosa.

Código 3.22: Archivo HQC-128/main.c.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stddef.h>
5
6  #include "pico/stdlib.h"
7  #include "api.h"
8  #include "main.h"
9
10 int main() {
11     stdio_init_all();
12     printf("Inicio con la opción %s\n\r", CRYPTO_ALGNAME);
13
14     //Generación de claves
15     unsigned char pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];

```

```

16
17 //Generación de texto cifrado y secreto compartido
18 uint8_t ct[CRYPTO_CIPHTEXTBYTES];
19 uint8_t ss_pub[CRYPTO_BYTES];
20
21 //Generación de secreto compartido
22 uint8_t ss_priv[CRYPTO_BYTES];
23
24 while (true) {
25     sleep_ms(10000);
26
27     //Generación de claves
28     if (crypto_kem_keypair(pk, sk) != 0){
29         printf("Generacion de claves fallida\n\r");
30     } else {
31         printf("Generacion de claves exitosa (%d bytes de clave publica y %d bytes de clave privada)\n\r", CRYPTO_PUBLICKEYBYTES, CRYPTO_SECRETKEYBYTES);
32     }
33
34     //Generación de texto cifrado y secreto compartido
35     if (crypto_kem_enc(ct, ss_pub, pk) != 0){
36         printf("Generacion de texto cifrado y secreto compartido fallida\n\r");
37     } else {
38         printf("Generacion de texto cifrado y secreto compartido exitosa (%d bytes de texto cifrado y %d bytes de secreto compartido)\n\r", CRYPTO_CIPHTEXTBYTES, CRYPTO_BYTES);
39     }
40
41     //Generación de secreto compartido
42     if (crypto_kem_dec(ss_priv, ct, sk) != 0) {
43         printf("Generacion de secreto compartido fallida\n\r");
44     } else {
45         printf("Generacion de secreto compartido exitosa (%d bytes de secreto compartido)\n\r", CRYPTO_BYTES);
46     }
47
48     //Comprobación de resultados
49     if (memcmp(ss_pub, ss_priv, (unsigned int) CRYPTO_BYTES) ) {
50         printf("Comprobación fallida\n\r");
51     } else{
52         printf("¡Comprobación superada!\n\r");
53     }
54 }
55 }

```

El archivo `main.c` requiere del archivo `main.h`, cuyo contenido se muestra en Código 3.23. En este código, se definen macros necesarias para el funcionamiento del algoritmo. Dichas macros se han debido definir debido a que no se definen en ningún otro fichero relativo al algoritmo. Estas macros son las receptoras del parámetro `-DPQCLEAN_NAMESPACE` explicado en el apartado 3.3.2.1. También, se elimina la definición de la macro `__linux__` y se define `RP2040`, cuyo motivo se explica en la sección 3.3.2.2

Código 3.23: Archivo HQC-128/main.h.

```

1  #ifndef MAIN_H
2  #define MAIN_H
3
4  #include <stddef.h>
5
6  #define PASTER(x, y) x##_##y
7  #define EVALUATOR(x, y) PASTER(x, y)
8  #define NAMESPACE(fun) EVALUATOR(PQCLEAN_NAMESPACE, fun)
9
10 #define CRYPTO_BYTES NAMESPACE(CRYPTO_BYTES)
11 #define CRYPTO_PUBLICKEYBYTES NAMESPACE(CRYPTO_PUBLICKEYBYTES)
12 #define CRYPTO_SECRETKEYBYTES NAMESPACE(CRYPTO_SECRETKEYBYTES)
13 #define CRYPTO_CIPHTEXTBYTES NAMESPACE(CRYPTO_CIPHTEXTBYTES)
14 #define CRYPTO_ALGNAME NAMESPACE(CRYPTO_ALGNAME)
15

```

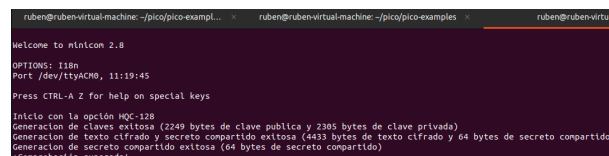
```

16 #define crypto_kem_keypair NAMESPACE(crypto_kem_keypair)
17 #define crypto_kem_enc     NAMESPACE(crypto_kem_enc)
18 #define crypto_kem_dec     NAMESPACE(crypto_kem_dec)
19
20 //Necesario para generación de números aleatorios
21 #ifdef __linux__
22     #undef __linux__
23 #endif
24 #ifndef RP2040
25     #define RP2040
26 #endif
27 #endif

```

3.3.2.4 Comprobación del algoritmo

Una vez realizados los pasos descritos anteriormente, se ejecuta la comprobación del algoritmo ante el cual se obtiene la Captura 3.7.



```

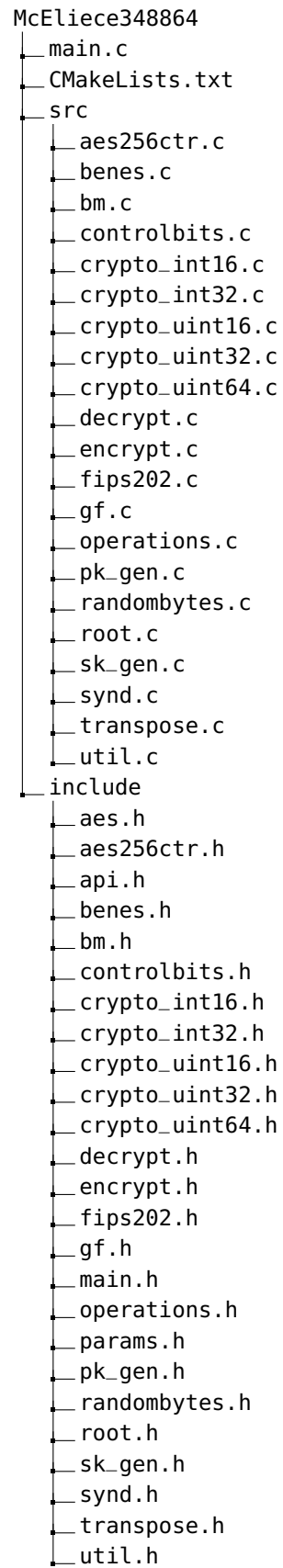
ruben@ruben-virtual-machine: ~/pico/pico-examples
Welcome to nmincon 2.8
OPTIONS: 128n
Port /dev/ttyACM0, 11:19:45
Press CTRL-A Z for help on special keys
Inicio con la opción HQC-128
Generacion de claves exitosa (2249 bytes de clave publica y 2305 bytes de clave privada)
Generacion de texto cifrado y secreto compartido exitosa (4433 bytes de texto cifrado y 64 bytes de secreto compartido)
Generacion de secreto compartido exitosa (64 bytes de secreto compartido)
¡Comprobación superada!

```

Figura 3.7: Comprobación de ejecución de algoritmo HQC-128.

3.3.3 McEliece348864

Para este algoritmo, la implementación también se ha obtenido del repositorio del proyecto PQ-Clean [36]. En este caso, se ha contruido el árbol de ficheros mostrado en la Figura 3.8. Al igual que en los casos anteriores, se incluye un directorio `src` con los ficheros de código fuente y un directorio `include` en el que se incluyen todas las librerías del algoritmo. Además, se incluye un archivo `CMakeLists.txt` y el fichero de comprobación `main.c`.



```
graph TD
    McEliece348864 --> main_c[main.c]
    McEliece348864 --> CMakeLists_txt[CMakeLists.txt]
    McEliece348864 --> src
    McEliece348864 --> include
    src --> aes256ctr_c[aes256ctr.c]
    src --> benes_c[benes.c]
    src --> bm_c[bm.c]
    src --> controlbits_c[controlbits.c]
    src --> crypto_int16_c[crypto_int16.c]
    src --> crypto_int32_c[crypto_int32.c]
    src --> crypto_uint16_c[crypto_uint16.c]
    src --> crypto_uint32_c[crypto_uint32.c]
    src --> crypto_uint64_c[crypto_uint64.c]
    src --> decrypt_c[decrypt.c]
    src --> encrypt_c[encrypt.c]
    src --> fips202_c[fips202.c]
    src --> gf_c[gf.c]
    src --> operations_c[operations.c]
    src --> pk_gen_c[pk_gen.c]
    src --> randombytes_c[randombytes.c]
    src --> root_c[root.c]
    src --> sk_gen_c[sk_gen.c]
    src --> synd_c[synd.c]
    src --> transpose_c[transpose.c]
    src --> util_c[util.c]
    include --> aes_h[aes.h]
    include --> aes256ctr_h[aes256ctr.h]
    include --> api_h[api.h]
    include --> benes_h[benes.h]
    include --> bm_h[bm.h]
    include --> controlbits_h[controlbits.h]
    include --> crypto_int16_h[crypto_int16.h]
    include --> crypto_int32_h[crypto_int32.h]
    include --> crypto_uint16_h[crypto_uint16.h]
    include --> crypto_uint32_h[crypto_uint32.h]
    include --> crypto_uint64_h[crypto_uint64.h]
    include --> decrypt_h[decrypt.h]
    include --> encrypt_h[encrypt.h]
    include --> fips202_h[fips202.h]
    include --> gf_h[gf.h]
    include --> main_h[main.h]
    include --> operations_h[operations.h]
    include --> params_h[params.h]
    include --> pk_gen_h[pk_gen.h]
    include --> randombytes_h[randombytes.h]
    include --> root_h[root.h]
    include --> sk_gen_h[sk_gen.h]
    include --> synd_h[synd.h]
    include --> transpose_h[transpose.h]
    include --> util_h[util.h]
```

McEliece348864

- main.c
- CMakeLists.txt
- src
 - aes256ctr.c
 - benes.c
 - bm.c
 - controlbits.c
 - crypto_int16.c
 - crypto_int32.c
 - crypto_uint16.c
 - crypto_uint32.c
 - crypto_uint64.c
 - decrypt.c
 - encrypt.c
 - fips202.c
 - gf.c
 - operations.c
 - pk_gen.c
 - randombytes.c
 - root.c
 - sk_gen.c
 - synd.c
 - transpose.c
 - util.c
- include
 - aes.h
 - aes256ctr.h
 - api.h
 - benes.h
 - bm.h
 - controlbits.h
 - crypto_int16.h
 - crypto_int32.h
 - crypto_uint16.h
 - crypto_uint32.h
 - crypto_uint64.h
 - decrypt.h
 - encrypt.h
 - fips202.h
 - gf.h
 - main.h
 - operations.h
 - params.h
 - pk_gen.h
 - randombytes.h
 - root.h
 - sk_gen.h
 - synd.h
 - transpose.h
 - util.h

Figura 3.8: Árbol de ficheros del proyecto MCEliece348864.

3.3.3.1 Generación de números aleatorios

Para adaptar la generación de números aleatorios en este algoritmo, se ha modificado el archivo McEliece348864/src/rndombytes.c de la forma mostrada en el Código 3.29. La función randombytes se utiliza para llamar a una función específica de generación de números aleatorios de acuerdo al sistema en el que se ejecuta, como por ejemplo Windows o un sistema basado en Linux. Por ello, se debe crear una opción para el caso de RP2040, que se ha llamado randombytes_rp2040_randombytes. El contenido de la función randombytes_rp2040_randombytes es idéntico a la función de generación de números aleatorios en el algoritmo HQC-128 3.3.2.2.

Código 3.24: Archivo McEliece348864/src/rndombytes.c.

```

1  int randombytes(uint8_t *output, size_t n) {
2      void *buf = (void *)output;
3      #if defined(__EMSCRIPTEN__)
4          ...
5      #elif defined(RP2040)
6          return randombytes_rp2040_randombytes(buf, n);
7      #else
8      # error "randombytes(...) is not supported on this platform"
9      #endif
10 }
11
12 #ifndef RP2040
13 static int randombytes_rp2040_randombytes(unsigned char *x, unsigned long long xlen){
14     // srand(time(NULL));
15     uint32_t aux = 0;
16     for (int i = 0; i < xlen; i+=4){
17         aux = get_rand_32();
18         memcpy(x+i, &aux, min(4, xlen-1));
19     }
20     return 0;
21 }
22 #endif

```

3.3.3.2 Fichero CMake

En cuanto al fichero CMakeLists.txt, este contiene el Código 3.25. En él, se especifica el valor de la macro PQCLEAN_NAMESPACE apropiado para la ejecución de este algoritmo junto al resto de opciones de compilación para mostrar los *warning* existentes. También, se incluyen todos los archivos fuente mostrados en el árbol 3.8. Además, se incluyen las dependencias comunes stdlib y pico_srand. Finalmente, se desactiva el uso de UART como salida y se activa el uso de USB.

Código 3.25: Archivo McEliece348864/CMakeLists.txt.

```

1  cmake_minimum_required(VERSION 3.13)
2
3  if (TARGET tinyusb_device)
4      add_compile_options(-Wall -Wextra -Wpedantic -Wredundant-decls -Wcast-align -Wmissing-prototypes -
5                          DPQCLEAN_NAMESPACE=PQCLEAN_MCELIECE348864_CLEAN)
6
7      add_executable(mceliece348864)
8
9      target_sources(mceliece348864 PUBLIC
10         ${CMAKE_CURRENT_LIST_DIR}/main.c
11         ${CMAKE_CURRENT_LIST_DIR}/src/aes256ctr.c
12         ${CMAKE_CURRENT_LIST_DIR}/src/benes.c
13         ${CMAKE_CURRENT_LIST_DIR}/src/bm.c
14         ${CMAKE_CURRENT_LIST_DIR}/src/controlbits.c
15         ${CMAKE_CURRENT_LIST_DIR}/src/crypto_int16.c
16         ${CMAKE_CURRENT_LIST_DIR}/src/crypto_int32.c
17         ${CMAKE_CURRENT_LIST_DIR}/src/crypto_uint16.c
18         ${CMAKE_CURRENT_LIST_DIR}/src/crypto_uint32.c
19         ${CMAKE_CURRENT_LIST_DIR}/src/crypto_uint64.c

```

```

19     ${CMAKE_CURRENT_LIST_DIR}/src/decrypt.c
20     ${CMAKE_CURRENT_LIST_DIR}/src/encrypt.c
21     ${CMAKE_CURRENT_LIST_DIR}/src/gf.c
22     ${CMAKE_CURRENT_LIST_DIR}/src/operations.c
23     ${CMAKE_CURRENT_LIST_DIR}/src/pk_gen.c
24     ${CMAKE_CURRENT_LIST_DIR}/src/root.c
25     ${CMAKE_CURRENT_LIST_DIR}/src/sk_gen.c
26     ${CMAKE_CURRENT_LIST_DIR}/src/synd.c
27     ${CMAKE_CURRENT_LIST_DIR}/src/transpose.c
28     ${CMAKE_CURRENT_LIST_DIR}/src/util.c
29     ${CMAKE_CURRENT_LIST_DIR}/src/randombytes.c
30     ${CMAKE_CURRENT_LIST_DIR}/src/fips202.c
31 )
32
33 target_include_directories(mceliece348864 PUBLIC
34     ${CMAKE_CURRENT_LIST_DIR}/include)
35
36 # pull in common dependencies
37 target_link_libraries(mceliece348864 pico_stdlib pico_rand)
38
39 # enable usb output, disable uart output
40 pico_enable_stdio_usb(mceliece348864 1)
41 pico_enable_stdio_uart(mceliece348864 0)
42
43 # create map/bin/hex/uf2 file etc.
44 pico_add_extra_outputs(mceliece348864)
45
46 # add url via pico_set_program_url
47 example_auto_set_url(mceliece348864)
48 elseif(PICO_ON_DEVICE)
49     message(WARNING "not building mceliece348864 because TinyUSB submodule is not initialized in the SDK"
50 )
51 endif()

```

3.3.3.3 Fichero de comprobación

Para la comprobación del algoritmo, se ha utilizado el mismo archivo descrito en el apartado 3.3.2.3 ya que ambos incluyen funciones con el mismo nombre. Únicamente se diferencian en el tamaño de las variables, lo cual viene indicado por la macro `PQCLEAN_NAMESPACE` especificada en el archivo `CMakeLists.txt`. Además, las variables de claves pública y privada hacen uso de memoria dinámica en vez de memoria estática.

3.3.3.4 Comprobación del algoritmo

Una vez completados estos pasos, se ha ejecutado el algoritmo utilizando el fichero descrito en el apartado 2.1.4.4. El resultado de esta comprobación ha sido el mostrado en la Figura 3.9.

Como se puede verificar en la Figura 3.9, la ejecución de este algoritmo arroja un error de memoria insuficiente. Este error se genera al reservar memoria para las variables de las claves pública y privada.

3.3.4 Sphincs

El último algoritmo comprobado en este dispositivo se trata de Sphincs. Para ello, se ha utilizado la implementación oficial obtenible en el repositorio de GitHub [39]. En este repositorio, se encuentran varias implementaciones, entre las cuales se pueden ver una implementación optimizada para la arquitectura Intel x86 y de referencia en lenguaje C. Para este trabajo, se ha utilizado la implementación de referencia incluida en el directorio `ref` del repositorio. Para poder utilizar estos archivos, se ha creado el árbol de ficheros mostrado en la Figura 3.10. Al igual que con el resto de proyectos, todos los archivos fuente del algoritmo se han incluido en un directorio `src` mientras que las librerías se han almacenado en el directorio `include`.

```

ruben@ruben-virtual-machine: ~/pico/pico-exampl... x

Welcome to minicom 2.8

OPTIONS: I18n
Port /dev/ttyACM0, 09:22:56

Press CTRL-A Z for help on special keys

Inicio con la opción Classic McEliece 348864

*** PANIC ***

Out of memory

```

Figura 3.9: Comprobación de ejecución de algoritmo McEliece348864.

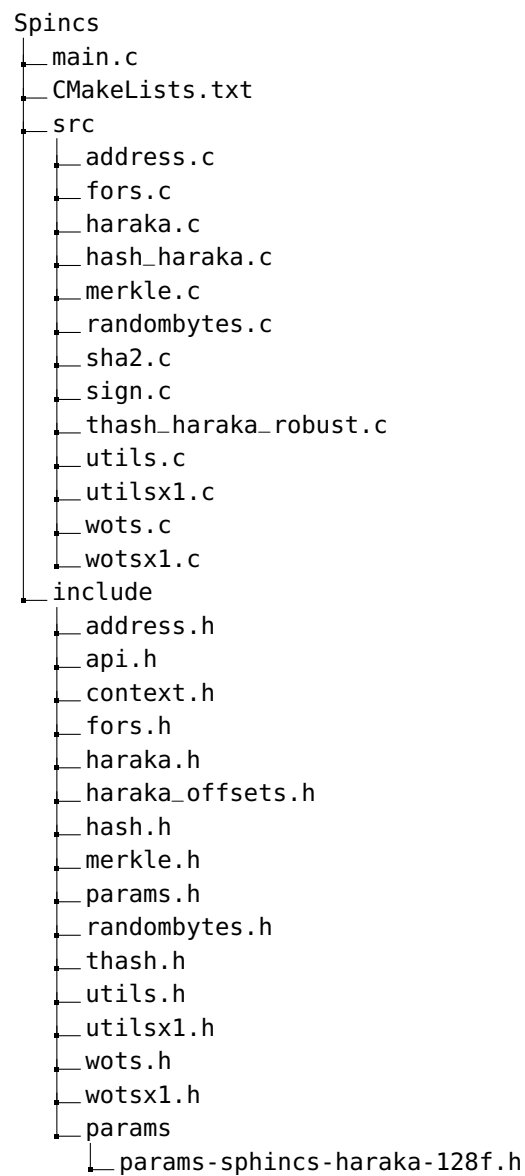


Figura 3.10: Árbol de ficheros del proyecto Sphincs.

3.3.4.1 Fichero CMake

En cuanto al fichero CMakeLists.txt, este contiene el Código 3.26. En él, se especifica el valor de la macro PARAMS apropiado para la ejecución de este algoritmo junto al resto de opciones de compilación para mostrar los *warning* existentes. También, se incluyen todos los archivos fuente mostrados en el árbol 3.10. Además, se incluyen las dependencias comunes `stdlib` y `pico_srand`. Por último, se activa el uso de USB y se desactiva el uso de UART como salida.

Código 3.26: Archivo Sphincs/CMakeLists.txt.

```

1 cmake_minimum_required(VERSION 3.13)
2
3 if (TARGET tinyusb_device)
4     add_compile_options(-Wall -Wextra -Wpedantic -Wconversion -Wmissing-prototypes -DPARAMS=sphincs-
5         haraka-128f)
6
7     add_executable(sphincs)
8
9     target_sources(sphincs PUBLIC
10         ${CMAKE_CURRENT_LIST_DIR}/main.c
11         ${CMAKE_CURRENT_LIST_DIR}/src/address.c
12         ${CMAKE_CURRENT_LIST_DIR}/src/randombytes.c
13         ${CMAKE_CURRENT_LIST_DIR}/src/merkle.c
14         ${CMAKE_CURRENT_LIST_DIR}/src/wots.c
15         ${CMAKE_CURRENT_LIST_DIR}/src/wotsx1.c
16         ${CMAKE_CURRENT_LIST_DIR}/src/utls.c
17         ${CMAKE_CURRENT_LIST_DIR}/src/utlsx1.c
18         ${CMAKE_CURRENT_LIST_DIR}/src/fors.c
19         ${CMAKE_CURRENT_LIST_DIR}/src/sign.c
20         ${CMAKE_CURRENT_LIST_DIR}/src/haraka.c
21         ${CMAKE_CURRENT_LIST_DIR}/src/hash_haraka.c
22         ${CMAKE_CURRENT_LIST_DIR}/src/thash_haraka_robust.c
23     )
24
25     target_include_directories(sphincs PUBLIC
26         ${CMAKE_CURRENT_LIST_DIR}/include
27         ${CMAKE_CURRENT_LIST_DIR}/include/params)
28
29     # pull in common dependencies
30     target_link_libraries(sphincs pico_stdlib pico_rand)
31
32     # enable usb output, disable uart output
33     pico_enable_stdio_usb(sphincs 1)
34     pico_enable_stdio_uart(sphincs 0)
35
36     # create map/bin/hex/uf2 file etc.
37     pico_add_extra_outputs(sphincs)
38
39     # add url via pico_set_program_url
40     example_auto_set_url(sphincs)
41 elseif(PICO_ON_DEVICE)
42     message(WARNING "not building sphincs because TinyUSB submodule is not initialized in the SDK")
43 endif()

```

En este proyecto, para modificar la versión del algoritmo utilizada, se debe modificar el archivo Sphincs/CMakeLists.txt. Más concretamente, se debe cambiar el parámetro PARAMS de forma que especifique la función *hash* utilizada (*haraka* o *sha2*), el número de bits a emplear (128, 192 o 256) y las opciones (s o f). También, se puede elegir entre el archivo `/src/thash_`(función *hash*)_robust.c o `/src/thash_`(función *hash*)_simple.c. En caso de seleccionar una función *hash* distinta a *haraka*, se deben alterar las inclusiones `/src/haraka.c`, `/src/haraka.c` y `src/hash_haraka.c`. Para utilizar la función *sha2*, se debe sustituir “*haraka*” por “*sha2*” en el nombre de dichos archivos. Para el caso de la función *shake*, se debe sustituir “*haraka*” por “*shake*” en el nombre de dichos archivos excepto `/src/haraka.c`, el cual se debe cambiar por `/src/fips202.c`.

3.3.4.2 Generación de números aleatorios

Para la generación de números aleatorios, se ha utilizado el archivo `randombytes.c`, sustituyendo al inicialmente empleado `rng.c` ya que este se basaba en realizar operaciones a nivel de bit. Por ello, se ha modificado la función que incluye el fichero `randombytes.c` para realizar la generación de números aleatorios mediante la función específica del dispositivo, ya que, originalmente, el archivo llevaba a cabo la generación mediante la apertura del fichero `/dev/urandom`, existente en los sistemas UNIX. El contenido final de este fichero es el mostrado en el Código 3.27. Como se puede comprobar, la función `randombytes` es idéntica a las utilizadas en los algoritmos anteriores.

Código 3.27: Archivo `Sphincs/src/randombytes.c`.

```

1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include "pico/rand.h"
5  #include "randombytes.h"
6
7  void randombytes(unsigned char *x, unsigned long long xlen){
8      uint32_t aux = 0;
9      for (int i = 0; i < xlen; i+=4){
10         aux = get_rand_32();
11         memcpy(x+i, &aux, min(4, xlen-1));
12     }
13 }
```

3.3.4.3 Fichero de comprobación

Para poder comprobar el algoritmo Sphincs, se ha diseñado el archivo `main.c` con el contenido mostrado en el código 3.28. En este fichero se ha seguido la nomenclatura seguida hasta, por lo que las claves pública (pk) y privada (sk) mediante la función `crypto_sign_keypair`. A continuación, se utiliza la clave pública para firmar un mensaje (m con longitud mlen) utilizando la función `crypto_sign`, la cual nos devuelve el mensaje firmado sm con longitud smlen. Por último, se obtiene el mensaje firmado (m1 con longitud mlen1) con la función `crypto_sign_open` y se verifica si es idéntico al firmado originalmente.

Código 3.28: Archivo `Sphincs/main.c`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "pico/stdlib.h"
5  #include "api.h"
6
7  int main() {
8      stdio_init_all();
9      printf("Inicio con la opción %s\n\r", xstr(PARAMS));
10
11     //Generación de claves
12     unsigned char pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
13
14     //Firma de mensaje
15     unsigned char m[] = "Esto es una prueba de la firma de mensajes utilizando Sphincs.";
16     unsigned char *sm;
17     unsigned long long mlen = sizeof(m), smlen;
18
19     //Comprobación de la firma
20     unsigned char *m1;
21     unsigned long long mlen1;
22
23     while (true) {
24         //Generación de claves
25         printf("Inicio\n");
```

```

26     if (crypto_sign_keypair(pk, sk) != 0){
27         printf("Generacion de claves fallida\n\r");
28     } else {
29         printf("Generacion de claves exitosa (%d bytes de clave publica y %d bytes de clave privada)\n\r", CRYPTO_PUBLICKEYBYTES, CRYPTO_SECRETKEYBYTES);
30     }
31
32     //Firma de mensaje
33     m1 = (unsigned char *)calloc(mlen+CRYPTO_BYTES, sizeof(unsigned char));
34     sm = (unsigned char *)calloc(mlen+CRYPTO_BYTES, sizeof(unsigned char));
35     if (crypto_sign(sm, &smlen, m, mlen, sk) != 0){
36         printf("Firma de mensaje fallida\n\r");
37     } else {
38         printf("Firma del mensaje exitosa (%llu bytes de firma)\n\r", smlen);
39     }
40
41     //Comprobación de la firma
42     if (crypto_sign_open(m1, &m1len, sm, smlen, pk) != 0) {
43         printf("Comprobacion de la firma fallida\n\r");
44     } else {
45         if (mlen != m1len) {
46             printf("Longitud del mensaje original distinta a la longitud del mensaje recuperado\n\r");
47             ;
48         } else if (memcmp(m, m1, mlen)) {
49             printf("Contenido del mensaje original distinto al contenido del mensaje recuperado\n\r");
50             ;
51         } else {
52             printf("Comprobacion de la firma del mensaje exitosa (%llu bytes de mensaje)\n\r", m1len);
53             ;
54         }
55     }
56
57     //Liberación de la memoria reservada
58     free(m1);
59     free(sm);
60     sleep_ms(10000);
61 }

```

3.3.4.4 Comprobación del algoritmo

Finalmente, se ha llevado a cabo la ejecución del fichero de comprobación y se ha obtenido el resultado mostrado en la Captura 3.11.

```

ruben@ruben-virtual-machine: ~/pico/pico-examples
Welcome to minicon 2.8
OPTIONS: 118h
Port: /dev/ttyACM0, 11:28:55
Press CTRL-A Z for help on special keys
Inicio con la opción sphincs-haraka-128s
Generacion de claves exitosa (32 bytes de clave publica y 64 bytes de clave privada)
Firma del mensaje exitosa (7919 bytes de firma)
Comprobacion de la firma del mensaje exitosa (63 bytes de mensaje)
Comprobacion de la firma del mensaje = "Esto es una prueba de la firma de mensajes utilizando Sphincs."

```

Figura 3.11: Comprobación de ejecución de algoritmo Sphincs.

3.4 STM32

El último dispositivo a utilizar se trata del dispositivo STM32, específicamente de la plataforma Nucleo-L4R5ZI.

Para poder programar esta plataforma, se disponen de dos posibilidades principales: Keil μ Vision5 [40] y STM32Cube [41]. STM32Cube se trata de una herramienta oficial desarrollada por el grupo ST. Esta herramienta incluye los módulos:

- **STM32CubeMX** [42]: Es un módulo gráfico utilizado para cualquier dispositivo STM32 el cual genera el código de inicialización en lenguaje C.
- **STM32CubeIDE** [43]: Este Integrated Development Environment (IDE) se basa en soluciones como Eclipse y aporta un compilador y depurador.
- **STM32CubeMonitor** [44]: Este módulo provee la posibilidad de comprobar datos de la aplicación en tiempo real.
- **STM32CubeProgrammer** [45]: Este módulo proporciona una manera de leer y escribir en la memoria del dispositivo en tiempo real.

Como inconveniente, esta herramienta requiere un tiempo excesivo para su dominio ya que entraña gran complejidad. Por el contrario, Keil μ Vision5 supone una menor dificultad por lo que será la herramienta utilizada.

3.4.1 Keil uVision5

Este *software* hace uso de librerías, módulos y ficheros de configuración específicos de los dispositivos utilizados. Para ello, se debe seleccionar el dispositivo utilizado en el menú ofrecido por la herramienta.

Para poder cargar *software* en el dispositivo, es necesario llevar a cabo la especificación del rango de memoria disponible para la carga del programa. Esto se realiza mediante la ventana a Cortex-M Target Driver Setup, a la cual se puede acceder desde la configuración del proyecto y, dentro de ella, el apartado “Settings” dentro de la ventana “Utilities”. Dentro de esta ventana, se debe seleccionar la opción “Add” y añadir la opción con nombre “STM32L4Rx 2MB Dual Bank Flash”, con dirección de inicio 0x800_0000 y tamaño 0x20_0000. También, se debe indicar la RAM disponible para el algoritmo. Esta comienza en la dirección de memoria 0x2000_0000 y con un tamaño 0xA_0000.

3.4.2 McEliece348864

Para la ejecución del algoritmo McEliece348864, en primer lugar, se requiere la selección del código fuente. Dicho código fuente se obtiene del repositorio perteneciente al proyecto PQClean [36].

Una vez se disponga del código, este se incluye en el proyecto de ejemplo que se menciona en la Subsección 3.4.2.1.

3.4.2.1 Generación de números aleatorios

Para la generación de números aleatorios, se ha utilizado el ejemplo propuesto en el repositorio STM32CubeL4 [46]. Este ejemplo se encuentra en la ruta STM32CubeL4/Projects/NUCLEO-L4R5ZI/Examples/RNG/RNG_MultiRNG. En este ejemplo, se utiliza la función HAL_RNG_GenerateRandomNumber, la cual obtiene números aleatorios de 32 bits. Por ello, se ha diseñado la función randombytes_stm32_randombytes para su uso en el algoritmo. Esta función se muestra en el Código 3.29.

Código 3.29: Función de generación de números aleatorios para McEliece348864.

```

1  int randombytes_stm32_randombytes(unsigned char *x, unsigned long long xlen){
2      // srand(time(NULL));
3      uint32_t aux = 0;
4      for (int i = 0; i < (int) xlen; i+=4){
5          HAL_RNG_GenerateRandomNumber(&RngHandle, &aux);
6          memcpy(x+i, &aux, min(4, xlen-1));
7      }
8      return 0;
9  }
```

3.4.2.2 Archivo startup_stm32l4r5xx.s

A continuación, se debe llevar a cabo la modificación del archivo `startup_stm32l4r5xx.s`. En este archivo se indica el puntero a la pila y el manejador de la interrupción de reinicio entre otras opciones al iniciar el dispositivo. Además, se indica el tamaño de la memoria *stack*, que inicialmente es 0x400. Este valor provoca que la ejecución del programa arroje un error a la hora de llamar a la función `pk_gen` en el archivo `pk_gen.c`. Los errores arrojados se muestran en la Figura 3.12. Ambos errores se refieren a un uso erróneo de la memoria *stack*.

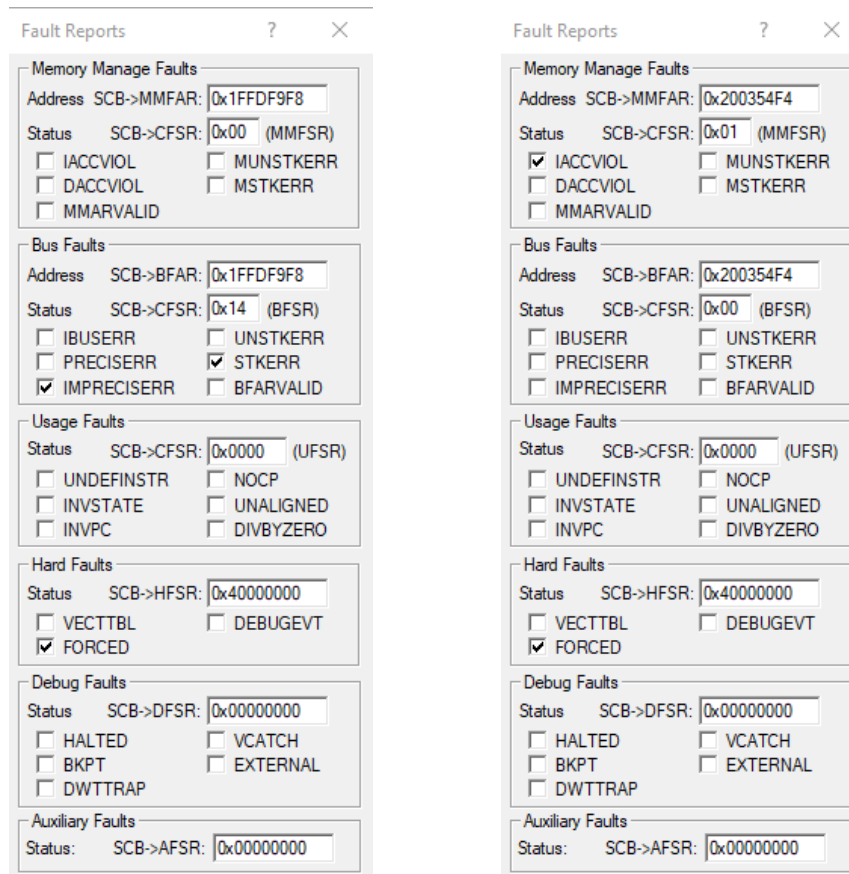


Figura 3.12: Errores de manejo de memoria *stack* al ejecutar McEliece348864.

Dicho error se debe a que las variables utilizadas en dicha función no caben en la memoria *stack*. Las definiciones de las variables más grandes se muestran en el Código 3.30.

Código 3.30: Variables en la función `pk_gen`.

```

1  #define GFBITS 12
2  #define SYS_N 3488
3  #define SYS_T 64
4  #define PK_NROWS (SYS_T*GFBITS)
5
6  typedef uint16_t gf;
7
8  ...
9
10 uint64_t buf[ 1 << GFBITS ];
11
12 unsigned char mat[ PK_NROWS ][ SYS_N / 8 ];
13
14 gf g[ SYS_T + 1 ]; // Goppa polynomial
15 gf L[ SYS_N ]; // support
16 gf inv[ SYS_N ];

```


La variable `buf` utiliza 32.768 bytes (0x800), lo cual ya es superior al tamaño de la memoria *stack* asignada inicialmente. La variable `mat` hace uso de 334.848 bytes (0x5_1C00). La variable `g` emplea 130 bytes (0x82) mientras que las variables `L` y `inv` requieren 6.976 bytes (0x1B40) cada una. En total, el tamaño requerido de memoria *stack* en esta función es de 381.650 bytes (0x5D2D2). Por ello, se debe asignar un tamaño mínimo a la memoria *stack* suficiente para llevar a cabo la ejecución del *software* pero no tanta como para que no sea posible almacenar el código ejecutable, lo cual sucede al asignar 0x6_0000 bytes a la memoria *stack*. La cantidad de memoria utilizada finalmente ha sido 0x5D800.

3.4.2.3 Función de encapsulado y desencapsulado

Como se ha indicado en la Subsección 3.4.2.2, para la ejecución de la función `pk_gen` se ha de aumentar el tamaño de la memoria *stack* casi hasta el máximo posible. Esto cubre únicamente la generación de claves del algoritmo. Al incluir las funciones de encapsulado y desencapsulado se obtiene el mismo error que se recibía inicialmente para la función de generación de claves. Esto hace que no sea viable la ejecución de la función de encapsulado y desencapsulado del algoritmo ya que no son manejables debido al tamaño disponible de memoria.

3.4.3 PQM4

Para la ejecución de otros algoritmos, se ha utilizado el repositorio existente PQM4 [47]. Este repositorio contiene implementaciones de distintos algoritmos postcuánticos diseñadas para su uso en dispositivos que empleen microcontroladores pertenecientes a la familia ARM Cortex-M4. Dentro de este repositorio, se pueden encontrar *scripts* en Python 3 que llevan a cabo diferentes medidas de la ejecución de los distintos algoritmos en el dispositivo. Estas medidas son el tiempo necesario para su ejecución, la cantidad de memoria *stack* utilizada y el tamaño del código. Algunas de las plataformas soportadas son Nucleo-L4R5ZI, Nucleo-L476RG, CW308T-STM32F3 y MPS2-AN386.

Para poder utilizar este *software*, se debe instalar el paquete `arm-none-eabi-gcc` y `OpenOCD`. Además, se debe utilizar una versión de Python igual o superior a la versión 3.8.

Para la obtención de las medidas de, por ejemplo, el algoritmo `Bike1`, se ha de ejecutar el comando mostrado en el Código 3.31.

Código 3.31: Comando para la obtención de medidas utilizando PQM4.

```
1 python3 benchmarks.py bike1 --platform nucleo-l4r5zi --uart /dev/ttyACM1
```

Este comando arrojará las medidas obtenidas en diferentes archivos (según medida obtenida y algoritmo comprobado) dentro del directorio llamado `benchmarks`.

Para este proyecto, se ha llevado a cabo las medidas de los algoritmos `Falcon` (versiones 512 y 1024), `Dilithium` (versiones 2, 3 y 5), `Kyber` (versiones 2, 3 y 4), `HQC` (versiones 128, 192 y 256) y `Bike1` (versiones 1 y 3).

Capítulo 4

Evaluación y resultados

Una vez realizados todos los pasos indicados en el Capítulo 3, se debe llevar a cabo la comprobación y medición de rendimiento de los algoritmos. En este capítulo, se explicará la realización de pruebas de comprobación de funcionamiento de los algoritmos y, posteriormente, las mediciones llevadas a cabo para conocer el rendimiento de los mismos.

4.1 Pruebas unitarias de comprobación

4.2 Medidas de ejecución

Para poder medir el rendimiento de los algoritmos, se han tenido en cuenta varios factores. Estos factores son el tamaño del *software* resultante, la energía consumida durante la ejecución del algoritmo y el tiempo de ejecución del algoritmo.

4.2.1 Consumo energético

Para realizar la medición del consumo energético, se ha empleado el dispositivo INA219 [48]. Este dispositivo se empleará junto a una tarjeta ESP32, de forma que el dispositivo INA219 realiza las medidas de energía consumida y el dispositivo ESP32 envía estos datos al ordenador conectado.

4.2.2 Tiempo de ejecución

En cuanto al tiempo de ejecución se refiere, para poder conseguir una medida significativa, se ha realizado la ejecución de cada función que componen los algoritmos. Para ello, se obtendrá una medida del tiempo transcurrido al inicio de la función y otra al finalizar esta, de forma que la resta de ambas suponga el tiempo requerido para la ejecución de esta función.

4.2.2.1 Temporización en dispositivo ESP32

Para poder medir el tiempo transcurrido durante la ejecución de cada función del algoritmo es necesaria la inclusión de la librería `esp_timer.h` y la utilización de la función `esp_timer_get_time` [49]. Se ha hecho uso de esta función ya que aporta una precisión del orden de microsegundos a la hora de llevar a cabo la medición de tiempo. La medición se realiza mediante el Código 4.1. En este código, la variable `time_ini` representa el tiempo transcurrido al inicio de la función, `time_end` indica el tiempo transcurrido hasta el final de la función y `time_ref` se utiliza para obtener el tiempo transcurrido en la llamada a la obtención del tiempo transcurrido. Con esto, el tiempo necesario para la ejecución de la función (en el Código 4.1 la función medida es la generación

de claves) se obtendría mediante la resta del tiempo al finalizar la ejecución de la función menos el tiempo de inicio y el tiempo requerido para la obtención del tiempo transcurrido. Es decir, $tiempo_{funcion} = tiempo_{final} - tiempo_{inicio} - (tiempo_{referencia} - tiempo_{final})$.

Código 4.1: Medición temporal en el dispositivo ESP32.

```
1 time_ini = esp_timer_get_time();
2 crypto_sign_keypair(pk, sk);
3 time_end = esp_timer_get_time();
4 time_ref = esp_timer_get_time();
```

4.2.2.2 Temporización en dispositivo RP2040

Para llevar a cabo la medición del tiempo transcurrido en el dispositivo RP2040 es necesaria la inclusión de la librería `pico/time.h` y la utilización de la función `get_absolute_time` [50]. Se ha escogido esta función ya que otorga una precisión de microsegundos. La medición del tiempo transcurrido se ha realizado, mediante el Código 4.2. Para este caso, las variables utilizadas implican el mismo funcionamiento que en el caso del dispositivo ESP32, mostrado en la Subsección 4.2.2.1. Por ello, la forma de obtener el tiempo necesario para la ejecución de la función es, también, idéntica al caso del dispositivo ESP32.

Código 4.2: Medición temporal en el dispositivo RP2040.

```
1 time_ini = get_absolute_time();
2 crypto_kem_keypair(pk, sk);
3 time_end = get_absolute_time();
4 time_ref = get_absolute_time();
```

4.2.2.3 Temporización en dispositivo STM32

En cuanto a la medición del tiempo transcurrido en el dispositivo STM32 es necesario llevar a cabo la diferenciación entre McEliece348864 y los algoritmos incluidos en el repositorio PQM4. Para el primero, debido a la gran cantidad de memoria requerida para el algoritmo y su ejecución, no es viable la posibilidad de incluir algún módulo de comunicación entre un ordenador y el dispositivo. Por ello, el único método utilizable consiste en el iluminado y apagado de los LEDs que incluye el dispositivo a modo de indicador de inicio y final de ejecución.

En el caso de los algoritmos incluidos en el repositorio PQM4, la ejecución de estos nos aporta el número de ciclos requeridos para su ejecución.

Para todos aquellos algoritmos que requieran de un mensaje para firmar, se utilizará el mensaje *Esto es una prueba de la firma de mensajes utilizando (nombre del algoritmo)*.

4.2.3 Pruebas en dispositivo ESP32

Para la medición de la temporización de los algoritmos Dilithium y Kyber, se ha llevado a cabo la ejecución de las 3 funciones características del algoritmo un total de 1000 iteraciones. Esta prueba requiere de la eliminación del módulo `watchdog` del sistema operativo utilizado por el dispositivo ESP32. Una vez se han realizado todas las ejecuciones, se ha obtenido el tiempo medio de ejecución y se ha representado en los gráficos mostrados en Figura 4.1 y la Figura 4.2.

Respecto a la ejecución del algoritmo Dilithium, se observa que el tiempo requerido para la verificación de la firma es ligeramente superior al requerido para la generación de las claves, mientras que el tiempo requerido para la firma de un mensaje es entre X y X veces superior al requerido para la generación de las claves. Es importante mencionar que, en el caso de la firma de un mensaje, el tiempo puede variar notablemente entre ejecuciones. Por ejemplo, en la versión Dilithium2, el tiempo mínimo observado ha sido

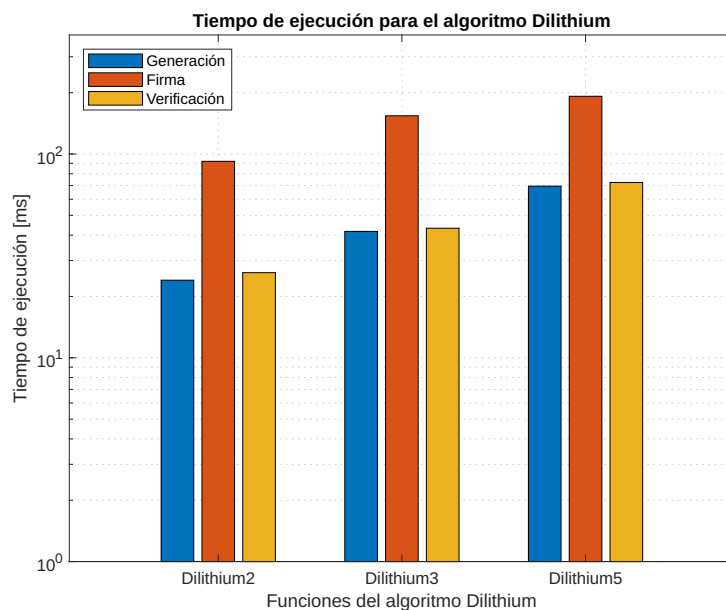


Figura 4.1: Tiempo de ejecución medio para el algoritmo Dilithium.

En el caso del algoritmo Kyber, los tiempos requeridos para las 3 funciones son más cercanos entre sí, como se puede comprobar en la Figura 4.2.

4.2.4 Pruebas en dispositivo RP2040

4.2.5 Pruebas en dispositivo STM32

McEliece claves tarda 1h y 20 min

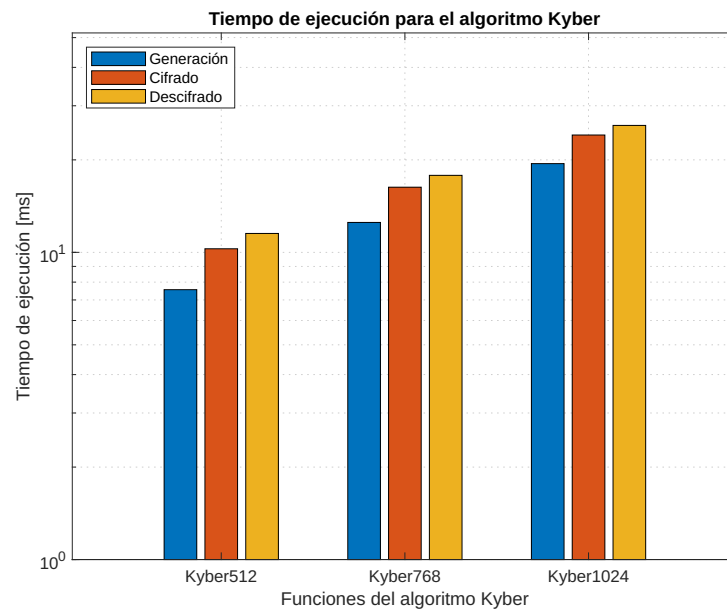


Figura 4.2: Tiempo de ejecución medio para el algoritmo Kyber.

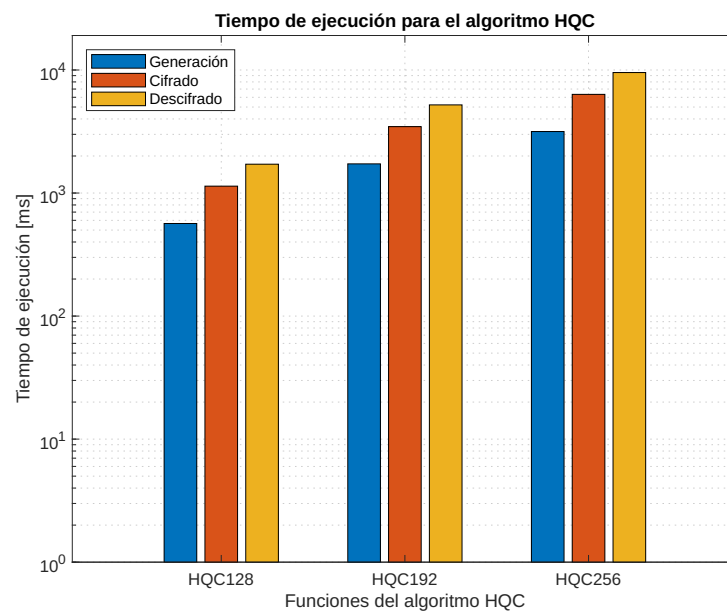


Figura 4.3: Tiempo de ejecución medio para el algoritmo HQC.

Capítulo 5

Conclusiones y líneas futuras

En este capítulo final se recopilan las conclusiones del proyecto basadas en los objetivos especificados en el capítulo 1 y en los resultados mostrados en el capítulo 4. Además, se exponen posibles continuaciones a este trabajo utilizando el desarrollo de este proyecto como base.

5.1 Conclusiones

5.2 Líneas futuras

Bibliografía

- [1] R. Rietsche, C. Dremel, S. Bosch, L. Steinacker, M. Meckel y J.-M. Leimeister, "Quantum computing", *Electronic Markets*, vol. 32, n.º 4, págs. 2525-2536, dic. de 2022, ISSN: 1422-8890. DOI: [10.1007/s12525-022-00570-y](https://doi.org/10.1007/s12525-022-00570-y). dirección: <https://doi.org/10.1007/s12525-022-00570-y>.
- [2] O. Ezratty, *Is there a Moore's law for quantum computing?*, mar. de 2023.
- [3] P. Schwabe, *CRYSTALS*, 2017. dirección: <https://pq-crystals.org/dilithium/> (visitado 21-06-2024).
- [4] *Selected Algorithms 2022*, 2017. dirección: <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022> (visitado 07-03-2024).
- [5] V. Lyubashevsky, "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures", en *Advances in Cryptology – ASIACRYPT 2009*, M. Matsui, ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, págs. 598-616, ISBN: 978-3-642-10366-7.
- [6] *CRYSTALS-Dilithium*, 2021. dirección: <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf> (visitado 07-03-2024).
- [7] *CRYSTALS-Kyber*, 2021. dirección: <https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf> (visitado 07-03-2024).
- [8] L. Lamport, "Constructing Digital Signatures from a One Way Function", inf. téc. CSL-98, oct. de 1979, This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. dirección: <https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/>.
- [9] *Sphincs+*, 2022. dirección: <https://sphincs.org/data/sphincs+-r3.1-specification.pdf> (visitado 07-03-2024).
- [10] *Classic McEliece: conservative code-based cryptography*, 2020. dirección: <https://classic.mceliece.org/nist/mceliece-20201010.pdf> (visitado 07-03-2024).
- [11] *Hamming Quasi-Cyclic (HQC)*, 2024. dirección: https://pqc-hqc.org/doc/hqc-specification_2024-02-23.pdf (visitado 07-03-2024).
- [12] *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU*, 2020. dirección: <https://falcon-sign.info/falcon.pdf> (visitado 07-03-2024).
- [13] *ESP32WROOM32E ESP32WROOM32UE Datasheet*, 2023. dirección: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e-esp32-wroom-32ue_datasheet_en.pdf (visitado 07-03-2024).
- [14] *PLACA ESP32 DEVKITC V4 WIFI+BT 4MB IOT*, 2015. dirección: <https://www.tiendatec.es/electronica/placas-de-desarrollo/1001-placa-esp32-devkit-wifi-bt-4mb-iot-8472496015325.html> (visitado 07-03-2024).
- [15] *RP2040 Datasheet*, 2020. dirección: <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf> (visitado 07-03-2024).
- [16] *RP2040*, 2012. dirección: <https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html> (visitado 07-03-2024).
- [17] *STM32L4R5xx STM32L4R7xx STM32L4R9xx*, 2020. dirección: <https://www.st.com/resource/en/datasheet/stm32l4r5vi.pdf> (visitado 07-03-2024).

- [18] *ST Nucleo L4R5ZI*, 2015. dirección: https://docs.zephyrproject.org/latest/boards/st/nucleo_l4r5zi/doc/index.html (visitado 07-03-2024).
- [19] *Getting Started with Arduino IDE 2*, 2015. dirección: <https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/> (visitado 07-03-2024).
- [20] *Get Started*, 2016. dirección: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html#introduction> (visitado 07-03-2024).
- [21] *PlatformIO Core (CLI)*, 2014. dirección: <https://docs.platformio.org/en/latest/core/index.html> (visitado 07-03-2024).
- [22] *FreeRTOS (IDF)*, 2015. dirección: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/freertos_idf.html (visitado 07-03-2024).
- [23] *Standard Toolchain Setup for Linux and macOS*, 2015. dirección: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/linux-macos-setup.html> (visitado 07-03-2024).
- [24] *ESP-Prog*, 2016. dirección: https://docs.espressif.com/projects/esp-dev-kits/en/latest/other/esp-prog/user_guide.html#start-application-development (visitado 07-03-2024).
- [25] *Espressif ESP-Prog ESP32 JTAG Debug Probe – Pinout Diagram*, 2024. dirección: <https://www.circuitstate.com/pinouts/espressif-esp-prog-esp32-jtag-debug-probe-pinout-diagram/> (visitado 07-03-2024).
- [26] *Please help me connecting Esp-prog to ESP32-DevkitC-v4*, 2019. dirección: <https://esp32.com/viewtopic.php?t=9045> (visitado 07-03-2024).
- [27] *Dilithium*, 2022. dirección: <https://github.com/pq-crystals/dilithium/tree/master> (visitado 07-03-2024).
- [28] *Random Number Generation*, 2016. dirección: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/random.html> (visitado 07-03-2024).
- [29] *Guru Meditation Error: Core 0 panic'ed (LoadProhibited). Exception was unhandled*. 2017. dirección: <https://www.esp32.com/viewtopic.php?t=19135> (visitado 07-03-2024).
- [30] *Troubleshooting Double Exception*, 2022. dirección: <https://www.esp32.com/viewtopic.php?t=27538> (visitado 07-03-2024).
- [31] *Kyber*, 2024. dirección: <https://github.com/pq-crystals/kyber> (visitado 07-03-2024).
- [32] *LilyGO T-Display RP2040*, 2023. dirección: <https://github.com/Xinyuan-LilyGO/LILYGO-T-display-RP2040/tree/main> (visitado 07-03-2024).
- [33] *Raspberry Pi Documentation*, 2023. dirección: <https://www.raspberrypi.com/documentation/pico-sdk/> (visitado 07-03-2024).
- [34] *Raspberry Pi Pico SDK*, 2023. dirección: <https://github.com/raspberrypi/pico-sdk> (visitado 07-03-2024).
- [35] M. J. Kannwischer, P. Schwabe, D. Stebila y T. Wiggers, "Improving Software Quality in Cryptography Standardization Projects", en *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, Los Alamitos, CA, USA: IEEE Computer Society, 2022, págs. 19-30. DOI: 10.1109/EuroSPW55150.2022.00010. dirección: <https://eprint.iacr.org/2022/337>.
- [36] *PQClean*, 2023. dirección: <https://github.com/PQClean/PQClean> (visitado 07-03-2024).
- [37] *add compile options*, 2023. dirección: https://cmake.org/cmake/help/latest/command/add_compile_options.html (visitado 07-03-2024).
- [38] *High Level APIs*, 2023. dirección: https://www.raspberrypi.com/documentation/pico-sdk/high_level.html#gac446d739bd6818ee25b5c8644ef7c8e8 (visitado 07-03-2024).
- [39] *SPHINCS+*, 2024. dirección: <https://github.com/sphincs/sphincsplus> (visitado 07-03-2024).

- [40] *μVision GUI*, 2024. dirección: <https://developer.arm.com/documentation/101407/0540/User-Interface/uVision-GUI> (visitado 07-03-2024).
- [41] *STM32Cube*, 2024. dirección: <https://www.st.com/en/ecosystems/stm32cube.html> (visitado 07-03-2024).
- [42] *STM32CubeMX*, 2024. dirección: <https://www.st.com/en/development-tools/stm32cubemx.html> (visitado 07-03-2024).
- [43] *STM32CubeIDE*, 2024. dirección: <https://www.st.com/en/development-tools/stm32cubeide.html> (visitado 07-03-2024).
- [44] *STM32CubeMonitor*, 2024. dirección: <https://www.st.com/en/development-tools/stm32cubemonitor.html> (visitado 07-03-2024).
- [45] *STM32CubeProg*, 2024. dirección: <https://www.st.com/en/development-tools/stm32cubeprog.html> (visitado 07-03-2024).
- [46] *STM32CubeL4*, 2024. dirección: <https://github.com/STMicroelectronics/STM32CubeL4/tree/master> (visitado 07-03-2024).
- [47] *μVision GUI*, 2024. dirección: <https://github.com/mupq/pqm4> (visitado 07-03-2024).
- [48] *INA219 Zero-Drift, Bidirectional Current/Power Monitor With I2C Interface*, 2015. dirección: https://www.ti.com/lit/ds/symlink/ina219.pdf?ts=1723724805837&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252Fes-mx%252FINA219%253Fbm-verify%253DAAQAAAAJ_-_-_-9ZEXlNQpN2tYM7QtmygnCEHfpzKoH6unttqZECxY5cY9dJtT_-s2R3eDAzfArUeFRvCfV4Ur08Y3HnKilKJ7Szrr9YX6z3mtMgAUHyZyfavn0iblkuyYFgzU0BjoNxskRH3NZ30R_-Rz4PP4y7yYuuafLrF1mdNPzHnvgmQUsBit_-Q5YboJ50t3vMdqkAWvwjk76camwz0fokZ6i2rXcC100bIFjenlbyFMy0912_-EbaAfidg74vDr2lgbl8JiSheo3NRK7xlrTBytuT54Il7WhxrhQp0yNis-XSxjL8gi6-4tJ9-1Wtgs- (visitado 07-03-2024).
- [49] *ESP Timer (High Resolution Timer)*, 2016. dirección: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/esp_timer.html (visitado 07-03-2024).
- [50] *High Level APIs*, 2012. dirección: https://www.raspberrypi.com/documentation/pico-sdk/high_level.html#group_pico_time (visitado 07-03-2024).

Apéndice A

Temporización y presupuesto

En este capítulo se incluye la temporización del proyecto y una estimación del coste total para la realización del mismo.

A.1 PPlanificación temporal

En primer lugar, se indica la organización temporal seguida a lo largo del proyecto mediante el diagrama mostrado en A.1 de acuerdo a las siguientes tareas:

- 1. Selección de algoritmos criptográficos.
- 2. Selección de dispositivos IoT empleados.
- 3. Análisis de compatibilidad.
- 4. Implementación de los algoritmos en los dispositivos seleccionados.
- 5. Diseño y realización de pruebas de rendimiento.
- 6. Documentación.

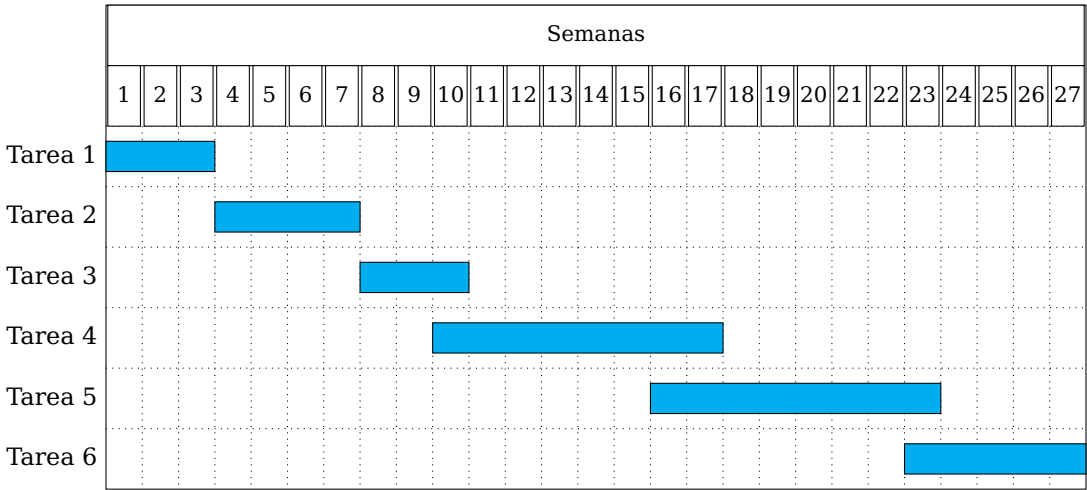


Figura A.1: Diagrama de Gantt

A.2 Recursos Hardware

Para poder llevar a cabo la realización de este proyecto, se han requerido varios dispositivos *hardware*. Estos han sido listados a continuación:

- Ordenador portátil.
- ESP-32 Kit de desarrollo C V4.
- ESP-Prog.
- Lilygo T-Display.
- Nucleo-L4R5ZI.

El precio de estos dispositivos se especifica en la tabla A.1 así como el total del coste de los dispositivos *hardware*.

Concepto	Precio por Unidad	Cantidad	Subtotal
Ordenador portátil	800,00 €	1	800,00 €
ESP-32 Kit de desarrollo C V4	12,99 €	1	12,99 €
ESP-Prog	14,63 €	1	14,63 €
Lilygo T-Display	8,99 €	1	8,99 €
Nucleo-L4R5ZI	20,22 €	1	20,22 €
TOTAL			856,83 €

Tabla A.1: Recursos hardware usados

A.3 Recursos Humanos

Es necesario incluir el coste que conlleva la contratación del personal para la ejecución del proyecto. En este caso, se ha requerido únicamente 1 ingeniero para llevarlo a cabo. Este coste se especifica en la tabla A.2.

Concepto	Precio por hora	Cantidad de horas	Subtotal
Ingeniero	40,00 €	600	24000,00 €
TOTAL			24000,00 €

Tabla A.2: Recursos humanos

A.4 Presupuesto de ejecución material

Finalmente, una vez se han especificado los distintos costes que se han afrontado para cada ámbito de este trabajo, se debe realizar la suma de los mismos para alcanzar el coste total del proyecto. La suma final se refleja en la tabla A.3

Concepto	Subtotal
Recursos hardware	856,83 €
Recursos software	0,00 €
Coste mano de obra	24000,00 €
TOTAL	24856,83 €

Tabla A.3: Presupuesto de ejecución material

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá