Estrategias de implementación de algoritmos criptográficos post-cuánticos

Implementation strategies for post-quantum cryptographic algorithms

TRABAJO FIN DE GRADO
CURSO 2022-2023

AUTORES
PETAR KONSTANTINOV IVANOV Y VÍCTOR MORENO PÉREZ

DIRECTORES
INMACULADA PARDINES LENCE Y MARCOS SÁNCHEZ-ÉLEZ MARTÍN

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Estrategias de implementación de algoritmos criptográficos post-cuánticos

Implementation strategies for post-quantum cryptographic algorithms

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

AUTORES

PETAR KONSTANTINOV IVANOV Y VÍCTOR MORENO PÉREZ

DIRECTORES

INMACULADA PARDINES LENCE Y MARCOS SÁNCHEZ-ÉLEZ MARTÍN

CONVOCATORIA: SEPTIEMBRE DE 2023

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Resumen

Estrategias de implementación de algoritmos criptográficos post-cuánticos

En los últimos años se han producido avances en la computación cuántica para poder elaborar algoritmos criptográficos que sean capaces de soportar y resistir ataques de ordenadores cuánticos. Estos son más potentes que un ordenador normal, ya que podrían explotar las vulnerabilidades de los algoritmos utilizados hoy en día.

Estos ordenadores supondrían una amenaza para los algoritmos de cifrado actuales, ya que la velocidad de procesado o computo de estos afecta significativamente a la seguridad que ofrecen, lo que permitiría romper aquellos algoritmos que hasta ahora creíamos que eran muy seguros. El sector de la criptografía post-cuántica está en un crecimiento constante y se espera que en los próximos 10 años estos algoritmos se usen para proteger la información importante del mundo.

Como consecuencia, NIST lanzó una convocatoria para elegir varios algoritmos criptográficos que pudiesen ser estandarizados y que pudiesen ser usados en el futuro por las grandes organizaciones para proteger sus datos. La selección de estos algoritmos es un proceso que dura mucho tiempo y que consta de distintas fases en las que estos algoritmos son sometidos a distintas pruebas por los expertos de criptografía.

En este contexto, en este Trabajo de Fin de Grado, primero, realizamos un análisis de aquellos algoritmos de cifrado que hemos considerado más interesantes dentro de la convocatoria NIST de algoritmos de cifrado post-cuántico. Después, nos centramos en el estudio del algoritmo McEliece, analizando varias implementaciones de dicho algoritmo y realizando un estudio de rendimiento de ellas. Por último, hemos realizado la paralelización de una de las implementaciones del algoritmo para ejecutarlo sobre GPUs, consiguiendo de esta forma una mejora del rendimiento del código.

Palabras clave: McEliece, NIST, computación cuántica.

Abstract

Implementation strategies for post-quantum cryptographic algorithms

Over the last years, there have been advances in quantum computing to be able to develop cryptographic algorithms that are capable of withstanding and resisting attacks from quantum computers. These are more powerful than normal computers, since they could exploit vulnerabilities in algorithms used today.

These computers would suppose a threat to current cryptographic algorithms, since their processing or computing speed significantly affects the security they offer. These would allow breaking those algorithms that until now we believed were very secure. The post-quantum cryptography sector is constantly growing and it is expected that these algorithms in 10 years will be used to protect the world's important information.

Consequently, NIST launched a calling signal to choose several cryptographic algorithms that could be standardized and that could be used in the future by large organizations to protect their data. The selection of these algorithms is a time-consuming process that consists of different phases in which these algorithms are put down to different tests by cryptography experts.

In this context, in this Final Degree Project, first, we carry out an analysis of those cryptographic algorithms that we have considered most interesting within the NIST signal call for post-quantum cryptographic algorithms. Afterwards, we focus on the study of the McEliece algorithm, analyzing several implementations of it and carrying out a performance study of them. Finally, we have parallelized one of the implementations of the algorithm to run it on GPUs, thus improving the performance of the code.

Keywords: McEliece, NIST, quantum computing.

ÍNDICE

1.INTRODUCCIÓN

1.1. Motivación

La seguridad informática es el área encargada de garantizar la confidencialidad y la integridad de la información y el correcto funcionamiento de las infraestructuras informáticas. El uso de las tecnologías de la información y de las comunicaciones es cada vez mayor y el número de ciberataques crece exponencialmente. El poder llegar a revelar, corromper o incluso modificar cierta información de terceros puede tener graves consecuencias, por lo que tener una buena seguridad es imprescindible en cualquier sistema informático. Como consecuencia la seguridad informática se ha convertido en uno de los principales retos de hoy en día para las empresas, organizaciones y gobiernos.

La criptografía es la ciencia encargada de estudiar las distintas técnicas usadas para transformar (cifrar) la información con el objetivo de hacerla irreconocible a receptores no autorizados. Por lo tanto, es una parte muy importante de la seguridad informática. El objetivo de la criptografía es proteger la información, cifrándola, y que solo el destinatario de la información sea capaz de recuperar (descifrar) la información original. Pero como todos sabemos, no hay nada seguro al 100% y hay algoritmos y mecanismos que pueden intentar descifrar la información sin conocer la clave necesaria para hacerlo. Entonces, es muy importante usar algoritmos de cifrado que sean robustos, es decir, muy difíciles de “romper”.

La integridad de la información es uno de los pilares fundamentales, que junto a la confidencialidad y la disponibilidad forman la conocida triada CIA. Es importante a la hora de garantizar la precisión y validez de la información evitar cualquier modificación sobre ellas no autorizada o no intencionada. Para mantener la integridad de los datos, se utilizan técnicas como el uso de firmas digitales, el control de versiones y los controles de cambios.

Previendo el avance de la computación cuántica en el futuro es necesario buscar nuevos algoritmos resistentes a este tipo de computación. Surge así la convocatoria del NIST para buscar algoritmos post cuánticos de firma digital y de intercambio de clave seguros. Entre los algoritmos de cifrado post-cuánticos que nos propusieron estudiar se encontraban el SIKE, el BIKE, el HQC y el McEliece. Estos algoritmos forman parte de la cuarta ronda de NIST para encontrar algoritmos de cifrado de clave pública con el propósito anteriormente mencionado. Esta ronda se lanzó en el año 2022 y concluyó en octubre de 2022 que fue cuando terminaba el tiempo para adjuntar las modificaciones de aquellos seleccionados. Estos son los algoritmos que siguen todavía en la competición. Sin embargo, NIST lanzó este concurso en el verano de 2016 y 7 años más tarde y cuatro rondas después, todavía se siguen buscando algoritmos que sean capaces de luchar contra los ordenadores cuánticos.

Tras realizar un extensivo estudio de los algoritmos propuestos, hemos ido descartando algunos de ellos hasta quedarnos finalmente solo con el algoritmo de McEliece. Descartamos el algoritmo SIKE porque un ordenador single-core [1] lo consiguió romper en una hora. En cuanto al algoritmo BIKE, su implementación está en VHDL y este lenguaje no resulta adecuado para los objetivos buscados en este trabajo. También descartamos el algoritmo HQC, porque la cantidad de información publicada sobre él es escasa, lo que complicaba nuestra investigación.

Finalmente, el algoritmo de cifrado post-cuántico McEliece acabó siendo nuestra opción definitiva ya que formó parte de la tercera ronda y fue uno de los finalistas de la última fase de esta ronda. Además, fue nuestra principal opción desde el principio, debido a la gran cantidad de información publicada y a la gran variedad de implementaciones encontradas en distintos lenguajes de programación.

1.2. Objetivos

A continuación, presentamos los objetivos en los que pondremos nuestro foco en el estudio y realización del presente Trabajo de Fin de Grado:

- Implementar de forma paralela uno de los algoritmos finalistas de la convocatoria del National Institute of Standars and Technology de Estados Unidos (NIST).
- Analizar desde un punto de vista funcional y de implementación los algoritmos finalistas de la convocatoria NIST con especial énfasis en aquel que consideremos que tiene más posibilidades de ser paralelizado.
- Comprender la amenaza que supone la computación cuántica, y la criptografía post-cuántica como futura solución a la exposición de la seguridad informática a ser cada vez más vulnerable.
- Analizar y comparar el rendimiento de las distintas implementaciones buscadas y estudiadas anteriormente.
- Modificar el código para que tome mensajes por consola y los convierta a binario para su posterior cifrado y descifrado (en la implementación original eran mensajes predefinidos en binario de un tamaño estándar).

1.3 Planificación

En este apartado expondremos una breve tabla con aquello que hemos ido realizando por etapas, a esto se le debería de agregar la lectura y actualización de información ya que el proceso NIST sigue vigente.

07/09/2022-10/10/2022	

	Se analizó la documentación proporcionada por los tutores sobre la amenaza de la computación cuántica y sobre la criptografía post-cuántica. Se realizó una pequeña investigación sobre los cuatro finalistas de la cuarta ronda del concurso de NIST y se procedió a realizar un pequeño documento con información sobre los algoritmos SIKE, BIKE, HQC y Classic McEliece. Además, se seleccionó el algoritmo Classic McEliece después de consultarlo con los tutores debido a la abundante información disponible sobre él.
11/10/2022- 23/1/2023	Se realizó una búsqueda intensiva para encontrar las mejores implementaciones posibles del algoritmo post-cuántico de Classic McEliece. Al final se encontraron tres implementaciones óptimas de este algoritmo en repositorios GitHub para trabajar con ellas. Se comenzó con la recopilación de información y el primer borrador de la memoria realizando una primera versión del resumen, la motivación, los objetivos del trabajo de fin de grado y la explicación de los lenguajes de programación utilizados para la implementación. Se comprobó que las implementaciones de los algoritmos funcionaban de forma correcta y se hizo un estudio del código para la comprensión de su funcionamiento.
24/1/2023 – 26/3/2023	Se realizaron mejoras sobre la implementación añadiendo la posibilidad de introducir mensajes en formato texto que posteriormente serán traducidos a formato binario para poder realizar la ejecución del cifrado. Se realizaron correcciones sobre el código de la implementación y se añadió una nueva función que rellena de forma automática con ceros, cuya finalidad es completar el texto para que sea múltiplo de 8. Se cambió el entorno de trabajo a GoogleColab que permite establecer un emulador de GPU para poder ejecutar las nuevas partes del código realizadas en lenguaje Open ACC para poder ejecutar de forma paralela la ejecución. Por último, se creó un repositorio de GitHub donde se han subido las versiones de la implementación.
27/3/2023 - 15/5/2023	Se escribió información detallada sobre el algoritmo SIKE y se explicó porque se decidió descartarlo. Se corrigieron

	errores relacionados con el capítulo 2 llamado Proceso de estandarización PQC y posteriormente se arreglaron problemas previos que fueron indicados por los tutores. Se buscó información detallada sobre los algoritmos post-cuanticos CRYSTALS-Kyber, McEliece, SABER y NTRU que fueron los finalistas de la tercera ronda del concurso de NIST y con ella se redactaron las correspondientes secciones de la memoria.
16/5/2023- 20/6/2023	Se escribió el capítulo relacionado con la implementación seleccionada, explicando en detalle todos los archivos que contiene, las funciones de cada archivo y para qué sirve cada archivo. Además, se le añadieron figuras para una mayor comprensión ya que hay muchas funciones y nombres. Después, se añadieron tablas con las comparaciones de tiempo y se especificó qué se ha paralelizado y por qué se ha llevado a cabo su paralelización.
20/6/2023 -18/7/2023	Se realizaron nuevas mejoras de la implementación seleccionada. Además, se añadió más información sobre las investigaciones realizadas acerca de los algoritmos seleccionados y se refinó la redacción de la memoria.
18/7/2023 - 4/9/2023	Se realizaron scripts de RSA y DH utilizando OpenSSL para realizar una comparación con nuestras implementaciones. Además, se realizaron correcciones de memoria, se añadieron citas bibliográficas y se ultimaron detalles. Además, se escribieron las secciones que faltaban por terminar.

1.4. Estado del arte

Uno de los algoritmos de cifrado más utilizados para firma digital en la actualidad es el RSA. Este algoritmo fue desarrollado en 1977 por Rivest, Shamir y Adleman. Su seguridad radica en la complejidad de la factorización de números enteros, pues su funcionamiento se basa en hacer el producto de dos números primos elegidos al azar y mantenidos en secreto. Encontrar estos dos números a partir de su producto es algo demasiado costoso para los ordenadores clásicos. Además, el tamaño de los dos números primos se hace mayor a medida que aumenta la capacidad de cálculo de los ordenadores. Sin embargo, la resolución de este problema dejará de ser imposible para los ordenadores cuánticos ya que mediante el algoritmo de Shor se podrá factorizar un número entero en tiempo polinomial. De todo esto surge la necesidad de buscar nuevos algoritmos más resistentes. Otro de los algoritmos que será vulnerable a la computación cuántica es el algoritmo Diffie-Hellman, que se utiliza para el intercambio de clave

simétrica y se basa en algoritmos discretos. El algoritmo Diffie-Hellman permite a dos partes acordar una clave secreta de forma segura. La clave secreta no viaja, si no que las dos partes acaban generando la misma clave. Para poder entender esto, pongamos un ejemplo muy sencillo:

- Tenemos dos ordenadores: un ordenador A tiene como clave secreta el color rojo y otro ordenador B tiene el color azul.
- Ambos ordenadores se ponen de acuerdo para tener el color amarillo como mezcla (este color se mezclará con las claves privadas de cada uno).
- Entonces al mezclar, el ordenador A tendría un color naranja y el ordenador B tendría un color verde.
- Ahora ambos ordenadores mandan el nuevo color que han generado al otro para que pueda seguir realizándose el proceso de creación de la clave secreta.
- A continuación, mezclan el color recibido con su clave secreta y este color resultante sería la clave secreta común.

También es importante mencionar el uso para la generación de firmas digitales del algoritmo DSA [2], que es un estándar del Gobierno Federal de los Estados Unidos de América. El algoritmo DSA se hizo público el 30 de agosto de 1991. Es un algoritmo puramente asimétrico, al igual que el RSA. Este algoritmo sirve para firmar, pero no para cifrar información. Una desventaja de este algoritmo es que requiere mucho más tiempo de cómputo que el RSA.

La duda de si la criptografía no resistirá futuros ataques surge unos años atrás, se pueden encontrar artículos del año 2009 poniendo en duda que los actuales algoritmos fuesen capaces de aguantar los ataques de un futuro ordenador cuántico y explicando que, dentro de poco la aplicación del algoritmo de Shor, que es un algoritmo capaz de descomponer en factores un número N de manera eficiente, desde un ordenador más potente supondrá una gran amenaza para la criptografía.

1.5. Estructura de la memoria

Nuestra memoria ha sido estructurada en cinco capítulos, los cuáles, vamos a describir brevemente a continuación.

En el capítulo de Introducción, se mencionan las motivaciones, los objetivos, la estructura de la memoria y el estado del arte, describiendo así tanto el problema a resolver, como el estado actual de los algoritmos de cifrado.

En el segundo capítulo se explica cómo surgió el concurso para elegir los mejores algoritmos post-cuánticos por parte de NIST y cuántas rondas se han realizado hasta la fecha. Posteriormente, se definen los algoritmos finalistas de la cuarta ronda explicando cómo funcionan y por qué hemos descartado los algoritmos SIKE, BIKE y HQC. Finalmente, se describen los algoritmos finalistas de la tercera ronda del concurso, exceptuando el Classic McEliece que hemos explicado en una sección aparte.

Analizamos cómo funcionan estos algoritmos y mencionamos cuáles son sus ventajas e inconvenientes.

En el tercer capítulo explicamos cuáles han sido los lenguajes de programación utilizados. Además, hemos añadido una breve explicación matemática de los códigos binarios de Goppa empleados en el funcionamiento del algoritmo de cifrado McEliece y explicamos cómo están estructuradas las implementaciones encontradas. También mostramos una tabla con los tiempos de ejecución para cada una de estas implementaciones.

En el cuarto capítulo describimos de forma precisa los ficheros que conforman la implementación con la que hemos decidido trabajar y las funciones que los componen. Asimismo, hemos realizado una mejora de rendimiento mediante la paralelización de la implementación para poder ejecutarla ensobre GPU y explicamos cómo y por qué se pueden paralelizar. Además, hemos añadido una tabla comparativa con los tiempos de ejecución de antes y después de realizar las mejoras en el código.

En el quinto capítulo, explicamos como se ejecutan nuestras implementaciones y realizamos una comparativa entre ellas. Además, extraemos información de otras investigaciones de McEliece. Y finalizamos el capítulo comparando la versión sin paralelizar y nuestra segunda versión paralelizada con los algortimos de RSA y Diffie-Hellman (DH) para obtener una visión de los tiempos de unos y otros algoritmos.

Y, por último, un sexto capítulo en el que describimos nuestras aportaciones a este trabajo, los objetivos logrados y las principales conclusiones que se pueden extraer.

1.INTRODUCTION

1.1. Motivation

Computer security is the area in charge of guaranteeing the confidentiality and integrity of information and the correct functioning of computer infrastructures. The use of information and communications technologies is increasing and the number of cyberattacks is growing exponentially. Being able to reveal, corrupt or even modify certain third-party information can have serious consequences, this is why having good security is essential in any computer system. As a consequence, computer security has become one of the main challenges today for companies, organizations and governments.

Cryptography is the science in charge of studying the different techniques used to transform (encrypt) information with the aim of making it unrecognizable to unauthorized receptors. Therefore, it is a very important part of computer security. The aim of cryptography is to protect the information, encrypting it, and that only the receptor of the information is able to recover (decrypt) the original information. But as we all know, nothing is 100% secure and there are algorithms and mechanisms that can try to decrypt information without knowing the key needed to do so. So, it is very important to use cryptographic algorithms that are robust, meaning they are difficult to "break".

The integrity of the information is one of the fundamental pillars, which together with confidentiality and availability form the well-known CIA triad. It is important to ensure the accuracy and validity of the information to avoid any unauthorized or unintentional modification to it. Techniques such as the use of digital signatures, version control, and change controls are used to maintain data integrity.

Anticipating the advance of quantum computing in the future, it is necessary to look for new algorithms resistant to this type of computing. Thus, the call from NIST arises to search for post-quantum algorithms for digital signature and secure key exchange. Among the post-quantum encryption algorithms that we were proposed to study were SIKE, BIKE, HQC and McEliece. These algorithms are part of NIST's fourth round of finding public key encryption algorithms for the aforementioned purpose. This round was launched in 2022 and concluded in October 2022, which was when the time to attach the modifications of those selected ended. These are the algorithms that are still in the competition. However, NIST launched this contest in the summer of 2016 and 7 years later and four rounds later, they are still looking for algorithms that are capable of fighting quantum computers.

After carrying out an extensive study of the proposed algorithms, we have discarded some of them until we are finally left with only the McEliece algorithm. We discarded the SIKE algorithm because a single-core computer managed to break it in one hour. Regarding the BIKE algorithm, its implementation is in VHDL and this language is not

suitable for the objectives sought in this work. We also discarded the HQC algorithm, because the amount of information published about it is scarce, which complicated our investigation.

Finally, the McEliece post-quantum encryption algorithm ended up being our final choice as it was part of the third round and was one of the finalists in the last phase of this round. Furthermore, it was our main choice from the beginning, due to the large amount of information published and the great variety of implementations found in different programming languages.

1.2. Objectives

Below, we present the objectives on which we will focus in the study and completion of this Final Degree Project:

- Implement in parallel one of the finalist algorithms from the call of the National Institute of Standards and Technology of the United States (NIST).
- Analyze the finalist algorithms of the NIST call from a functional and implementational point of view with special emphasis on the one that we consider has the most possibilities of being parallelized.
- Understand the threat meant by quantum computing, and post-quantum cryptography as a future solution to the exposure of computer security to being increasingly vulnerable.
- Analyze and compare the performance of the different implementations searched and studied previously.
- Modify the code so that it takes messages from console and translates them to binary for subsequent encryption and decryption (in the original implementation they were predefined binary messages of a standard size).

1.3 Planification

In this section we will present a brief table with what we have been doing in stages, to this we should add the reading and updating of information since the NIST process is still in force.

07/09/2022-10/10/2022	The documentation provided by the tutors on the threat of quantum computing and post-quantum cryptography was analyzed. A small investigation was carried out on the four finalists of the fourth round of the NIST contest and a small document was prepared with information on the SIKE, BIKE, HQC and Classic McEliece algorithms. Furthermore, the Classic McEliece algorithm was selected after

	consultation with the tutors due to the abundant information available about it.
11/10/2022- 23/1/2023	An intensive search was done to find the best possible implementations of Classic McEliece's post-quantum algorithm. In the end, three optimal implementations of this algorithm were found in GitHub repositories to work with. We began with the collection of information and the first draft of the memory, making a first version of the summary, the motivation, the objectives of the Final Degree Project and the explanation of the programming languages used for the implementation. The correct working of the implementations of the algorithms was checked and a study of the code was made to understand its operation.
24/1/2023 – 26/3/2023	Improvements were made to the implementation by adding the possibility of entering messages in text format that will later be translated into binary format to be able to execute the encryption. Corrections were made to the implementation code and a new function was added that automatically fills with zeros, whose purpose is to complete the text so that it is a multiple of 8. The work environment was changed to GoogleColab, which allows establishing a GPU emulator to be able to execute the new parts of the code made in Open ACC language to be able to execute the execution in parallel. Finally, a GitHub repository was created where the implementation versions have been uploaded.
27/3/2023 - 15/5/2023	Detailed information about the SIKE algorithm was written and it was explained why it was decided to discard it. Errors related to chapter 2 called PQC standardization process were corrected and previous problems that were indicated by the tutors were subsequently fixed.
16/5/2023- 20/6/2023	The chapter related to the selected implementation was written, explaining in detail all the files it contains, the

	functions of each file and what each file is for. In addition, figures were added for greater understanding since there are many functions and names. Afterwards, tables were added with the time comparisons and it was specified what has been parallelized and why its parallelization has been carried out.
20/6/2023 -18/7/2023	New improvements were made to the selected implementation. In addition, more information on the research carried out on the selected algorithms was added and the wording of the memory was refined.
18/7/2023 - 4/9/2023	RSA and DH scripts were made using OpenSSL for comparison with our implementations. In addition, memory corrections were made, bibliographic citations were added and details were finalized. In addition, the sections that remained to be completed were written.

1.4. State of the art

Nowadays RSA is one of the most used encryption algorithms for digital signatures. RSA algorithm was developed in 1977 by Rivest, Shamir and Adleman. Its security lies in the complexity of factoring integers, since its operation is based on making the product of two prime numbers chosen at random and kept secret. Finding these two numbers from its product is too expensive for classical computers. Furthermore, the size of the two prime numbers becomes larger as the computing power of computers increases. However, solving this problem will no longer be impossible for quantum computers since Shor's algorithm can factorize an integer in polynomial time. From all this arises the need to look for new, more resistant algorithms. Another algorithm that will be vulnerable to quantum computing is the Diffie-Hellman algorithm, which is used for symmetric key exchange and is based on discrete algorithms. The Diffie-Hellman algorithm allows two parts to securely agree on a secret key. The secret key does not travel, but rather the two parts end up generating the same key. To understand this, let's take a very simple example:

- We have two computers: a computer A has the color red as a secret key and another computer B has the color blue.
- Both computers agree to have the color yellow as a mix (this color will be mixed with each other's private keys).
- So when mixing, computer A would have an orange color and computer B would have a green color.

- Now both computers send the new color they have generated to the other so that the process of creating the secret key can continue.
- They then mix the received color with their secret key and this resulting color would be the common secret key.

It is also important to mention the use of the DSA algorithm for generating digital signatures, which is a standard of the Federal Government of the United States of America. The DSA algorithm was made public on August 30, 1991. It is a purely asymmetric algorithm, just like RSA. This algorithm is used to sign, but not to encrypt information. A disadvantage of this algorithm is that it requires much more computing time than RSA.

The question of whether cryptography will not resist future attacks arose a few years ago. Articles from 2009 can be found questioning whether current algorithms would be able to withstand the attacks of a future quantum computer and explaining that, soon the application of Shor's algorithm, which is an algorithm capable of factoring a number N efficiently, from a more powerful computer will pose a great threat to cryptography.

1.5. Memory structure

Our memory has been structured into six chapters, which we will briefly describe below.

In the Introduction chapter, the motivations, objectives, memory structure and state of the art are mentioned, thus describing both the problem to be solved and the current state of encryption algorithms.

The second chapter explains how the contest to choose the best post-quantum algorithms by NIST came about and how many rounds have been carried out to date. Subsequently, the finalist algorithms of the fourth round are defined, explaining how they work and why we have discarded the SIKE, BIKE and HQC algorithms. Finally, the finalist algorithms of the third round of the contest are described, except for the Classic McEliece that we have explained in a separate section. We analyze how these algorithms work and mention their advantages and disadvantages.

In the third chapter we explain which programming languages have been used. In addition, we have added a brief mathematical explanation of the Goppa binary codes used in the operation of the McEliece encryption algorithm and we explain how the found implementations are structured. We also show a table with the execution times for each of these implementations.

In the fourth chapter we describe precisely the files that make up the implementation with which we have decided to work and the functions that compose them. We have also made a performance improvement by parallelizing the implementation to be able to run it on GPU and explain how and why they can be parallelized. In addition, we have added a comparative table with the execution times before and after making the improvements in the code.

In the fifth chapter, we explain how our implementations are executed and we make a comparison between them. In addition, we draw information from other McEliece investigations. And we end the chapter by comparing the unparallelized version and our second parallelized version with the RSA and Diffie-Hellman (DH) algorithms to obtain a view of the times of each algorithm.

And finally, a sixth chapter in which we describe our contributions to this work, the objectives achieved and the main conclusions that can be drawn.

2. Proceso de estandarización PQC de NIST

El Instituto Nacional de Estándares y Tecnología de EE. UU. (NIST) [3] ha iniciado un proceso para solicitar, evaluar y estandarizar uno o más algoritmos de criptografía de clave pública resistentes a la computación cuántica. Actualmente, los algoritmos de cifrado de clave pública se especifican en el estándar de firma digital (FIPS 186-4), en la recomendación para esquemas de establecimiento de claves por pares utilizando cifrados de logaritmo discreto (SP 800-56A Revisión 2) y en la recomendación para esquemas de establecimiento de claves por pares utilizando cifrados de factorización de enteros (SP 800-56B Revisión 1). Sin embargo, estos algoritmos son vulnerables a futuros ataques de computadoras cuánticas a gran escala. Los ordenadores hoy en día no tienen suficiente capacidad para romper los algoritmos de cifrados clásicos actuales y los algoritmos de firma digital, pero la situación puede cambiar en los próximos años y es necesario preparar la base para la transferencia de criptosistemas a nuevos estándares.

Se pretende que los nuevos estándares de cifrado de clave pública especifiquen uno o más algoritmos adicionales de firma digital no clasificada y divulgada públicamente, cifrado de clave pública y establecimiento de claves que estén disponibles en todo el mundo y sean capaces de proteger la información confidencial en un futuro previsible, incluso después de la llegada de las computadoras cuánticas.

El concurso se inició en el año 2016 con el proceso de estandarización PQC (Post-Quantum Cryptography Standardization Process). Tiene como objetivo elegir algoritmos de cifrado post-cuántico adecuados para su promoción como estándares. Los algoritmos propuestos, 23 algoritmos de firma digital y 59 algoritmos de cifrado con distintas aproximaciones, fueron analizados por expertos independientes en busca de posibles vulnerabilidades y debilidades.

El 15 de julio de 2022 hubo una nueva ronda sobre algoritmos universales de cifrado post-cuántico. Entre los finalistas se encontraba el algoritmo de McEliece, que aún no se estandarizaría debido al gran tamaño de su clave pública.

El concurso hasta ahora ha realizado un total de 4 rondas. Una ronda suele durar entre 12 y 18 meses, ahora mismo el concurso se encuentra en la ronda 4 donde compiten los algoritmos BIKE, McEliece y HQC, basados en código, y por otra parte el algoritmo SIKE, basado en isogenia de la curva elíptica supersingular. En la ronda 3 el ganador fue CRYSTALS-Kyber [4] que ganó a NTRU porque este algoritmo generaba las claves lentamente y tenía claves públicas y textos cifrados muy largos. El coste total de la generación de las claves en NTRU es un 30% superior al de Kyber, lo que hace imposible su implementación en dispositivos limitados, es decir, en dispositivos con unos recursos limitados. Otro de los finalistas fue el algoritmo SABER, que tenía el coste más bajo gracias a sus claves públicas y a sus textos cifrados de pequeño tamaño. Esto le permitió competir con Kyber, ya que al igual que este algoritmo se puede implementar en dispositivos limitados. SABER perdió simplemente porque Kyber era el número uno en

todos los *benchmarks* y pruebas realizadas. El último finalista de la ronda 3 fue Classic McEliece, que tenía un rendimiento diferente respecto a los otros finalistas por lo que no fue comparado directamente con el rendimiento del resto. Utiliza claves públicas y privadas grandes pero la velocidad de su cifrado y descifrado y el pequeño tamaño de sus textos de cifrado lo mantuvieron a la par con el resto, por ello este algoritmo puede ofrecer el mejor rendimiento en aplicaciones donde no se tiene en cuenta la generación de las claves en el coste total. Este algoritmo es unos de los participantes de la ronda 4 anunciados en julio de 2022.

2.1. Algoritmo post-cuántico SIKE (Supersingular Isogeny Key Encapsulation)

SIKE [5] es un algoritmo post-cuántico para establecer una clave secreta entre dos partes a través de un canal de comunicaciones no confiable. Este algoritmo es análogo al intercambio de claves de Diffie-Hellman, pero se basa en recorridos en un grafo de isogenia supersingular y fue diseñado para resistir el ataque criptoanalítico de un atacante en posesión de una computadora cuántica. Cuenta con una de las claves de cifrado con menor tamaño entre los algoritmos post-cuánticos, sus claves públicas son de 2688 bits y su nivel de seguridad cuántica de 128 bits. Proporciona secreto perfecto hacia adelante para evitar que las claves comprometidas vulneren confidencialidad de futuras sesiones, es decir que, en el caso de que se descubra la clave privada, toda la información compartida en el futuro estaría libre del peligro de ser descubierta. Este algoritmo es el principal candidato para sustituir al algoritmo Diffie-Hellman efímero y al algoritmo de curva elíptica Diffie-Hellman efímero.

SIKE tiene como ventaja principal el pequeño tamaño de sus claves comparándolo con otras primitivas que ofrecen una seguridad cuántica razonable. Las claves públicas que están sin comprimir pueden ser de distintos tamaños, como las versiones de 330 y 378 bytes que se comparan con el módulo de 384 bytes que ofrece una seguridad clásica de 128 bits. Por otro lado, están los modelos de mayor tamaño con claves de 462 bytes y 564 bytes respectivamente, con textos de cifrado de 596 bytes. Se ha propuesto una mejora sobre la última versión que permite comprimir aún más las claves públicas, de tal manera que ofrece una mejora general de un 60% de su tamaño original. Además, se mejora el rendimiento de un 139% a un 160% durante la generación de las claves públicas, del 66% al 90% durante el cifrado y del 59% al 68% durante el descifrado.

Por otro lado, otra ventaja de este algoritmo es que en las librerías que usa se encuentra todo lo necesario para implementar de forma segura la curva elíptica de Diffie-Hellman con un esquema híbrido de intercambio de claves con un coste mínimo de programación. Sin embargo, respecto a los otros candidatos, la principal desventaja de SIKE es que es de los algoritmos de peor rendimiento.

La razón por la que decidimos descartar este algoritmo fue porque se consiguió romper usando un ordenador con un solo núcleo. Los investigadores del grupo de Seguridad Informática de Criptografía Industrial de KU Leuven acabaron con el algoritmo atacando la matemática en la que se basaba su diseño en vez de intentar romper su seguridad buscando vulnerabilidades en su código. Utilizaron el teorema de Glue and Split, al que el algoritmo SIDH (similar al SIKE) es vulnerable. Este algoritmo usa curvas de género 2 para atacar las curvas elípticas. El creador del algoritmo, David Jao, salió en su defensa afirmando que sus criptógrafos son especialistas en matemáticas puras y no se esperaban estos resultados. Sin embargo, esto supone un grave problema de seguridad, ya que demuestra que se puede romper con un simple ordenador y suficiente conocimiento de las matemáticas en las que se basa el algoritmo.

2.2. Algoritmo post-cuántico BIKE (Bit Flipping Key Encapsulation)

BIKE [6] [7] es un KEM (Key Encapsulation Protocol) basado en QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check), que puede ser fácilmente descifrado mediante técnicas basadas en intercambios de bits. Fue diseñado originalmente para protocolos de comunicación síncrona (como TLS) con claves efímeras. Solo debería estar permitido descifrar una única vez con una clave privada dada, lo que hace que proporcione secreto perfecto hacia delante. La reutilización de claves o adaptación a protocolos de comunicación asíncrona, como puede ser el caso de Gmail, requiere utilizar claves estáticas de larga duración, que no garantizan el secreto perfecto hacia delante.

Lo mencionado anteriormente lleva a dos conclusiones. La primera, que es inmune a los ataques de reacción GJS, ya que estos ataques necesitan observar una gran cantidad de textos descifrados para una misma clave privada (algo que es imposible si están siendo utilizadas claves efímeras). Y la otra es que la generación de las claves debe ser eficiente debido a que se realiza en cada encapsulación de claves. Podemos ver los tamaños de las claves en la Tabla 1.

La seguridad IND-CPA es alcanzada para BIKE si los parámetros son escogidos de manera que los problemas computacionales subyacentes genéricos cuasi-cíclicos basados en código son lo suficientemente difíciles. Es decir, si los posibles atacantes no pueden distinguir entre los cifrados de dos mensajes a su elección.

La seguridad IND-CCA es alcanzada para BIKE si BIKE es instanciado con un descifrador que tenga un DFR (Decoding Failure Rate). IND-CCA es un modelo de ataque de criptoanálisis que permite recoger información mediante la obtención de los textos descifrados a partir de textos cifrados elegidos.

	TAMAÑO	NIVEL 1	NIVEL 3	NIVEL 5
Clave Privada	l+w *$[log_2(r)]$	2,244	3,346	4,640
Clave Pública	r	12,323	24,659	40,973
Texto Cifrado	r + l	12,579	24,915	41,229

Tabla 1: Tamaños mínimos de la clave pública, de la clave privada y del texto a cifrar en bits requeridos por el algoritmo BIKE

2.3. Algoritmo post-cuántico Classic McEliece

Classic McEliece [8] [9]Es un algoritmo de cifrado asimétrico creado en 1978 por Robert McEliece, que fue el primero en utilizar aleatorización en el proceso de cifrado. Se considera un algoritmo post-cuántico ya que es capaz de resistir ataques que utilizan el algoritmo Shor y su seguridad se basa en la dureza de la decodificación de un código lineal general.

Para la descripción de la clave privada se usa un código de corrección de errores para el que se conoce un algoritmo de decodificación capaz de corregir un número finito de errores. Utiliza códigos de Goppa que se pueden decodificar de manera eficiente con un algoritmo de Patterson. La clave pública se deriva de la privada cubriendo el código seleccionado como código general, para ello la matriz generadora del código es perturbada por dos matrices invertibles al azar. Los ataques más efectivos contra este algoritmo son ataques que usan algoritmos de decodificación de conjuntos de información. Respecto al RSA este algoritmo ofrece un cifrado y descifrado más rápido, y también se puede usar para generar construir firmas. Una de sus desventajas es el uso de matrices de gran tamaño para las claves pública y privada.

Los ataques de fuerza bruta no tienen éxito ya que los algoritmos para decodificar el conjunto de la información tienen un tiempo de ejecución exponencial. Los ataques estructurales son más eficaces, pero al usar códigos Goppa son más resistentes y persistentes a la hora de ser decodificados. McEliece sugirió originalmente tamaños de parámetros de seguridad de n=1024, k=524, t=50 dando como resultado un tamaño de clave pública de 524*(1024−524) = 262.000 bits.

En conclusión, la ventaja más importante del McEliece es la seguridad. Sin embargo, la eficiencia, concretamente en la generación de claves, no es su punto fuerte. Para ello, las aplicaciones que lo usen deben aumentar el tiempo de vida de dichas claves para reducir el coste de generar y distribuir las claves. El cifrado y el descifrado son razonablemente rápidos en software e impresionantemente rápidos en hardware.

2.4. Algoritmo post-cuántico HQC (Hamming Quasi-Cyclic)

HQC [10] Es un esquema de cifrado de clave pública basado en un código diseñado para proporcionar seguridad contra ataques de computadoras clásicas y cuánticas. Ofrece un tamaño de claves más grande que el clásico McEliece. Provee un análisis detallado de la probabilidad de fallo del cifrado, lo que hace posible elegir parámetros que probablemente eviten ataques reactivos que comprometen la seguridad de los esquemas de seguridad del QC-LDPC y QC-MDPC. Esto lo convierte en uno de los mejores candidatos en el proceso de estandarización del NIST de algoritmos de cifrado post-cuánticos. Este algoritmo tiene una estructura específica que permite abrir la puerta a ciertos ataques específicos estructurales [11]. El primer ataque genérico es el ataque DOOM, que debido a la ciclicidad implica una ganancia de O (√ n). También es posible considerar ataques a la forma del polinomio que genera la estructura cíclica. Estos ataques son especialmente eficaces cuando el polinomio X^(n-1) tiene muchos factores de bajo grado. HQC usa dos tipos de código: uno decodificable que es capaz de corregir un número determinado de errores a través de un algoritmo eficiente de C y otro código de doble circulación aleatoria con una matriz de comprobación. En la Figura 1 podremos ver un ejemplo simplificado de su funcionamiento.

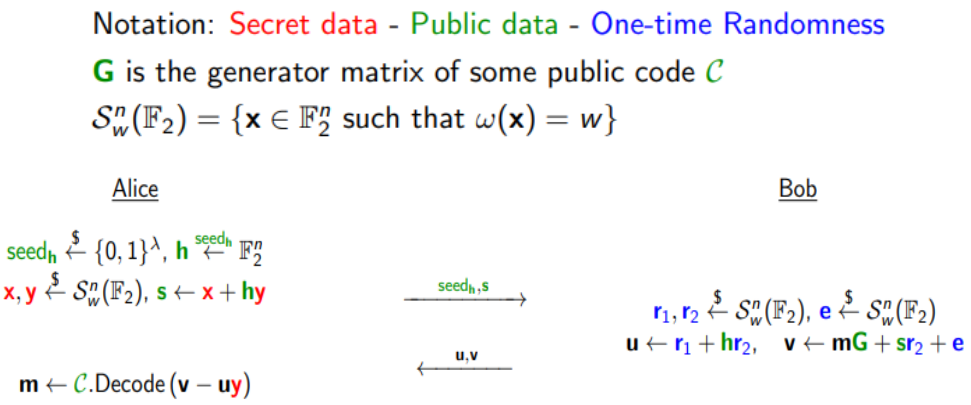

Figura 1: Esquema de ejemplo funcionamiento HQC.

2.5. Finalistas de la tercera ronda del concurso

La tercera ronda del concurso de NIST para encontrar algoritmos post-cuánticos finalizó en julio de 2022 y en ella se seleccionaron cuatro algoritmos resistentes, que tanto los ordenadores convencionales como los cuánticos deberían de tener dificultades para romper. Los algoritmos están diseñados para dos tareas principales para las que normalmente se utiliza el cifrado: cifrado general, utilizado para proteger la información intercambiada a través de una red pública; y firmas digitales, utilizadas para la autenticación de identidad. Estos algoritmos finalistas fueron: Classic McEliece, CRYSTALS-Kyber, NTRU y SABER.

2.5.1 CRYSTALS-Kyber

El algoritmo post-cuántico CRYSTALS-Kyber [12] fue unos de los finalistas y posteriormente ganador de la tercera ronda del concurso NIST. Ha sido seleccionado

para que pueda ser estandarizado para el cifrado con la finalidad de proteger la información de ataques de ordenadores cuánticos. Kyber utiliza un sistema de cifrado basado en claves públicas, cuyo uso principal es establecer sistemas de claves simétricas utilizando protocolos de alto nivel como TLS, Signal o OpenPGP. El algoritmo tiene tres niveles de seguridad y los tamaños de las claves son similares a los tamaños de las claves del algoritmo RSA.

Al ser un algoritmo que utiliza claves públicas y privadas, para poder realizar el cifrado y descifrado, se necesitan operaciones de polinomios y un módulo q. Para la generación de la clave privada se necesitan dos polinomios con pequeños coeficientes y para la generación de la clave pública se necesitan dos elementos: Una matriz de polinomios aleatorios y un vector de polinomios que necesita un vector adicional de errores. Este proceso de generación de claves permite que sea casi imposible recuperar la clave privada ya que el algoritmo utiliza el módulo de aprendizaje con errores.

El proceso de cifrado utiliza un error y un vector de números aleatorios. Cada uno de esos vectores se crean desde cero para cada cifrado que se va a realizar. Para cifrar el mensaje hay que convertirlo en un polinomio. Esto se hace utilizando la forma binaria del mensaje, cada bit del mensaje se usa como coeficiente, y luego hay que escalar el polinomio. En el proceso de descifrado, primero, se necesita crear un resultado “noisy”, esto quiere decir que no tiene ninguna importancia. Esto es debido a que después de realizar la creación de ese resultado, se emplea la clave privada para descifrar el texto cifrado y recuperar el texto original.

Kyber es un sistema de cifrado de tipo que utiliza el sistema de módulo de aprendizaje con errores, esto permite que el proceso de descifrado sea un proceso bastante complicado de averiguar para los ordenadores cuánticos. Es decir, es complicado de revertir el proceso aplicado para generar las claves y conseguir obtener así la clave privada. Es un algoritmo que tiene las ventajas de los algoritmos basados en celosías, entre las que destaca la rapidez.

2.5.2. NTRU

NTRU [13]es un algoritmo post-cuántico finalista de la tercera ronda del concurso de NIST. Utiliza criptografía basada en celosía para cifrar y descifrar datos y es resistente a los ataques que utilizan el algoritmo de Shor.

Este algoritmo realiza operaciones de clave privada mucho más costosas y rápidas que el algoritmo RSA. Este algoritmo no es vulnerable a ataques de ordenadores cuánticos como lo son el RSA y la criptografía de curva elíptica.

La seguridad del algoritmo procede de la interacción del sistema de cruce de polinomios con la reducción de independencia de los módulos p y q. En el *framework* del algoritmo, su cifrado utiliza un elemento aleatorio que permite que cada mensaje tenga variedad de cifrados. El cifrado, el descifrado y la creación de las claves son operaciones fáciles, rápidas y sencillas. Se usan operaciones cuadráticas para realizar el cifrado y el descifrado de mensajes de longitud N, haciéndolo mucho más rápido que las

operaciones cúbicas que requiere el algoritmo RSA. Las claves tienen una longitud lineal que resulta ser bastante buena en comparación con las claves cuadráticas de otros algoritmos rápidos como el McEliece.

2.5.3. SABER

SABER [14] es un algoritmo post-cuántico que fue finalista en la tercera ronda del concurso de NIST. Este algoritmo utiliza el mecanismo de cifrado de claves más conocido como KEM, que ofrece un método eficiente de transferencia de mensajes de gran tamaño con las ventajas de las claves públicas. También simplifica el proceso de cifrado al generar un elemento aleatorio en un grupo finito que proporciona la clave simétrica creando un hash del elemento aleatorio.

La seguridad de SABER depende de la dureza del problema del módulo de aprendizaje con redondeo que lo protege contra los ordenadores cuánticos. Este algoritmo está formado por tres niveles que son Light SABER, SABER y FireSABER. Los principales bloques aritméticos consisten en multiplicaciones de polinomios, operaciones bit a bit y operaciones enrejadas. Los módulos de números enteros son cuadráticos por lo que no hace falta utilizar módulos de reducción con operaciones bit a bit.

SABER se constituye de los algoritmos Saber.PKE y Saber.KEM. Saber.PKE es el esquema de cifrado de la clave pública de seguridad IND-CPA. Y a partir de este, usando una versión de la transformada de Fujisaki-Okamoto, surgió Saber. KEM que es el mecanismo de encapsulación de la clave de seguridad IND-CCA en el que tras su generación las claves públicas y privadas se devuelven en dos arrays separados por un byte para que puedan ser utilizadas en las operaciones de cifrado y descifrado. La operación de cifrado coge la clave pública y genera unas claves de sesión y el texto cifrado utilizando la semilla de la clave de sesión y el proceso de descifrado coge el texto cifrado y la clave privada y recupera la clave de sesión que pertenece al texto cifrado.

El algoritmo puede tener fallos como el fallo del descifrado si los errores son demasiado grandes y hay posibles ataques que pueden explotar esta vulnerabilidad. Si el mensaje se reutiliza entonces Saber.PKE es vulnerable a ataques que son adaptivos. También hay ataques que pueden recuperar la clave privada como pueden ser los de timing, electromagnéticos o simples fallos.

Las únicas ventajas que presenta SABER son la fuerza de dos módulos que hacen que la transformación de A2B (A to B) a B2A (B to A) sea más eficiente y que el aprendizaje con redondeo no requiera generación de errores.

Tras haber superado numerosas pruebas, los expertos de NIST lo consideran un algoritmo bastante seguro lo que le ha permitido llegar a la final de la tercera ronda del concurso. Sin embargo, se quedó a las puertas de ganar la ronda que se la acabó llevando CRYSTALS-Kyber.

3.IMPLEMENTACIONES

En este apartado comenzaremos con una breve sección introductoria en la que se hará referencia a los lenguajes/estándares empleados en este trabajo. Después se proporciona una explicación matemática sobre los códigos binarios de Goppa para finalmente centrarnos en la descripción de las distintas implementaciones del algoritmo McEliece y su estructura, destacando la implementación finalmente seleccionada.

3.1. C y OpenACC

El lenguaje de programación empleado en las implementaciones estudiadas es C [15] , el cual fue creado por Brian Kernighan y Dennis Ritchie a mediados de los años 70. La primera implementación de este lenguaje la realizó Dennis Ritchie sobre un computador DEC PDP-11 con sistema operativo UNIX. C es el resultado de un proceso de desarrollo que comenzó con un lenguaje anterior, el BCPL, el cual influyó en el desarrollo por parte de Ken Thompson de un lenguaje llamado B, el cual es el antecedente directo del lenguaje C. El lenguaje C es un lenguaje para programadores en el sentido de que proporciona una gran flexibilidad de programación y una muy baja comprobación de incorrecciones, de forma que el lenguaje deja bajo la responsabilidad del programador acciones que otros lenguajes realizan por sí mismos. El lenguaje C es un lenguaje estructurado, en el mismo sentido que lo son otros lenguajes de programación tales como el lenguaje Pascal, el Ada o el Modula-2, pero no es estructurado por bloques, o sea, no es posible declarar subrutinas (pequeños trozos de programa) dentro de otras subrutinas, a diferencia de como sucede con otros lenguajes estructurados tales como el Pascal. Además, el lenguaje C no es rígido en la comprobación de tipos de datos, permitiendo fácilmente la conversión entre diferentes tipos de datos y la asignación entre tipos de datos diferentes.

Para la paralelización del código empleamos OpenACC [16], que es un estándar de la programación en paralelo en aceleradores sobre todo de NVIDIA. Está diseñado para simplificar la programación de las GPU. Básicamente consiste en insertar simples directivas en el código para que se transfiera esa parte del código y pueda ser paralelizada por la GPU a nivel de sus núcleos. Esto permite a los programadores crear un código muy eficiente en OpenACC simplemente haciendo pequeñas modificaciones en el código de la CPU. Las directivas dicen al compilador que genere un código paralelo para la GPU reserve memoria en la GPU, copie los datos de entrada, ejecute el código paralelo en la GPU, copie los datos de salida hacia la CPU y libere la memoria reservada previamente en la GPU. Las primeras directivas llamadas kernels demandan al compilador que genere el código para la GPU y permite que el compilador elija el mejor modo de paralelizar el código y realizar la transferencia de datos. Podemos ver las ventajas de la paralelización del código en la Figura 2.

Figura 2 : Ventajas de la paralelización del código.

3.2. Códigos binarios de Goppa

En matemáticas y ciencias de la computación el código binario Goppa [17]es un código de corrección de errores que pertenece a la clase de códigos generales Goppa, originalmente descritos por Valerii Denisovich Goppa. La estructura binaria le da varias ventajas matemáticas sobre variantes no binarias, proporcionando también un mejor ajuste para uso común en computadores y telecomunicaciones. Los códigos binarios Goppa tienen propiedades interesantes adecuadas para la criptografía en criptosistemas similares a McEliece.

Un código binario de Goppa está definido por un polinomio g(x) de grado t sobre un campo finito GF(2^{m}) sin varios ceros y una secuencia L de n elementos distintos de GF(2^{m}) que no son raíces del polinomio:

$$\forall\, i,j\ \in \{0,\dots,n-1\}: L_i\ \in GF(2^m)\ \ AND\ \left(L_i = \ L_j \rightarrow i = j\right)\ \ AND\ g(L_i)\ \neq\ 0$$

Las palabras en clave pertenecen al núcleo de la función del síndrome, formando un subespacio de {0,1} ^{n}:

$$\tau(g,L) = \{\, c\ \in\ \{0,1\}^n \mid \sum_{i=0}^{n-1} \frac{c_i}{x - \ L_i} \equiv 0\ \ mod\ g\ (x)\}$$

El código definido por una tupla (g,L) tiene distancia mínima 2t+1, por lo que puede corregir

$[\frac{(2t+1)-1}{2}]$ errores en una palabra de tamaño n-mt usando palabras en clave de tamaño n. También posee una adecuada matriz de verificación de paridad H con la siguiente forma:

$$H = VD = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ L_0^1 & L_1^1 & L_2^1 & \cdots & L_{n-1}^1 \\ L_0^2 & L_1^2 & L_2^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_0^{t-1} & L_1^{t-1} & L_2^{t-1} & \cdots & L_{n-1}^{t-1} \end{pmatrix} \begin{pmatrix} \frac{1}{g(L_0)} & & & & \\ & \frac{1}{g(L_1)} & & & \\ & & \frac{1}{g(L_2)} & & \\ & & & \ddots & \\ & & & & \frac{1}{g(L_{n-1})} \end{pmatrix}$$

Hay que tener en cuenta que esta forma de la matriz de verificación de paridad, que se compone de una matriz de Vandermonde V y una matriz diagonal D, comparte el esquema con matrices de verificación de códigos alternativos, por lo que se pueden utilizar decodificadores alternativos. Dichos decodificadores generalmente proporcionan solo una capacidad limitada de corrección de errores (en la mayoría de los casos t/2).

Para fines prácticos, la matriz de verificación de paridad de un código Goppa binario generalmente se convierte a una forma binaria más amigable para el computador mediante una construcción de seguimiento, que convierte la t-por-n matriz sobre GF(2^{m}) a un mt-por-n matriz binaria escribiendo coeficientes polinomiales de GF(2^{m}) elementos en m filas sucesivas.

La decodificación de códigos binarios Goppa se realiza tradicionalmente mediante el algoritmo de Patterson, que ofrece una buena capacidad de corrección de errores (corrige los t errores de diseño), y también es bastante simple de implementar.

El algoritmo de Patterson convierte un síndrome en un vector de errores. El síndrome de una palabra binaria $c = (c_0, \dots, c_{n-1})$ se espera que tome una forma $s(x) \equiv \sum_{i=0}^{n-1} \frac{ci}{x-Li} \ mod\ g(x)$.

La forma alternativa de una matriz de verificación de paridad basada en la fórmula para s(x) se puede utilizar para producir dicho síndrome con una simple multiplicación de matrices.

El algoritmo luego calcula $v(x) \equiv \sqrt{s(x)^{-1} - x}\ mod\ g(x)$,que falla cuando $s(x) \equiv 0$, pero ese es el caso cuando la palabra de entrada es una palabra de código, por lo que no es necesaria la corrección de errores.

v(x) se reduce a polinomios a(x) y b(x) utilizando el algoritmo euclidiano extendido, de modo que

$a(x) \equiv b(x) * v(x)\ \ mod\ g(x)$, tiempo $\deg(a) \leq [\frac{t}{2}]$.

Finalmente, el polinomio del localizador de errores se calcula como $\sigma\ (x) = a(x)^2 + x * b(x)^2$. Hay que tener en cuenta que, en el caso binario, localizar los errores es suficiente para corregirlos, ya que solo hay otro valor posible. En casos no binarios, también se debe calcular un polinomio de corrección de errores separado.

Si la palabra de código original es decodificable y la $e = (e_0, e_1, \dots, e_{n-1})$ es el vector de error binario, entonces $\sigma(x) = \prod_{i=0}^{n-1} e_i\ (x - L_i)$.

Por lo tanto, factorizar o evaluar todas las raíces de $\sigma(x)$ proporciona suficiente información para recuperar el vector de error y corregir los errores.

Los códigos binarios Goppa vistos como un caso especial de los códigos Goppa tienen la propiedad interesante de que corrigen por completo deg(g) errores, mientras que solo (deg(g) / 2) errores son corregidos en códigos ternarios Goppa y todos los demás casos.

Asintóticamente, esta capacidad de corrección de errores cumple con el famoso límite de Gilbert-Varshamov.

Debido a la alta capacidad de corrección de errores en comparación con la tasa de código y la forma de la matriz de verificación de paridad (que generalmente apenas se distingue de una matriz binaria aleatoria de rango completo), los códigos binarios Goppa se utilizan en varios criptosistemas post-cuánticos, especialmente el criptosistema McEliece y criptosistema Niederreiter.

3.3. Estructura códigos estudiados

Hemos realizado una búsqueda extensa para poder encontrar implementaciones del algoritmo de McEliece con la finalidad de realizar una paralelización sobre una de ellas. En total hemos encontrado ocho implementaciones distintas de las cuales hemos seleccionado tres que son: The-McEliece-Cryptosystem-master [ref], McEliece-master [ref] y node-mceliece-nist-main [ref]. Estas implementaciones fueron seleccionadas por nosotros para compararlas, probarlas y comprobar cuál de ellas es la más rápida y eficiente. Al final elegimos The-McEliece-Cryptosystem-master porque es una implementación con una estructura de código sencilla y fácil de entender y comprender su funcionamiento. Por otra parte, es la implementación cuyo código ofrece mayor número de opciones para su paralelización.

De las tres implementaciones, dos de ellas (The-McEliece-Cryptosystem-master y McEliece-master) tienen una estructura muy similar (véase Figura 3), en la que las funciones principales se encuentran en el archivo “mceliece.c” que se encarga de llamar a las funciones de cifrado y descifrado que se encuentran en los archivos decrypt.c y encrypt.c. Por otro lado, “mceliece.c” llama a las funciones que se encuentran en el archivo “key_gen.c” que se encarga de crear las claves pública y privada. Todos los archivos anteriores dependen del archivo “matrix.c” que contiene todas las operaciones relacionadas con el manejo y creación de matrices.

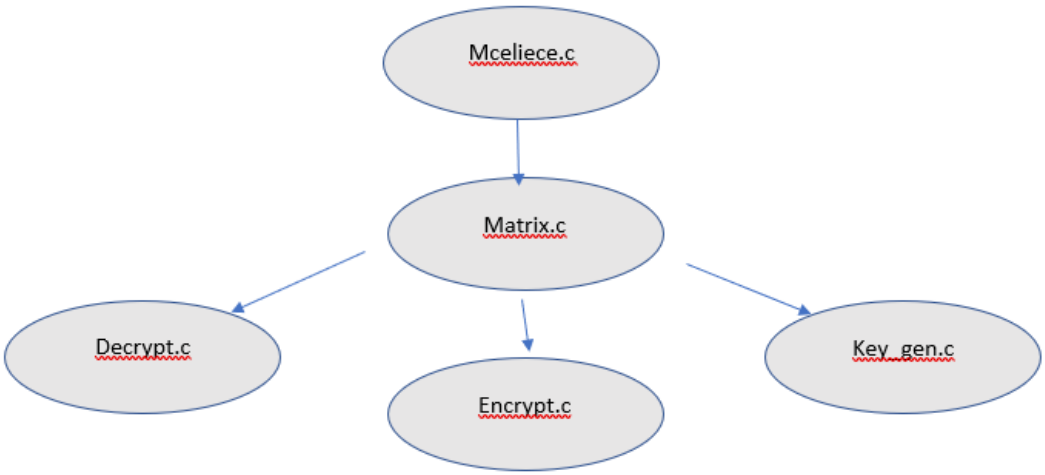

Figura 3:Estructura del código de la primear y segunda implementación.

El fichero matrix.c ofrece las siguientes funciones:

- Inicializar la matriz.
- Devolver el elemento de una posición indicada por los índices.
- Establecer el valor en una posición dada por los índices.
- Establecer la fila indicada de la matriz igual al vector.
- Eliminar la matriz y liberar el espacio en memoria.
- Devolver la matriz traspuesta.
- Copiar los datos de la matriz A a la B.
- Añadir dos matrices.
- Añadir la fila 1 a la fila 2 de la matriz.
- Intercambiar dos filas de la matriz.
- Obtener la fila reducida de Echlon de la matriz.
- Multiplicación de dos matrices A y B que se guardan en C.
- Establecer la matriz como matriz de identidad.
- Comprobar si la matriz es nula o dos matrices son iguales.
- Añadir la fila 1 a la fila 2 en la columna con rango [a,b].
- Añadir la columna 1 a la columna 2 en el rango de la fila [a,b]
- Invertir la matriz.
- Obtener el numero específico de filas y columnas.
- Encontrar la base kernel de la matriz.
- Concatenar dos matrices A y B.
- Imprimir la matriz.

La única diferencia entre estas dos implementaciones es que una de ellas ofrece más automatización a la hora de compilar y ejecutar la implementación teniendo un archivo makefile que se encarga de compilar el resto de los archivos o simplemente compilar cada archivo por separado.

La tercera implementación tiene una desventaja muy importante que es el idioma, se ha utilizado el francés, siendo muy complicado comprender las funciones ya que todos los comentarios y textos que se escriben por pantalla están escritos en este idioma y no se entiende qué acciones realiza cada función y cómo se ejecuta la implementación.

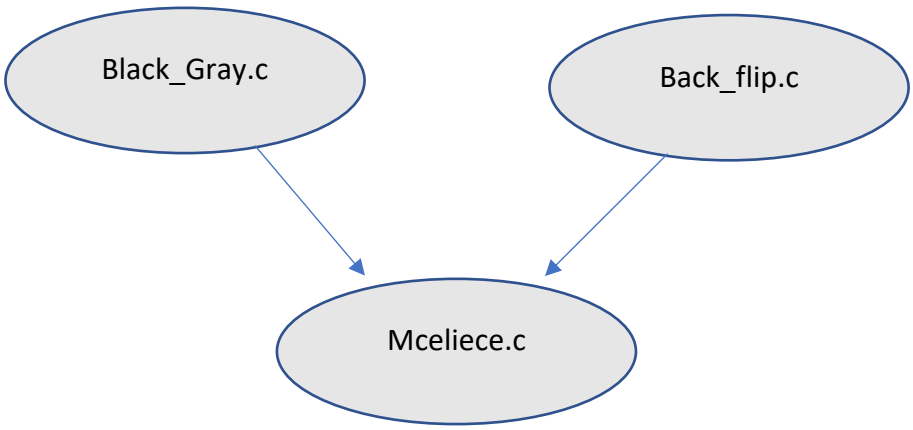

Figura 4 : Estructura del código de la tercera implementación (francesa).

El funcionamiento de la tercera implementación (francesa) es el siguiente (véase Figura 4): Primero hay que introducir dos vectores h0 y h1 que serán mezclados consigo mismos. Después se crea la matriz H, que es la matriz de paridad a partir de los vectores h0 y h1. Posteriormente, se llama al archivo "Black_gray.c" para poder crear la matriz generatriz G utilizando el vector inverso de h1. Al finalizar estos procesos hay que introducir el mensaje que se quiere cifrar y realizar el producto de la matriz G y el mensaje M. Este producto se transforma en un mensaje en formato binario que luego será cifrado y descifrado al invocar las funciones que se encuentran en el archivo "Back_flip.c".

Para poder realizar las ejecuciones de las tres implementaciones hemos utilizado una máquina virtual que tiene un sistema operativo Linux-Ubuntu de 64 bits, con 5GB de memoria RAM, 100GB de espacio de almacenamiento y 20MB de memoria de vídeo. Para poder editar y ejecutar el código se ha utilizado VS Code para Linux. Los mensajes que cifrамos para la comparación de los códigos son de 1000 caracteres.

IMPLEMENTACIÓN DEL ALGORITMO	TIEMPO TOTAL MEDIO EMPLEADO
The-McEliece-Cryptosystem-master (Seleccionada)	85 segundos
McEliece-master (Similar a la seleccionada)	87 segundos
node-mceliece-nist-main (francesa)	95 segundos

Tabla 2: Comparación de los tiempos de ejecución de las implementaciones estudiadas.

La implementación seleccionada finalmente fue The-McEliece-Cryptosystem-master que está formada por un total de 18 archivos. Seleccionamos esta implementación porque al compararla con las otras fue la más eficiente al realizar las pruebas, ofreciendo mejores tiempos de cifrado y descifrado del mensaje con respecto a las otras dos implementaciones. Por otro lado, esta implementación tiene un archivo Makefile que permite compilar todos los archivos con un solo comando. Esto nos permitía ahorrar tiempo a la hora de realizar la compilación de la implementación lo que supone una ventaja importante respecto a las otras implementaciones.

La implementación se puede ejecutar desde dos archivos distintos:

El primer archivo es init.c que es una versión más automática y completa a la hora de ejecutar la implementación porque al ejecutarla exige que se introduzcan los cuatro parámetros: n0, p, w y t. Permitiéndonos hacer la inicialización de la matriz principal. Tras finalizar la implementación, se muestra un menú que ofrece las siguientes opciones:

0) Quit, que permite finalizar la ejecución y terminar el proceso.

1) Keygen, que es la opción que se encarga de generar las claves privada y pública.

2) Encrypt, la cual permite ejecutar la función de cifrado del mensaje que está en formato bit.

3) Decrypt, que se encarga de ejecutar la función de descifrado del texto cifrado y devolver el mensaje original en formato bit.

El segundo archivo es test.c que esta más automatizado que el primer archivo. Para los parámetros n0, p, t y w puedes tener dos opciones que son: introducirlos a mano o definir unos parámetros por defecto en el propio código (así no hace falta estar pensando en parámetros para introducir en cada ejecución). Posteriormente la ejecución exige que se introduzca un mensaje, que será el texto para cifrar y descifrar. Después de introducir el mensaje se invoca una función llamada "text_to_binary" que se encarga de traducir el mensaje, de formato texto a formato binario. Se procede a inicializar los parámetros n0, w, p y t y también se inicializa la matriz binaria. A continuación, se llama a la función "encrypt" que se encarga de cifrar el mensaje que está en formato binario, después se invoca la función "decrypt" que se encarga de descifrar el texto cifrado devolviendo el mensaje original en formato binario y para acabar se invoca la función "binary_to_text" que traduce el mensaje en formato binario a mensaje en formato texto y este se imprimirá por pantalla.

Después de haber compilado y ejecutado la implementación a partir de los dos archivos, se recomienda utilizar el archivo test.c porque es mucho más rápido, eficaz, automatizado y permite jugar con los parámetros y el mensaje a introducir.

3.4. Implementación seleccionada

Después de estudiar distintas implementaciones del algoritmo de Classic McEliece, nos decantamos por la implementación The-Classic-McEliece-master-cryptosystem. Esta implementación permite una mejor compilación ya que esta más automatizada permitiendo ejecutar cada parte del proceso por separado, se obtienen mejores resultados en comparación con las otras implementaciones y la estructura del código resulta más fácil de paralelizar.

Hay dos métodos para poder ejecutar el proceso. El primero se realiza mediante la compilación y ejecución del archivo "init.c", que al ejecutarse pide al usuario que introduzca los parámetros iniciales n0, p, w y t para poder inicializar el proceso invocando a la función "mceliece_init" del fichero "mceliece.c". Esta función realiza la inicialización del sistema de cifrado, reservando primero espacio en memoria y luego invocando a la función "qc_mdpc_init" del fichero "qc_mdpc.c". Esta función se encargaría de generar la matriz de código con los elementos iniciales proporcionados y una semilla aleatoria o elegida por el usuario.

Después invoca la función “generator_matrix” del fichero “qc_mdpc.c”, pasándole la matriz de código obtenida por el método “qc_mdpc_init” para poder crear la matriz generatriz G. Para poder crear esta última es necesario que se hayan creado previamente las matrices H, H_inv, H_0, Q, M e I. H es la matriz de paridad que se crea llamando a la función “parity_check_matrix”. Este método crea dos matrices H y M, usa las funciones “make_matrix” y “splice” para crearlas y luego las concatena horizontalmente mediante la función “concat_horizontal” creando así la matriz de paridad. H_inv se obtiene a través de la matriz de paridad, en este caso como su nombre indica H_inv es la matriz invertida de H creada con el método “circ_matrix_inverse”. H_0 y M son simplemente unas matrices nulas que se crean con las funciones de “make_matrix” y “splice”. Q se obtiene a través de la matriz traspuesta de la multiplicación de H_inv y H_0 usando las funciones de “transpose” y “matrix_mult”. M se transforma haciendo la matriz traspuesta de la multiplicación entre la versión inicial de M y H_inv de la misma manera cuando se crea Q. Q tiene también una versión mejorada que se obtiene al hacer la concatenación vertical entre la versión inicial de Q y la nueva versión de M usando la función “concat_vertical”. I es la matriz de identidad que se crea con los métodos de “mat_init” y “make_identity”. Para terminar, después de generar todas las matrices anteriores, la matriz generatriz G se crea a través de la concatenación horizontal de I y Q. Esta matriz G se guarda como clave publica y así finaliza el proceso de inicialización del sistema de cifrado. Para una mayor comprensión de lo que se ha descrito véase la Figura 5.

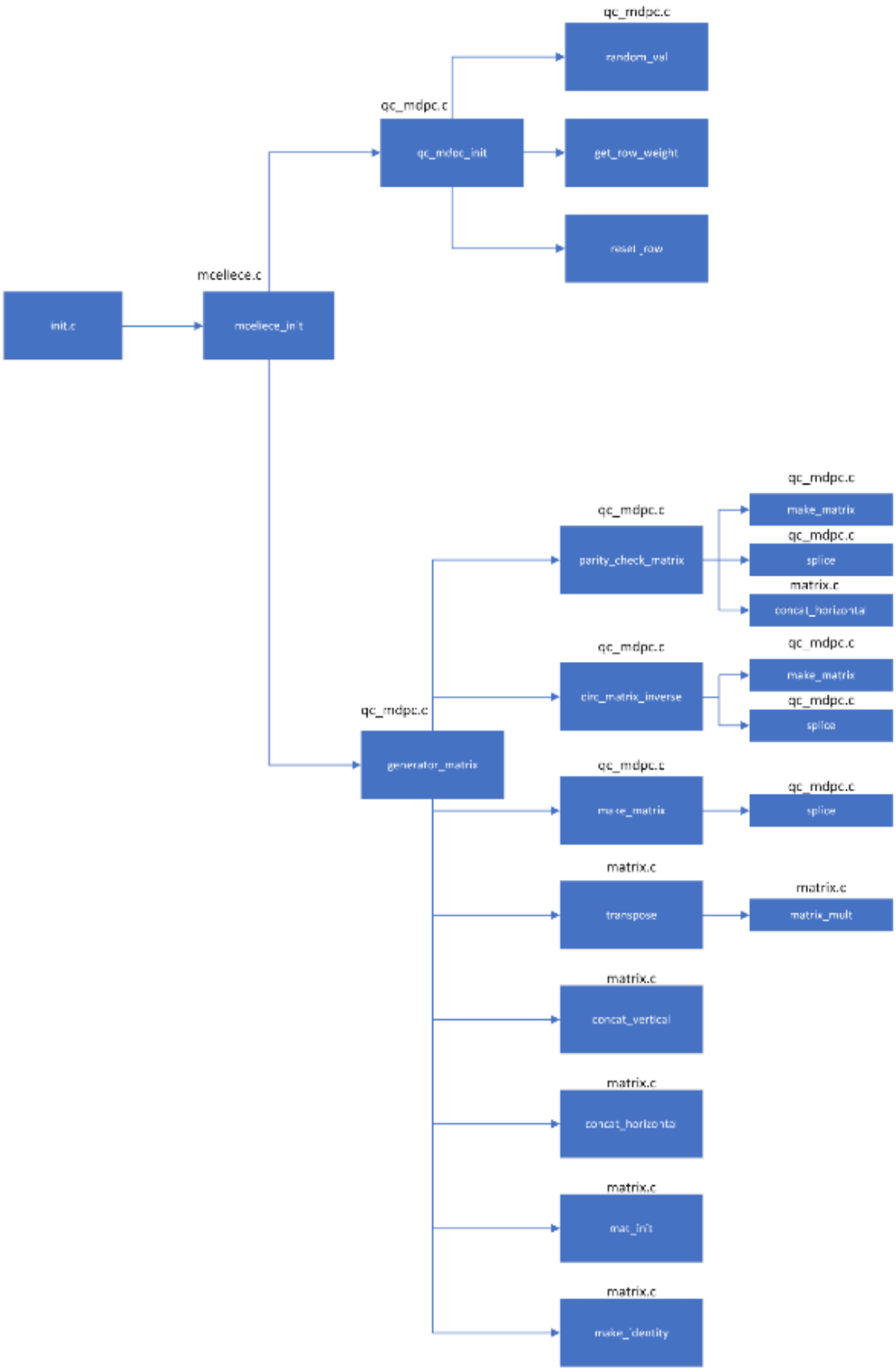

Figura 5: Estructura del código

Después de finalizar con el proceso de inicialización del sistema de cifrado, se mostrará un menú que permite seleccionar entre las siguientes opciones:

- QUIT: Al seleccionar esta opción se ejecutará un break para finalizar el proceso.

```
if(opt == 0)
{
  break;
}
```

Código 1

- KEYGEN: Se crearán las claves pública y privada y se grabarán en unos ficheros txt.
  Para obtener las claves pública y privada se crean dos matrices: la matriz H, que se corresponde con la clave privada y se crea con el método "parity_check_matrix" explicado previamente en la inicialización del sistema de cifrado, y la matriz G, que se corresponde con la clave publica y que se crea mediante el método "generator matrix" también mencionado anteriormente.

  Posteriormente se usa el método "get_matrix_element" para comprobar que los rangos de los elementos son correctos y al mismo tiempo insertarlos en el txt correspondiente y así completar el proceso de generar los ficheros txt de las claves pública y privada. Para una mayor comprensión vease la Figura 6.

```
else if(opt == 1)
        {
            bin_matrix H = parity_check_matrix(crypt->code);
            bin_matrix G = generator_matrix(crypt->code);
            FILE *fp1, *fp2;
            fp1 = fopen("Private_Key.txt", "a");
            fprintf(fp1, "Private Key: Parity Check Matrix: \n");
            for(int i = 0; i < H->rows; i++)
            {
                for(int j = 0; j < H->cols; j++)
                {
                    fprintf(fp1, "%hu ", get_matrix_element(H, i, j));
                }
                fprintf(fp1, "\n \n");
            }
            fclose(fp1);

            fp2 = fopen("Public_Key.txt", "a");
            fprintf(fp2, "Public Key: Generator Matrix: \n");
            for(int i = 0; i < G->rows; i++)
            {
                for(int j = 0; j < G->cols; j++)
                {
                    fprintf(fp2, "%hu ", get_matrix_element(G, i, j));
```

```
                }
                fprintf(fp2, "\n \n");
            }
            fclose(fp2);
            printf("Keys Generated...\n");
        }
```

Código 2

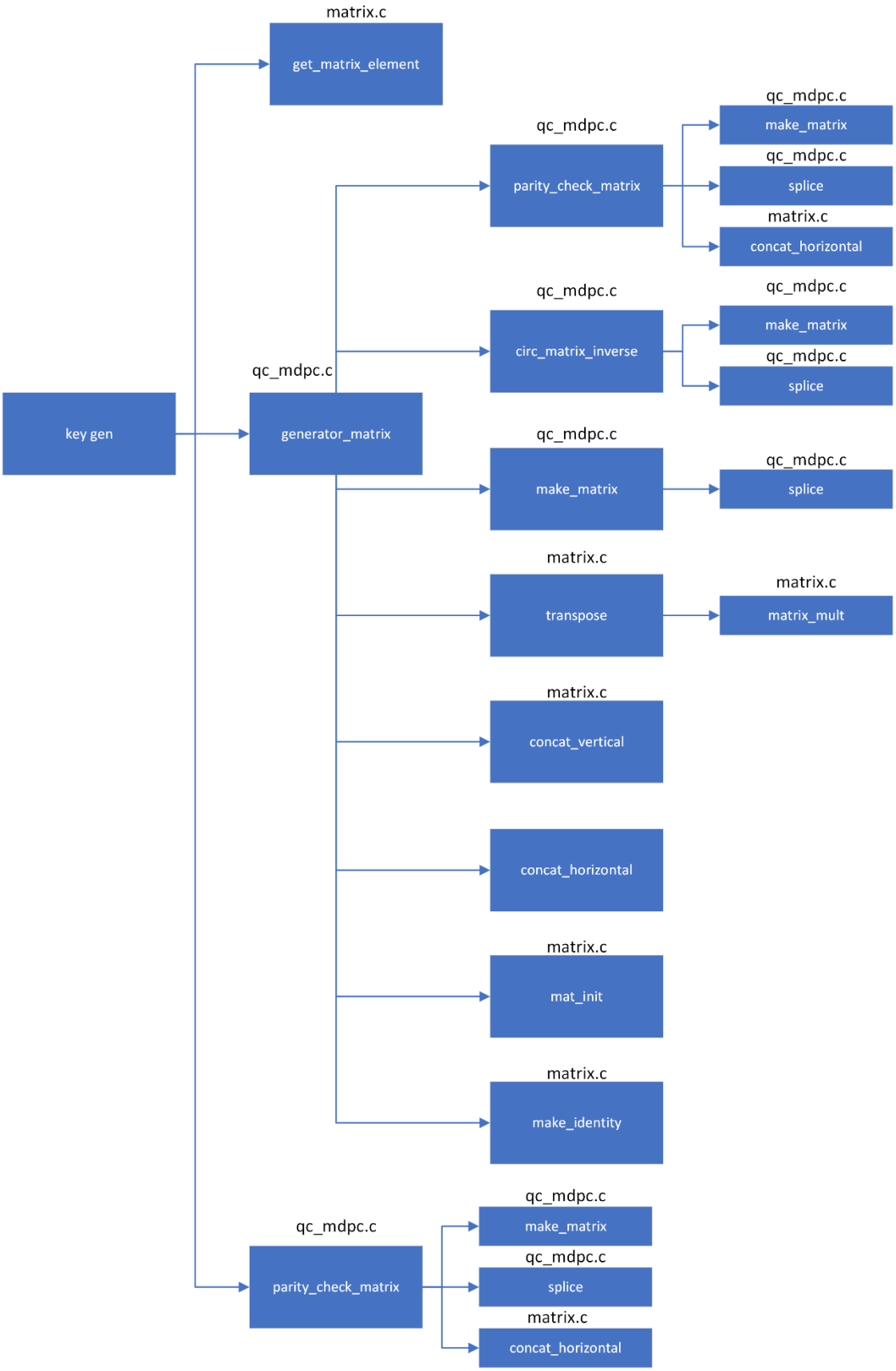

Figura 6: Estructura del código de la opción KEY GEN.

- ENCRYPT: Se realiza el proceso de cifrado de un mensaje introducido por el usuario y el resultado se guardará en un fichero txt. El usuario primero introduce un mensaje de tamaño K que posteriormente se va a insertar en una matriz binaria mediante el método “mat_init” para crear la matriz de tamaño k y el método “set_matrix_element” para insertar el mensaje dentro de la matriz. A continuación, para poder inicializar el proceso de sistema de cifrado se invoca la función “mceliece_init” explicada en la creación del “init.c”.

  Posteriormente se crea una matriz M que llama al método “encrypt” al que se le pasa el sistema de cifrado creado y el mensaje. El método primero comprueba si la longitud del mensaje es correcta y después crea una matriz error invocando al método “get_error_vector” que permite crear una matriz con valores aleatorios de error. Esto se hace primero creando una matriz con el mismo tamaño del mensaje usando la función “mat_init” y luego se procede a crear un valor aleatorio empleando la función “random_val”. Después se comprueba si el elemento aleatorio no está en la matriz usando “get_matrix_element” y si ese elemento no está entonces se inserta en la matriz con el método “set_matrix_element”. Este proceso se repite hasta alcanzar el tamaño del sistema y completar el proceso de cifrado. Después se utilizan los métodos “matrix_mul” para obtener la multiplicación entre la matriz que contiene la clave publica creada previamente y la matriz que contiene el mensaje cuyo resultado será usado por el método “add_matrix” para incorporarlo a la matriz de error.
  El proceso finaliza usando el método “get_matrix_element” para comprobar que los rangos de los elementos están correctos y al mismo tiempo insertarlos en el txt correspondiente. Para una mayor comprensión véase la Figura 7.

```
else if(opt == 2)
        {
            printf("Enter Message of length %d: \n", k);
            unsigned short inp;

            bin_matrix msg = mat_init(1, k);
            for(int i = 0; i < k; i++)
            {
                scanf("%hu", &inp);
                set_matrix_element(msg, 0, i, inp);
            }

            mcc crypt = mceliece_init(n0, p, w, t);
            bin_matrix m = encrypt(msg, crypt);

            FILE *fp1;
            fp1 = fopen("Encryption.txt", "a");
            fprintf(fp1, "Encrypted message: \n");
            for(int i = 0; i < m->cols; i++)
            {
```

```
            fprintf(fp1, "%hu ", get_matrix_element(m, 0, i));
        }
        fclose(fp1);
        printf("Encrypted...\n");
    }
```

Código 3

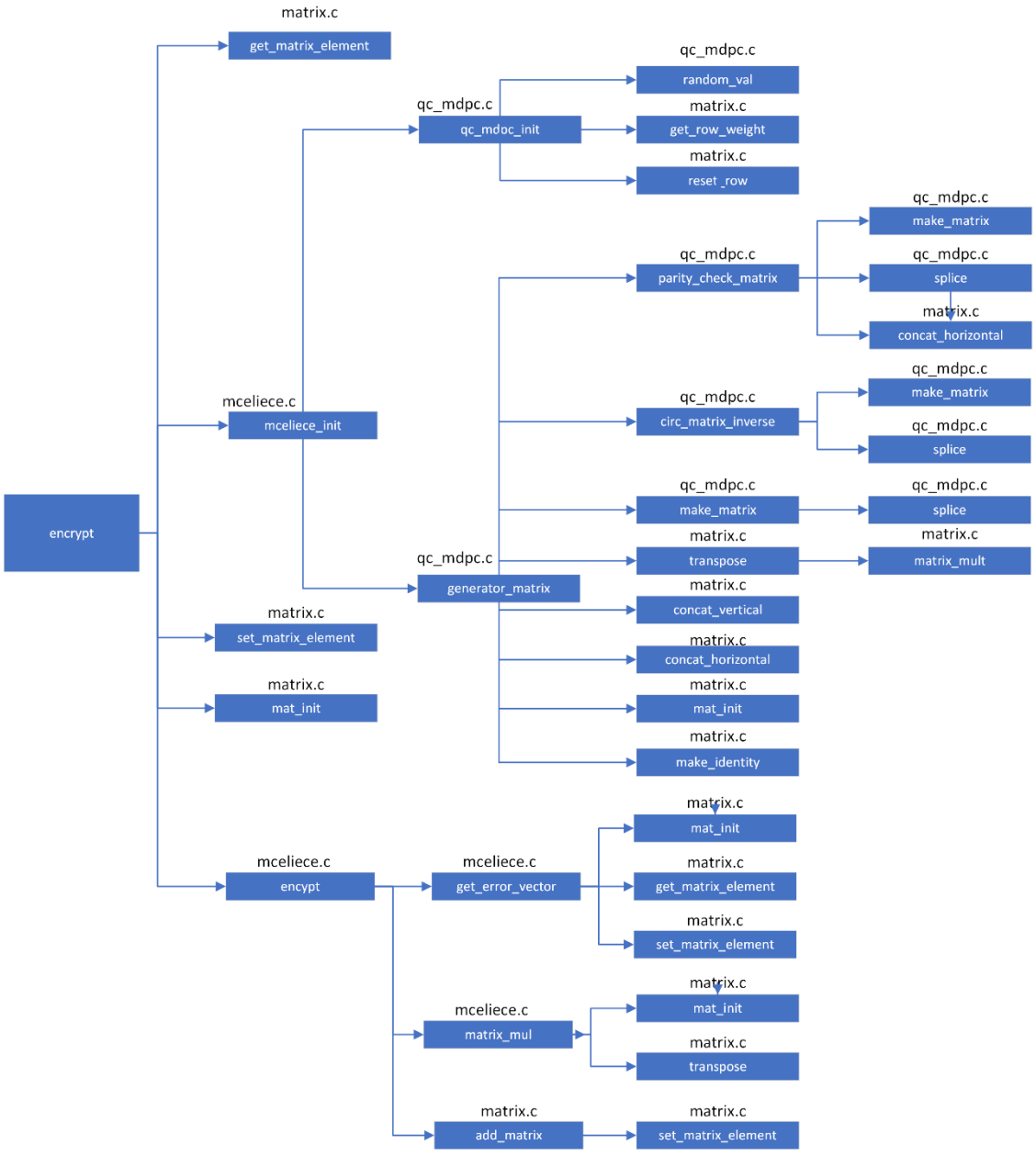

Figura 7: Estructura del código de la opción ENCRYPT.

- DECRYPT: Se realiza el proceso de descifrado de un mensaje introducido por el usuario y el resultado se guardará en un fichero txt. El usuario primero introduce un mensaje de tamaño K que posteriormente se va a insertar en una matriz binaria mediante el método “mat_init” para crear la matriz de tamaño k y el método “set_matrix_element” para insertar el mensaje dentro de la matriz. A continuación, para poder inicializar el proceso de sistema de cifrado se invoca la función “mceliece_init” explicada anteriormente.

  Posteriormente se crea una matriz M que llama al método “decrypt” al que se le pasa el sistema de cifrado creado y el mensaje. El método primero comprueba si la longitud del mensaje es correcta y después crea una matriz msg invocando al método “decode” que se encargaría de decodificar el mensaje cifrado. Para ello, primero crea la matriz H (la matriz de paridad) utilizando la función “parity_check_matrix” del fichero “qc_mdpc.c”, empleando la matriz de código. Además, se usan los métodos “splice” y “make_matrix” del fichero “qc_mdpc.c”, y “concat_horizontal” del fichero “matrix.c” para crear una matriz M y posteriormente concatenarla horizontalmente con la matriz H. A continuación, se crea una matriz syn, creando para ello una matriz traspuesta con la función “transpose” y luego multiplicándola con la matriz H usando el método “matrix_mul”. Posteriormente, en un doble bucle for se comprueba qué elementos de las matrices H y syn están a 1 para marcarlos dentro del vector *unsitesfied* empleando la función “get_matrix_element”. Luego se usa la función “get_max” para obtener el máximo elemento del vector *unsitesfied* que será utilizado como límite menor a la hora de comprobar el resto de los elementos del vector para poder reordenar la matriz inicial usando “get_matrix_element” y “set_matrix_element”. Finalmente, se usa el método “add_matrix” para añadir la parte divida de la matriz H por el método “mat_splice” a la matriz syn y así concluiría el proceso de decodificación del mensaje cifrado. Para terminar con el proceso del descifrado se llama a la función “mat_splice” para dividir la matriz msg en un número determinado de filas y columnas. Después se usa el método “get_matrix_element” para comprobar que los rangos de los elementos están correctos y al mismo tiempo insertarlos en el txt correspondiente. Para una mayor comprensión véase la Figura 8.

```
else if(opt == 3)
        {
            printf("Enter code of length %d: ", k);
            unsigned short inp;
            bin_matrix msg = mat_init(1, k);
            for(int i = 0; i < k; i++)
            {
                scanf("%hu", &inp);
                set_matrix_element(msg, 0, i, inp);
            }
            mcc crypt = mceliece_init(n0, p, w, t);
```

```
            bin_matrix m = decrypt(msg, crypt);

            FILE *fp1;
            fp1 = fopen("Decryption.txt", "a");
            fprintf(fp1, "Decrypted message: \n");
            for(int i = 0; i < m->cols; i++)
            {
                fprintf(fp1, "%hu ", get_matrix_element(m, 0, i));
            }
            fclose(fp1);
            printf("Decrypted...\n");
        }
```

Código 4

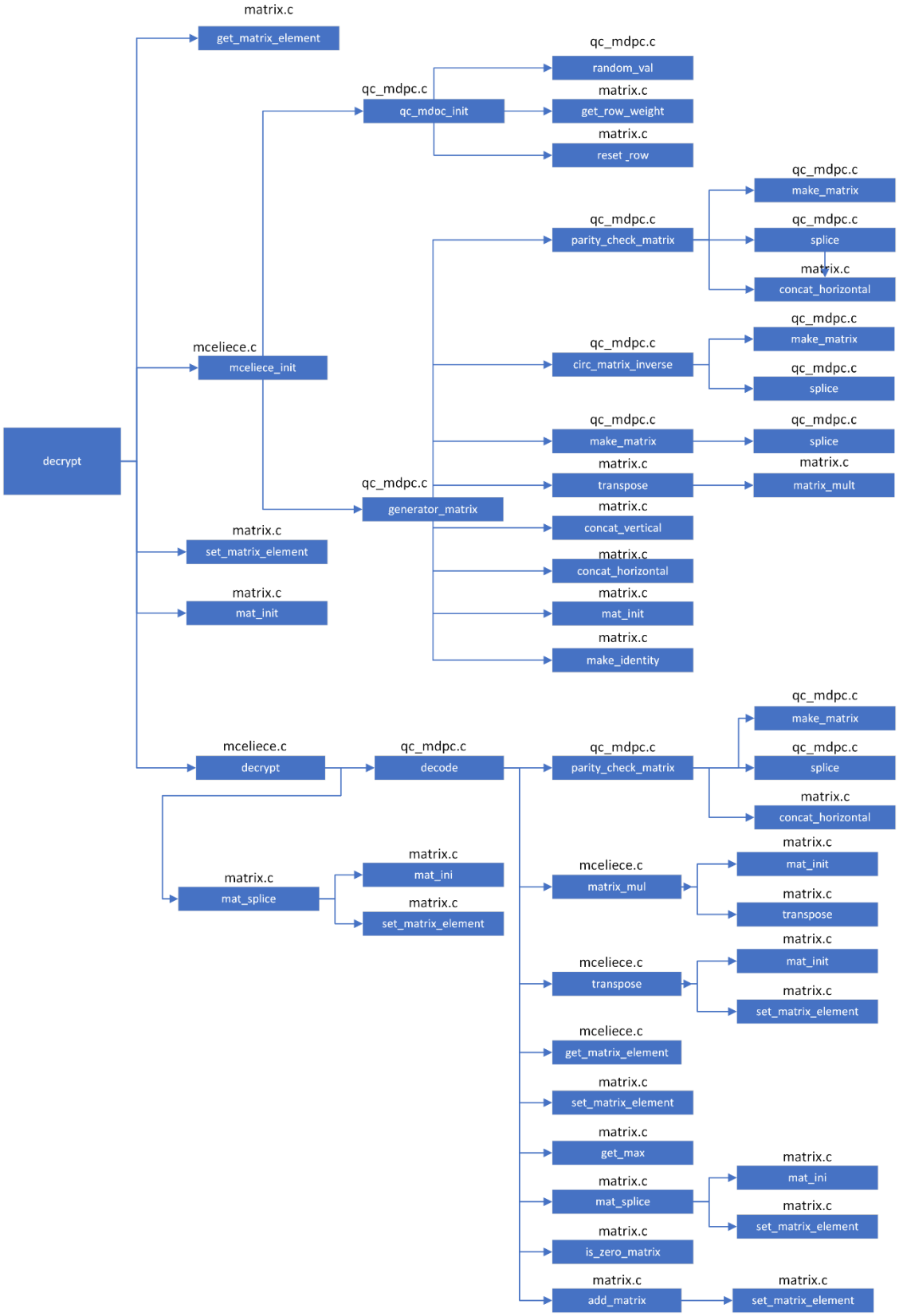

Figura 8: Estructura del código de la opción DECRYPT.

El segundo método que se utiliza para ejecutar el proceso es “test.c”. Este método es el principal a la hora de realizar todas las pruebas necesarias y ejecutar el proceso.

Esta versión es diferente a la anterior ya que es un proceso de ejecución directa, es decir, no tiene un menú para seleccionar los procesos, sino que todo se ejecuta de forma directa. También hay modificaciones realizadas por nosotros también respecto a la versión original ya que se han añadido dos nuevos métodos que permiten traducir texto de caracteres a cadena de números binarios y viceversa y otro método más que permite imprimir la cadena de texto en forma de matriz por pantalla. La estructura de la ejecución también ha sido modificada cambiando la primera parte de la ejecución, exigiendo ahora al usuario, que introduzca un mensaje de caracteres y no uno en binario. El resto de la estructura se mantiene más o menos igual excepto algunos elementos puntuales.

En este método los parámetros iniciales n0, t, p y w ya vienen definidos dentro del código, pero el código se puede modificar para que el usuario pueda introducirlos de forma manual o modificarlos dentro del propio código antes de compilarlo y ejecutarlo. Cuando se ejecute el código se le exigirá al usuario que introduzca un mensaje en formato de texto de cualquier longitud, luego este mensaje será procesado y traducido a una cadena de números binarios por la función “text_to_binary”. A continuación, para poder inicializar el proceso de sistema de cifrado se invoca la función “mceliece_init” al igual que en “init.c”.

Después se crea una matriz llamada msg y se inicializa con la función “mat_init” dándole un tamaño k. El mensaje binario será introducido dentro de la matriz msg utilizando la función “set_matrix_element” y al finalizar el proceso se invoca la función “print_matrix” para imprimir la matriz por pantalla.

Ahora se crean las matrices v y s. Se utiliza el método “encrypt” para inicializar la matriz v, explicado en la opción de menú ENCRYPT de “init.c”, para poder realizar el cifrado sobre el mensaje. Al completar el proceso de cifrado el mensaje cifrado (matriz v) se imprime por pantalla. Se utiliza el método “decrypt” para inicializar la matriz s, explicado en la opción de menú DECRYPT de “init.c”, para poder realizar la operación de descifrado sobre el mensaje cifrado contenido en la matriz v. Al finalizar el proceso de descifrado del mensaje el resultado (matriz s) se imprime por pantalla.

Se invocan los métodos “binary_to_text” para traducir el mensaje de una cadena de números binarios a una cadena de texto y “print_matrix_char” para imprimir el mensaje en cadena de texto por pantalla. El proceso finaliza con una comprobación invocando al método “mat_is_equal” que indica si el proceso ha tenido éxito o ha fallado y luego se libera el espacio de memoria reservado para el proceso con la función “delete_mceliece”. Para una mayor comprensión véase la Figura 9.

```
int main(int argc, char const *argv[])
{
```

```
int n0 = 2;
int t = 20;
int p = 1000;
int w = 45;

char input1[1000];
char output1[8000];

char input2[8000];
char output2[1000];

printf("Introduzca mensjae:\n");
if(fgets(input1, 1000, stdin) == NULL) {
    printf("Error: Input not read successfully!\n");
    return -1;
}
printf("%s\n",input1);

text_to_binary(input1,output1);

printf("%s\n",output1);
printf("\n");

int i,cont;
cont = 0;
unsigned short vi;
for(i = 0; i < strlen(output1);i++){
    vi = output1[i]-'0';
    cont++;
    printf("%d ",vi);
}

printf("\n");
p = cont;

printf("Starting Encryption...\n");
clock_t start, end;
double cpu_time_used;
start = clock();
mcc crypt = mceliece_init(n0, p, w, t);
bin_matrix msg = mat_init(1, crypt->code->k);




unsigned short num;
printf("Mensaje a introducir de longitud %i:\n",crypt->code->k);
for(int i = 0; i < crypt->code->k; i++)
```

```
    {
        num = output1[i]-'0';
        set_matrix_element(msg, 0, i, num);
    }

    printf("Mensaje:\n");
    print_matrix(msg);
    bin_matrix v = encrypt(msg, crypt);
    printf("Cifrado:\n");
    print_matrix(v);
    bin_matrix s = decrypt(v, crypt);
    printf("Descifrado:\n");
    print_matrix(s);
    printf("\n");
    print_matrix_char(s,input2);
    binary_to_text(input2,output2);
    printf("Mensaje despues de descifrar: %s\n",output2);



    if(mat_is_equal(msg, s))
    {
        end = clock();
        printf("Decryption successful...\n");
        cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
        printf("Time taken by cryptosystem: %f\n", cpu_time_used);
    }
    else
    {
        printf("Failure....\n");
    }
    delete_mceliece(crypt);
    return 0;
}
```

Código 5

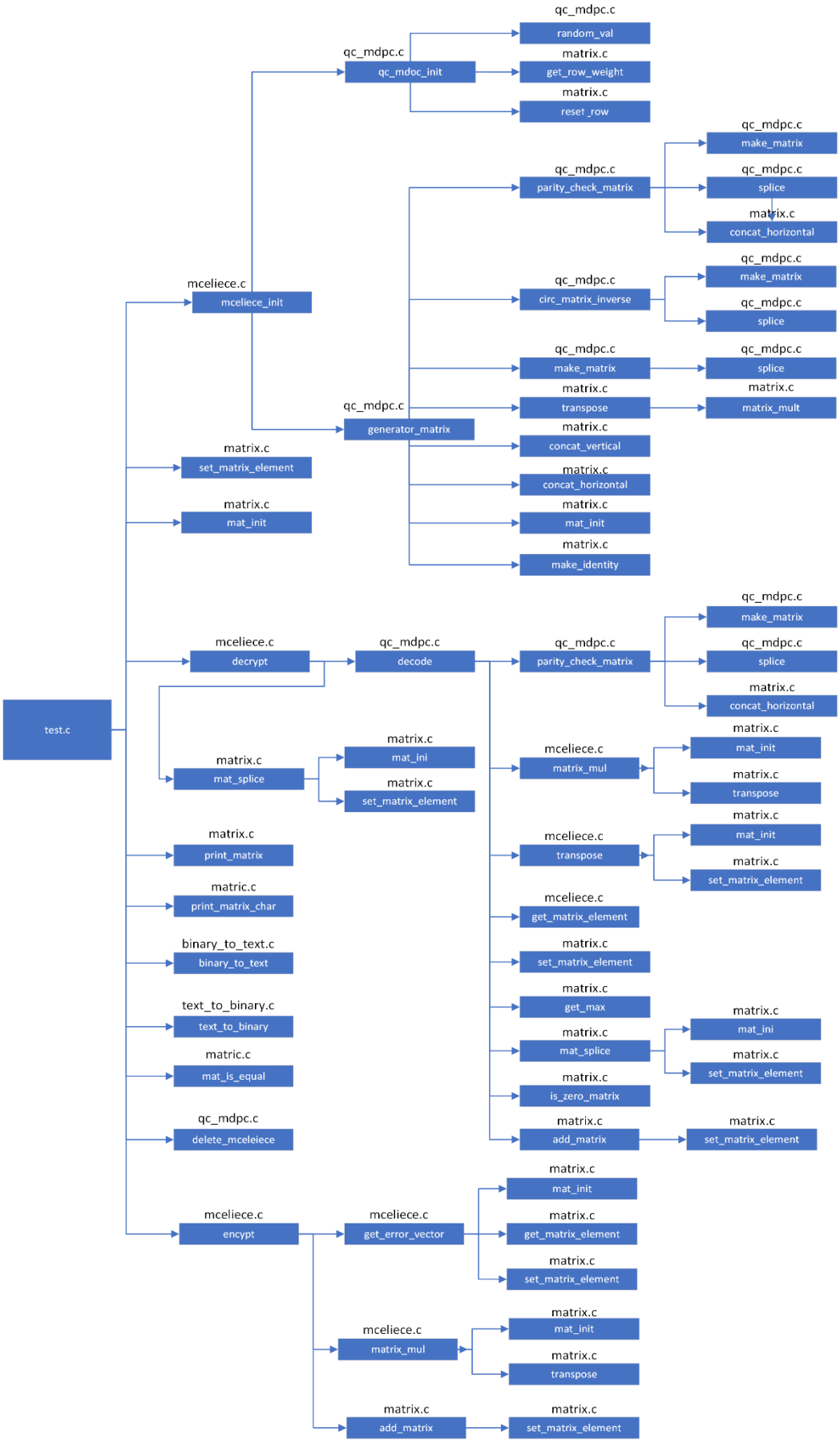

Figura 9: Estructura del código test.c..

Después de explicar las dos formas de realizar el proceso mediante las llamadas a los distintos métodos o funciones, a continuación, vamos a describir en detalle los distintos archivos de la implementación. Se procederá a analizarlos indicando las funciones que tiene cada archivo y cuál es la finalidad con la que se utilizan durante la ejecución de la implementación. Podemos ver la estructura del proyecto en la tabla de abajo.

BinaryText.c	Traducen/convierten el mensaje en formato binario
BinaryText.h	
BinaryText.o	
Makefile	Compila el programa
TextBinary	Traducen/converten el mensaje en formato texto
TextBinary.c	
TextBinary.h	
TextBinary.o	
decrypt.c	Descifra texto
encrypt.c	Cifra texto
init.c	Modo de ejecución que toma parámetros por consola y que **no** cuenta con los traductores de binario a texto y de texto a binario
key_gen.c	Crea claves públicas y privadas
matrix.c	Contienen funciones para inicializar matrices y realizar operaciones con ellas
matrix.h	
matrix.o	
mceliece.c	Inicializan parámetros y generan vectores
mceliece.h	
mceliece.o	
qc_mdpc.c	Funciones con numerosas operaciones
qc_mdpc.h	
qc_mdpc.o	
test.c	Modo de ejecución que toma parámetros predefinidos y que cuenta con los traductores de binario a texto y de texto a binario
utility.c	Aseguran uso correcto memoria
utility.h	
utility.o	

Tabla 3

3.4.1. BinaryText.h y BinaryText.c

Este archivo contiene el código que se encarga de traducir/convertir el mensaje en formato binario que se le pasa por parámetro. Básicamente realiza una conversión de números binarios que son ceros y unos a caracteres.

```
#include <stdio.h>
#include <string.h>
```

```
#ifndef _BINARYTEXT_H
#define _BINARYTEXT_H

void binary_to_text(char* input, char* output);

#endif
```

Código 6

```
#include "BinaryText.h"

void binary_to_text(char* input, char* output) {
    int len = strlen(input), i, j;
    unsigned char c;

    for (i = 0, j = 0; i < len; i += 8, j++) {
        c = (input[i] - '0') * 128 + (input[i + 1] - '0') * 64 + (input[i
+ 2] - '0') * 32 +
            (input[i + 3] - '0') * 16 + (input[i + 4] - '0') * 8 +
(input[i + 5] - '0') * 4 +
            (input[i + 6] - '0') * 2 + (input[i + 7] - '0') * 1;
        output[j] = c;
    }
    output[j] = '\0';
}
```

Código 7

3.4.2. TextBinary.h y TextBinary.c

Este archivo contiene el código que se encarga de traducir/convertir el mensaje en formato texto que se le pasa por parámetro. Básicamente realiza una conversión de caracteres a números binarios.

```
#include <stdio.h>
#include <string.h>

#ifndef _TEXTBINARY_H
#define _TEXTBINARY_H

void text_to_binary(char* input, char* output);

#endif
```

Código 8

```
#include "TextBinary.h"
#include <stdio.h>
#include <string.h>

void text_to_binary(char* input, char* output) {
    int len = strlen(input), i, j;
```

```
    for (i = 0, j = 0; i < len; i++, j += 8) {
        unsigned char c = input[i];
        output[j] = (c & 128) ? '1' : '0';
        output[j + 1] = (c & 64) ? '1' : '0';
        output[j + 2] = (c & 32) ? '1' : '0';
        output[j + 3] = (c & 16) ? '1' : '0';
        output[j + 4] = (c & 8) ? '1' : '0';
        output[j + 5] = (c & 4) ? '1' : '0';
        output[j + 6] = (c & 2) ? '1' : '0';
        output[j + 7] = (c & 1) ? '1' : '0';
    }
    output[j] = '\0';
}
```

Código 9

3.4.3. Decrypt.c

Es un archivo que permite introducir los parámetros n0, p, t y w por consola. A continuación, procede a inicializarlos llamando a la función “mceliece_init”, luego invoca la función decrypt que se encarga de descifrar el texto cifrado devolviendo el mensaje original en formato binario.

```
#include "qc_mdpc.h"
#include "matrix.h"
#include "mceliece.h"
#include <stdlib.h>
#include <stdio.h>


int main(int argc, char const *argv[])
{
    int n0, p, w, t;
    printf("Enter n0: ");
    scanf("%d", &n0);
    printf("Enter p: ");
    scanf("%d", &p);
    printf("Enter w: ");
    scanf("%d", &w);
    printf("Enter t: ");
    scanf("%d", &t);
    int k = (n0 - 1) * p;
    printf("Enter code of length %d: ", k);
    unsigned short inp;
    bin_matrix msg = mat_init(1, k);
    for(int i = 0; i < k; i++)
    {
        scanf("%hu", &inp);
        set_matrix_element(msg, 0, i, inp);
```

```
    }
    mcc crypt = mceliece_init(n0, p, w, t);
    bin_matrix m = decrypt(msg, crypt);

    FILE *fp1;
    fp1 = fopen("Decryption.txt", "a");
    fprintf(fp1, "Decrypted message: \n");
    for(int i = 0; i < m->cols; i++)
    {
        fprintf(fp1, "%hu ", get_matrix_element(m, 0, i));
    }
    fclose(fp1);

    return 0;
}
```

Código 10

3.4.4. Encrypt.c

Es un archivo que permite introducir los parámetros n0, p, t y w por consola. A continuación, procede a inicializarlos llamando a la función “mceliece_init”, luego invoca la función encrypt que se encarga de cifrar el mensaje que se le pasa en formato binario y procede a devolver el texto cifrado.

```
#include "qc_mdpc.h"
#include "matrix.h"
#include "mceliece.h"
#include <stdlib.h>
#include <stdio.h>


int main(int argc, char const *argv[])
{
    int n0, p, w, t;
    printf("Enter n0: ");
    scanf("%d", &n0);
    printf("Enter p: ");
    scanf("%d", &p);
    printf("Enter w: ");
    scanf("%d", &w);
    printf("Enter t: ");
    scanf("%d", &t);
    int k = (n0 - 1) * p;
    printf("Enter Message of length %d: \n", k);
    unsigned short inp;

    bin_matrix msg = mat_init(1, k);
    for(int i = 0; i < k; i++)
    {
```

```
        scanf("%hu", &inp);
        set_matrix_element(msg, 0, i, inp);
    }

    mcc crypt = mceliece_init(n0, p, w, t);
    bin_matrix m = encrypt(msg, crypt);

    FILE *fp1;
    fp1 = fopen("Encryption.txt", "a");
    fprintf(fp1, "Encrypted message: \n");
    for(int i = 0; i < m->cols; i++)
    {
        fprintf(fp1, "%hu ", get_matrix_element(m, 0, i));
    }
    fclose(fp1);

    return 0;
}
```

Código 11

3.4.5. key_gen.c

Este archivo se encarga de crear las claves pública y privada. Primero hay que introducir los parámetros n0, w, t y p, luego se generan las matrices H y G después de inicializar los parámetros invocando la función “qc_mdpc_init”. Después se crean dos archivos uno que guardará la clave privada que se genera usando la matriz H y el otro archivo guardará la clave pública que se genera utilizando la matriz G.

```
#include "qc_mdpc.h"
#include "matrix.h"
#include <stdlib.h>
#include <stdio.h>


int main(int argc, char const *argv[])
{
    int n0, p, w, t;
    printf("Enter n0: ");
    scanf("%d", &n0);
    printf("Enter p: ");
    scanf("%d", &p);
    printf("Enter w: ");
    scanf("%d", &w);
    printf("Enter t: ");
    scanf("%d", &t);
    mdpc code = qc_mdpc_init(n0, p, w, t);
    bin_matrix H = parity_check_matrix(code);

    bin_matrix G = generator_matrix(code);
```

```
    FILE *fp1, *fp2;
    fp1 = fopen("Private_Key.txt", "a");
    fprintf(fp1, "Private Key: Parity Check Matrix: \n");
    for(int i = 0; i < H->rows; i++)
    {
        for(int j = 0; j < H->cols; j++)
        {
            fprintf(fp1, "%hu ", get_matrix_element(H, i, j));
        }
        fprintf(fp1, "\n \n");
    }
    fclose(fp1);

    fp2 = fopen("Public_Key.txt", "a");
    fprintf(fp2, "Public Key: Generator Matrix: \n");
    for(int i = 0; i < G->rows; i++)
    {
        for(int j = 0; j < G->cols; j++)
        {
            fprintf(fp2, "%hu ", get_matrix_element(G, i, j));
        }
        fprintf(fp2, "\n \n");
    }
    fclose(fp2);

    return 0;
}
```

Código 12

3.4.6. matrix.h y matrix.c

Este archivo contiene la mayor parte de funciones que se invocan para inicializar parámetros, calcular operaciones relacionadas con matrices e inicializar las matrices binarias. Las funciones que contiene este archivo son:

```
#include <stdbool.h>
#include "utility.h"

#ifndef _MATRIX_H
#define _MATRIX_H

typedef struct matrix
{
   int rows;             //number of rows.
   int cols;             //number of columns.
   unsigned short *data;
}*bin_matrix;
```

```
bin_matrix mat_init(int rows, int cols);
unsigned short get_matrix_element(bin_matrix mat, int row_idx, int
col_idx);
void set_matrix_element(bin_matrix A, int row_idx, int col_idx, unsigned
short val);
void set_matrix_row(bin_matrix A, int row, unsigned short* vec);
void delete_matrix(bin_matrix A);
bin_matrix transpose(bin_matrix A);
bin_matrix mat_copy(bin_matrix A);
bin_matrix add_rows(bin_matrix A,int row1, int row2);
bin_matrix add_rows_new(bin_matrix A,int row1, int row2, int i1, int i2);
bin_matrix add_cols(bin_matrix A,int col1, int col2, int a, int b);
bin_matrix add_matrix(bin_matrix A, bin_matrix B);
void swap(bin_matrix A, int row1, int row2);
bin_matrix matrix_rref(bin_matrix A);
bin_matrix matrix_mult(bin_matrix A, bin_matrix B);
void make_identity(bin_matrix A);
bool is_identity(bin_matrix A);
int is_zero_matrix(bin_matrix A);
int mat_is_equal(bin_matrix A, bin_matrix B);
bin_matrix matrix_inverse(bin_matrix A);
bin_matrix circ_matrix_inverse(bin_matrix A);
bin_matrix mat_splice(bin_matrix A, int row1, int row2, int col1, int
col2);
bin_matrix mat_kernel(bin_matrix A);
bin_matrix concat_horizontal(bin_matrix A, bin_matrix B);
bin_matrix concat_vertical(bin_matrix A, bin_matrix B);
void print_matrix(bin_matrix A);
void print_matrix_char(bin_matrix A, char* output);

#endif
```

Código 13

- *Mat_init*: esta función se encarga de inicializar la matriz binaria recibiendo como parámetros las filas y columnas de la matriz, y devuelve una matriz binaria.

```
bin_matrix mat_init(int rows, int cols)
{
  if(rows <= 0 || cols <= 0)
  {
    return NULL;
  }
  bin_matrix A;
  A = (bin_matrix)safe_malloc(sizeof(struct matrix));
  A->cols = cols;
  A->rows = rows;
  A->data = (unsigned short *)safe_malloc(rows*cols*sizeof(unsigned
short));
  return A;
```

```
}
```

Código 14

- *Get_matrix_element*: esta función recibe como parámetros una matriz binaria, un índice de fila y un índice de columna. Se encarga de devolver un elemento de la matriz dado por las posiciones de la fila y columna pasadas previamente por parámetros.

```
unsigned short get_matrix_element(bin_matrix mat, int row_idx, int
col_idx)
{
  if(row_idx < 0 || row_idx >= mat->rows || col_idx < 0 || col_idx >=
mat->cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  return mat->data[row_idx * (mat->cols) + col_idx];
}
```

Código 15

- *Set_matrix_element*: esta función recibe por parámetros una matriz binaria, un índice de posición de fila, un índice de posición de columna y un valor. Se encarga de insertar el valor pasado por parámetro dentro de la matriz binaria empleando los índices de posición de la fila y la columna.

```
void set_matrix_element(bin_matrix A, int row_idx, int col_idx,
unsigned short val)
{
  if(row_idx < 0 || row_idx >= A->rows || col_idx < 0 || col_idx >= A-
>cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  mat_element(A, row_idx, col_idx) = val;
}
```

Código 16

- *Set_matrix_row*: esta función recibe por parámetros una matriz binaria, una fila y un vector. Procede a insertar el vector en la fila de la matriz binaria. El bucle se puede paralelizar perfectamente porque simplemente se están colocando los elementos en una matriz A y estos elementos se pueden ejecutar en hilos independientes a la vez.

```
void set_matrix_row(bin_matrix A, int row, unsigned short* vec)
{
  if(row < 0 || row >= A->rows)
  {
    printf("Row index out of range\n");
```

```
    exit(0);
  }

  for(int i = 0; i < A->cols; i++)
  {
    set_matrix_element(A, row, i, vec[i]);
  }
}
```

Código 17

- *Delete_matrix*: esta función recibe por parámetros una matriz binaria. Procede a borrar la matriz binaria y liberar el espacio en memoria que ocupaba la matriz.

```
void delete_matrix(bin_matrix A)
{
  free(A);
}
```

Código 18

- *Transpose*: esta función recibe por parámetros una matriz binaria, procede a devolver la matriz traspuesta de la matriz binaria. Esta función se puede paralelizar porque los elementos de la matriz A son independientes entre sí con lo cual el doble bucle se puede colapsar en uno y ejecutar hilos independientes al mismo tiempo para realizar la misma operación sobre todos los elementos a la vez. Pero hay que tener cuidado porque el bucle interno tiene que ser secuencial respecto al primer bucle para poder seguir la cadena de ejecución de los datos.

```
bin_matrix transpose(bin_matrix A)
{
  bin_matrix B;
  B = mat_init(A->cols, A->rows);

  for(int i = 0; i < A->rows; i++)
  {

    for(int j = 0; j < A->cols; j++)
    {
      set_matrix_element(B, j, i, mat_element(A, i, j));
    }
  }
  return B;
}
```

Código 19

- *Mat_copy*: esta función recibe por parámetros una matriz binaria. Procede a copiar los datos de esa matriz en otra matriz binaria que posteriormente la función devuelve.

```
bin_matrix mat_copy(bin_matrix A)
```

```
{
  bin_matrix B;
  int i;

  B = mat_init(A->rows, A->cols);
  memcpy(B->data, A->data, (A->rows)*(A->cols)*(sizeof(unsigned
short)));
  return B;
}
```

Código 20

- *Add_rows*: esta función recibe por parámetros una matriz binaria y dos filas. Se encarga de insertar una fila dentro de otra fila de la matriz binaria y procede a devolverla al finalizar la operación. Para paralelizar la función habría que seguir el mismo concepto que en la función "set_matrix_row".

```
bin_matrix add_rows(bin_matrix A,int row1, int row2)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->rows)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }

  for(int i = 0; i < A->cols; i++)
  {
    mat_element(A, row2, i) = (mat_element(A, row1, i) ^
mat_element(A, row2, i));
  }
  return A;
}
```

Código 21

- *Add_matrix*: esta función recibe por parámetros dos matrices binarias, procede a insertar una de las matrices dentro de la otra y devuelve el resultado. Habría que seguir el mismo esquema de paralelización que en la función "transpose".

```
bin_matrix add_matrix(bin_matrix A, bin_matrix B)
{
  if(A->rows != B->rows || A->cols != B->cols)
  {
    printf("Incompatible dimenions for matrix addition.\n");
    exit(0);
  }
  bin_matrix temp = mat_init(A->rows, A->cols);

  for(int i = 0; i < A->rows; i++)
  {

    for(int j = 0; j < A->cols; j++)
```

```
    {
      set_matrix_element(temp, i, j, (mat_element(A, i, j) ^
mat_element(B, i, j)));
    }
  }
  return temp;
}
```

Código 22

- *Swap*: esta función recibe por parámetros una matriz binaria y dos filas. Se encarga de intercambiar dos filas de una matriz y la devuelve. Para poder paralelizar esta función habría que aplicar el mismo método utilizado en la función "transpose".

```
void swap(bin_matrix A, int row1, int row2)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->rows)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  int temp;
  for(int i = 0; i < A->cols; i++)
  {
    temp = mat_element(A, row1, i);
    mat_element(A, row1, i) = mat_element(A, row2, i);
    mat_element(A, row2, i) = temp;
  }
}
```

Código 23

- *Matrix_rref*: esta función recibe por parámetros una matriz binaria. La función se encarga de obtener una versión echlon reducida de la matriz binaria y devolver ese resultado. Para paralelizar la función habría que seguir el mismo concepto que en la función "set_matrix_row".

```
bin_matrix matrix_rref(bin_matrix A)
{
  int lead = 0;
  int row_count = A->rows;
  int col_count = A->cols;
  bin_matrix temp = mat_init(row_count, col_count);
  temp = mat_copy(A);

  int r = 0;
  bool return_flag=false;
  for(r=0;r<row_count;r++)
  {
    if(mat_element(temp, r, r) == 0)
    {
```

```
      int i;

      for(i = r + 1; i < temp->rows; i++)
      {
        if(mat_element(temp, i, r) == 1 && return_flag==false)
        {
          swap(temp, r, i);
          return_flag=true;
        }
      }
      if(i == row_count)
      {
        printf("Matix cannot be transformed into row echlon form...");
        exit(1);
      }
    }
    else
    {

      for(int i = 0; i < row_count; i++)
      {
        if(mat_element(temp, i, r) == 1 && i != r)
        {
          add_rows(temp, r, i);
        }
      }
    }
  }
  return temp;
}
```

Código 24

- *Matrix_mult*: esta función recibe por parámetros dos matrices binarias. Se encarga de multiplicar las dos matrices que fueron pasadas por parámetro y devuelve el resultado. El primer bucle de esta paralelización se puede ejecutar de forma independiente respecto a los bucles internos ya que las operaciones dentro del bucle se pueden ejecutar en hilos independientes al mismo tiempo y los bucles internos seguirían el mismo concepto que la función “transpose”.

```
bin_matrix matrix_mult(bin_matrix A, bin_matrix B)
{
  if (A->cols != B->rows)
  {
    printf("Matrices are incompatible, check dimensions...\n");
    exit(0);
  }

  bin_matrix C;
  C = mat_init(A->rows, B->cols);
```

```
  bin_matrix B_temp = transpose(B);




    for(int i = 0; i < A->rows; i++)
    {
      for(int j = 0  ; j < B->cols; j++)
      {
        unsigned short val = 0;

        for(int k = 0; k < B->rows; k++)
        {
          val = (val ^ (mat_element(A, i, k) & mat_element(B_temp, j,
k)));
        }
        mat_element(C, i, j) = val;
      }
    }


  return C;
}
```

Código 25

- *Make_identity*: esta función recibe por parámetros una matriz binaria y procede a establecerla como una matriz de identidad. Habría que aplicar el mismo método que a la función "transpose".

```
void make_identity(bin_matrix A)
{

  for(int i = 0; i < A->rows; i++)
  {

    for(int j = 0; j < A->cols; j++)
    {
      if(i == j)
      {
        mat_element(A, i, j) = 1;
      }
      else
      {
        mat_element(A, i, j) = 0;
      }
    }
  }
}
```

Código 26

- *Is_identity*: esta función recibe por parámetros una matriz binaria y procede a comprobar si es una matriz de identidad devolviendo una señal para indicar si es verdadero o falso. Habría que aplicar el mismo método que a la función "transpose".

```
bool is_identity(bin_matrix A)
{
  bool flag = true;
  bool return_flag = false;

  for(int i = 0; i < A->rows; i++)
  {
    if(return_flag == false)
    {

      for(int j = 0; j < A->cols; j++)
      {
        if(return_flag == false){
          if(i == j)
          {
            if(mat_element(A, i, j) == 0)
            {
              flag = false;
              return_flag = true;
            }
          }
          else
          {
            if(mat_element(A, i, j) == 1)
            {
              flag = false;
              return_flag = true;
            }
          }
        }
      }
    }
  }
  return flag;
}
```

Código 27

- *Is_zero_matrix*: esta función recibe por parámetros una matriz binaria y procede a comprobar si es una matriz de ceros devolviendo una señal para indicar si es verdadero o falso. De nuevo habría que aplicar el mismo método que a la función "transpose".

```
int is_zero_matrix(bin_matrix A)
{
```

```
  int flag = 1;
  bool return_flag = false;

  for(int i = 0; i < A->rows; i++)
  {
    if(return_flag == false){

      for(int j = 0; j < A->cols; j++)
      {
        if(mat_element(A, i, j) != 0 && return_flag==false)
        {
          flag = 0;
          return_flag = true;
        }
      }
    }
  }
  return flag;
}
```

Código 28

- *Mat_is_equal*: esta función recibe por parámetros dos matrices binarias y procede a comprobar si esas matrices son iguales devolviendo una señal para indicar si es verdadero o falso. Habría que aplicar el mismo método que a la función “transpose”.

```
int mat_is_equal(bin_matrix A, bin_matrix B)
{
  int flag = 1;
  bool return_flag = false;
  if(A->rows != B->rows || A->cols != B->cols)
  {
    flag = 0;
    return flag;
  }

  for(int i = 0; i < A->rows; i++)
  {

    for(int j = 0; j < A->cols; j++)
    {
      if(mat_element(A, i, j) != mat_element(B, i, j) &&
return_flag==false)
      {
        flag = 0;
        return_flag=true;
      }
    }
  }
```

```
  return flag;
}
```

Código 29

- *Add_rows_new*: esta función recibe por parámetros una matriz binaria, dos filas y un rango. Se encarga de insertar los elementos de la primera fila a la segunda fila en el límite del rango y procede a devolver la matriz. Esta función se puede paralelizar porque los elementos de las filas 1 y 2 se pueden insertar en la matriz A en el rango [a, b] de forma independiente ejecutando distintos hilos al mismo tiempo ya que no existen dependencias entre los datos .

```
bin_matrix add_rows_new(bin_matrix A,int row1, int row2, int a, int b)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }

  for(int i = a; i < b; i++)
  {
    mat_element(A, row2, i) = (mat_element(A, row1, i) ^
mat_element(A, row2, i));
  }
  return A;
}
```

Código 30

- *Add_cols*: esta función recibe por parámetros una matriz binaria, dos columnas y un rango. Se encarga de insertar los elementos de las columnas en el límite del rango y procede a devolver la matriz. Esta función utilizaría el mismo concepto de paralelización que la función anterior.

```
bin_matrix add_cols(bin_matrix A,int col1, int col2, int a, int b)
{
  if(col1 < 0 || col1 >= A->cols || col2 < 0 || col2 >= A->cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  for(int i = a; i < b; i++)
  {
    mat_element(A, i, col2) = (mat_element(A, i, col1) ^
mat_element(A, i, col2));
  }
  return A;
}
```

Código 31

- *Circ_matrix_inverse*: esta función recibe por parámetros una matriz binaria, y procede a devolver la matriz inversa de la matriz dada. Habría que aplicar el mismo método que a la función "transpose".

```
bin_matrix circ_matrix_inverse(bin_matrix A)
{
  if(A->rows != A->cols)
  {
    printf("Inverse not possible...\n");
    exit(0);
  }

  if(is_identity(A))
  {
    return A;
  }

  bin_matrix B;
  B = mat_init(A->rows, A->cols);
  make_identity(B);


  int i;
  int flag, prev_flag = 0;


  for(i = 0; i < A->cols; i++)
  {
    if(mat_element(A, i, i) == 1)
    {

      for(int j = 0; j <  A->rows; j++)
      {
        if(i != j && mat_element(A, j, i) == 1)
        {
          add_rows_new(B, i, j, 0, A->cols);
          add_rows_new(A, i, j, i, A->cols);
        }
      }
    }
    else
    {
      int k;

      for(k = i + 1; k < A->rows; k++)
      {
        if(mat_element(A, k, i) == 1)
        {
          add_rows(B, k, i);
```

```
          add_rows(A, k, i);
          i = i - 1;
          break;
        }
      }
    }
  }
  //printf("Out of for loop...\n");
  if(!is_identity(A))
  {
    printf("Could not find inverse, exiting...\n");
    exit(-1);
  }


  return B;
}
```

Código 32

- *Mat_splice*: esta función recibe por parámetros una matriz binaria, dos filas y dos columnas. Se encarga de devolver una matriz binaria que está formada por los datos de la matriz binaria pasada por parámetro limitándola a un tamaño formado por las filas y columnas que fueron pasadas como parámetros. Esta función se puede paralelizar porque los elementos de la matriz A son independientes entre sí con lo cual el doble bucle se puede colapsar en uno y ejecutar simultáneamente distintos hilos para realizar la misma operación sobre todos los elementos a la vez. Pero hay que tener cuidado porque el bucle interno tiene que ser secuencial respecto al primer bucle para poder seguir la cadena de ejecución de los datos.

```
bin_matrix mat_splice(bin_matrix A, int row1, int row2, int col1, int
col2)
{
  int row_count = row2 - row1 + 1;
  int col_count = col2 - col1 + 1;
  int idx1, idx2;

  bin_matrix t = mat_init(row_count, col_count);
  for(int i = 0; i < row_count; i++)
  {
    idx1 = row1 + i;
    for(int j = 0; j < col_count; j++)
    {
      idx2 = col1 + j;
      set_matrix_element(t, i, j, mat_element(A, idx1, idx2));
    }
  }
  return t;
```

```
}
```

Código 33

- *Mat_kernel*: esta función recibe por parámetros una matriz binaria y se encarga de devolver una matriz donde se encontró su base del espacio del kernel. Esta función se puede paralelizar porque no hay dependencias entre los elementos de la matriz A con lo cual el doble bucle se puede colapsar en uno y ejecutar al mismo tiempo varios hilos independientes para realizar la misma operación sobre todos los elementos a la vez. Pero hay que tener cuidado porque el bucle interno tiene que ser secuencial respecto el primer bucle para poder seguir la cadena de ejecución de los datos. El segundo bucle se puede paralelizar perfectamente porque simplemente se están colocando los elementos en una matriz temp y estos elementos se pueden ejecutar a la vez en hilos independientes. La última parte de la función seguiría la misma secuencia que el primer bucle.

```
bin_matrix mat_kernel(bin_matrix A)
{
  int row_count = A->rows;
  int col_count = A->cols;

  bin_matrix temp = mat_init(col_count, row_count + col_count);

  bin_matrix ans = mat_init(col_count, col_count - row_count);

  for(int i = 0; i < temp->rows; i++)
  {

    for(int j = 0; j < row_count; j++)
    {
      set_matrix_element(temp, i, j, mat_element(A, j, i));
    }
  }


  for(int i = 0; i < col_count; i++)
  {
    set_matrix_element(temp, i, i + row_count, 1);
  }

  int r = 0;
  bool return_flag = false;
  for(int r = 0; r < row_count;r++)
  {
    if(mat_element(temp, r, r) == 0)
    {
      int i;
```

```
      for(i = r + 1; i < temp->rows; i++)
      {
        if(mat_element(temp, i, r) && return_flag==false)
        {
          swap(temp, r, i);
          return_flag=true;
        }
      }
      if(i == temp->rows)
      {
        ans = mat_splice(temp, row_count, col_count - 1, row_count,
row_count + col_count - 1);
        return (matrix_rref(ans));
      }
    }
    else
    {
      for(int i = 0; i < temp->rows; i++)
      {
        if(mat_element(temp, i, r) && i != r)
        {
          add_rows(temp, r, i);
        }
      }
    }
  }
  ans = mat_splice(temp, row_count, col_count - 1, row_count,
row_count + col_count - 1);
  return (matrix_rref(ans));
}
```

Código 34

- *Concat_horizontal*: esta función recibe por parámetros dos matrices binarias. Se encarga de devolver la matriz obtenida al realizar la concatenación horizontal de las matrices pasadas como parámetros. Para paralelizar esta función habría que aplicar el mismo método que a la función “transpose”.

```
bin_matrix concat_horizontal(bin_matrix A, bin_matrix B)
{
  if(A->rows != B->rows)
  {
    printf("Incompatible dimensions of the two matrices. Number of
rows should be same.\n");
    exit(0);
  }
  bin_matrix temp = mat_init(A->rows, A->cols + B->cols);


  for(int i = 0; i < temp->rows; i++)
```

```
  {

    for(int j = 0; j < temp->cols; j++)
    {
      if(j < A->cols)
      {
        set_matrix_element(temp, i, j, mat_element(A, i, j));
      }
      else
      {
        set_matrix_element(temp, i, j, mat_element(B, i, j - A-
>cols));
      }
    }
  }
  return temp;
}
```

Código 35

- *Concat_vertical*: esta función recibe por parámetros dos matrices binarias. Se encarga de devolver la matriz obtenida al realizar la concatenación vertical de las matrices pasadas como parámetros. Para paralelizar esta función habría que aplicar el mismo método que a la función “transpose”.

```
bin_matrix concat_vertical(bin_matrix A, bin_matrix B)
{
  if(A->cols != B->cols)
  {
    printf("Incompatible dimensions of the two matrices. Number of
rows should be same.\n");
    exit(0);
  }
  bin_matrix temp = mat_init(A->rows + B->rows, A->cols);


  for(int i = 0; i < temp->rows; i++)
  {

    for(int j = 0; j < temp->cols; j++)
    {
      if(i < A->rows)
      {
        set_matrix_element(temp, i, j, mat_element(A, i, j));
      }
      else
      {
        set_matrix_element(temp, i, j, mat_element(B, i - A->rows,
j));
      }
```

```
    }
  }
  return temp;
}
```

Código 36

- *Print_matrix*: esta función recibe por parámetros una matriz binaria y procede a imprimir esa matriz por pantalla en formato binario. Para paralelizar esta función habría que aplicar el mismo método que a la función "transpose".

```
void print_matrix(bin_matrix A)
{

  for(int i = 0; i < A->rows; i++)
  {

    for (int j = 0; j < A->cols; j++)
    {
      printf("%hu ", mat_element(A, i, j));
    }
    printf("\n");
  }
}
```

Código 37

- *Print_matrix_char*: esta función recibe por parámetros una matriz binaria y un output y procede a imprimir esa matriz por pantalla en formato texto. Para paralelizar esta función habría que aplicar el mismo método que a la función "transpose".

```
void print_matrix_char(bin_matrix A, char* output)
{

  for(int i = 0; i < A->rows; i++)
  {

    for (int j = 0; j < A->cols; j++)
    {
      char num[2];
      sprintf(num,"%hu",mat_element(A, i, j));
      strcat(output,num);
    }
    printf("\n");
  }
}
```

Código 38

3.4.7. mceliece.h y mceliece.c

Este archivo contiene funciones que se invocan y utilizan para inicializar parámetros y generar vectores. Las funciones que contiene el archivo son las siguientes:

```
#include "qc_mdpc.h"
#include "matrix.h"
#include "utility.h"


#ifndef MCELIECE_H
#define MCELIECE_H

typedef struct mceliece
{
   mdpc code;
   bin_matrix public_key;
}*mcc;

mcc mceliece_init(int n0, int p, int w, int t);
void delete_mceliece(mcc A);
bin_matrix get_error_vector(int len, int t);
bin_matrix encrypt(bin_matrix msg, mcc crypt);
bin_matrix decrypt(bin_matrix word, mcc crypt);
#endif
```

Código 39

- *Mceliece_init*: esta función recibe como parámetros los datos n0, p, t y w y procede a crear el criptosistema de mceliece utilizando los parámetros dados.

```
mcc mceliece_init(int n0, int p, int w, int t)
{
    mcc crypt;
    crypt = (mcc)safe_malloc(sizeof(struct mceliece));
    crypt->code = qc_mdpc_init(n0, p, w, t);
    crypt->public_key = generator_matrix(crypt->code);

    return crypt;
}
```

Código 40

- *Delete_mceliece*: esta función recibe como parámetro un criptosistema, y procede a borrarlo y liberar el espacio en memoria que fue ocupado por él.

```
void delete_mceliece(mcc A)
{
    delete_qc_mdpc(A->code);
    delete_matrix(A->public_key);
    free(A);
}
```

Código 41

- *Get_error_vector*: esta función recibe como parámetros una longitud y un peso. Se encarga de generar un vector de error aleatorio utilizando los parámetros dados .

```
bin_matrix get_error_vector(int len, int t)
{
     bin_matrix error = mat_init(1, len);
     int weight = 0;
     int idx;
     while(weight < t)
     {
        idx = random_val(1, len - 1, -1);
        if(!get_matrix_element(error, 0, idx))
        {
            set_matrix_element(error, 0, idx, 1);
            weight++;
        }
     }
     return error;
}
```

Código 42

- *Encrypt*: esta función recibe como parámetros una matriz binaria y un criptosistema. Se encarga de cifrar la matriz binaria que contiene el mensaje, para ello invoca a las funciones de add_matrix y get_error_vector y usa el criptosistema para devolver el texto cifrado.

```
bin_matrix encrypt(bin_matrix msg, mcc crypt)
{
    if(msg->cols != crypt->public_key->rows)
    {
        printf("Length of message is incorrect.\n");
        exit(0);
    }
    bin_matrix error = get_error_vector(crypt->code->n, crypt->code-
>t);

    bin_matrix word = add_matrix(matrix_mult(msg, crypt->public_key),
error);

    return word;
}
```

Código 43

- *Decrypt*: esta función recibe como parámetros una matriz binaria y un criptosistema. Se encarga de descifrar el texto cifrado que viene dado en la matriz binaria, llamando para ello a las funciones decode y mat_splice, que emplean el criptosistema para devolver el mensaje original en formato binario.

```
bin_matrix decrypt(bin_matrix word, mcc crypt)
```

```
{
    if(word->cols != crypt->code->n)
    {
        printf("Length of message is incorrect.\n");
        exit(0);
    }

    bin_matrix msg = decode(word, crypt->code);
    msg = mat_splice(msg, 0, msg->rows - 1, 0, crypt->code->k - 1);
    return msg;
}
```

Código 44

3.4.8. qc_mdpc.h y qc_mdpc.c

Este archivo contiene una serie de funciones que se encargan de realizar numerosas operaciones. Estas funciones son:

```
#include "matrix.h"
#include "utility.h"


#ifndef QC_MDPC_H
#define QC_MDPC_H
typedef struct qc_mdpc
{
    unsigned short* row;
    int n0, p, w, t, n, k, r;
}*mdpc;

mdpc qc_mdpc_init(int n0, int p, int w, int t);
void delete_qc_mdpc(mdpc A);
int random_val(int min, int max, unsigned seed);
int get_row_weight(unsigned short* row, int min, int max);
void reset_row(unsigned short* row, int min, int max);
unsigned short* shift(unsigned short* row, int x, int len);
unsigned short* splice(unsigned short* row, int min, int max);
bin_matrix make_matrix(int rows, int cols, unsigned short* vec, int x);
bin_matrix generator_matrix(mdpc code);
bin_matrix parity_check_matrix(mdpc code);
int get_max(int* vec, int len);
bin_matrix encode(bin_matrix vec, mdpc code);
bin_matrix decode(bin_matrix word, mdpc code);

#endif
```

Código 45

- *Qc_mdpc_init*: esta función recibe como parámetros los datos n0, p, t y w y procede a crear un código de comprobación de paridad de densidad moderada (MDPC)utilizando los parámetros dados.

```
mdpc qc_mdpc_init(int n0, int p, int w, int t)
{
    mdpc code;
    code = (mdpc)safe_malloc(sizeof(struct qc_mdpc));
    code->n0 = n0;
    code->p = p;
    code->w = w;
    code->t = t;
    code->n = n0 * p;
    code->r = p;
    code->k = (n0 - 1) * p;
    unsigned seed;
    code->row = (unsigned short*)calloc(n0 * p, sizeof(unsigned
short));
    printf("Input seed or -1 to use default seed: ");
    scanf("%u", &seed);
    time_t tx;
    if(seed == -1)
    {
      srand((unsigned) time(&tx));
    }
    else
    {
        srand(seed);
    }

    while(1)
    {
        int flag = 0;
        int idx;
        while(flag < w)
        {
            idx = random_val(0, (n0 * p) - 1, seed);
            if(!code->row[idx])
            {
              code->row[idx] = 1;
              flag = flag + 1;
            }
        }
        if((get_row_weight(code->row, (n0 - 1) * p, (n0 * p)-1)) % 2
== 1)
        {
            break;
        }
        reset_row(code->row, 0, n0 * p);
    }
    printf("MDPC code generated....\n");
    return code;
}
```

Código 46

- *Delete_qc_mdpc*: esta función recibe como parámetro un código MDPC y procede a borrarlo y liberar el espacio en memoria que fue ocupado por este código.

```
void delete_qc_mdpc(mdpc A)
{
  free(A);
}
```

Código 47

- *Random_val*: esta función recibe como parámetros un valor mínimo, un valor máximo y una semilla. Se encarga de devolver un número entero aleatorio creado dentro del rango limitado por estos valores máximo y mínimo.

```
int random_val(int min, int max, unsigned seed)
{
    int r;
    const unsigned int range = 1 + max - min;
    const unsigned int buckets = RAND_MAX / range;
    const unsigned int limit = buckets * range;

    do
    {
        r = rand();
    } while (r >= limit);

    return min + (r / buckets);
}
```

Código 48

- *Get_row_weight*: esta función recibe como parámetros un mínimo, un máximo y una fila. Procede a devolver el peso de la fila en el rango formado por el máximo y el mínimo. Existe la opción de paralelización porque al ser independientes los elementos del vector *row* estos se pueden comprobar si son iguales a 1 de golpe ejecutándolos en hilos separados en vez de esperar a que finalice cada vuelta del bucle para iniciar la siguiente comprobación y esto permitirá mejorar el rendimiento de la función.

```
int get_row_weight(unsigned short* row, int min, int max)
{
    int weight = 0;
    int i;
    for(i = min; i < max + 1; i++)
    {
        if(row[i] == 1)
        {
            weight++;
        }
```

```
    }
    return weight;
}
```

Código 49

- *Reset_row*: esta función recibe como parámetros un valor mínimo, un valor máximo y una fila. Procede a reiniciar todas las posiciones de la fila a 0 en el rango formado por los valores máximo y mínimo. Esta función se puede paralelizar perfectamente porque solo se resetea el vector *row* con lo cual si se paralelizan los elementos que se inicializan en el vector se pueden ejecutar en hilos distintos al mismo tiempo.

```
void reset_row(unsigned short* row, int min, int max)
{
    int i;

    for(i = min; i < max + 1; i++)
    {
      row[i] = 0;
    }
}
```

Código 50

- *Shift*: esta función recibe como parámetros una longitud, un número de posiciones y una fila. Procede a devolver una fila donde un número de posiciones han sido rotadas a la derecha. Esta función se puede paralelizar porque los elementos de los vectores *temp* y *row* son independientes entre sí y eso permite ejecutarlos a la vez en hilos independientes.

```
unsigned short* shift(unsigned short* row, int x, int len)
{
    unsigned short* temp = (unsigned short*)calloc(len,
sizeof(unsigned short));
    int i;

    for(i = 0; i < len; i++)
    {
      temp[(i + x) % len] = row[i];
    }
    return temp;
}
```

Código 51

- *Make_matrix*: esta función recibe como parámetros filas, columnas, vector y número de posiciones. Procede a devolver una matriz binaria circular. Esta función se puede paralelizar porque los elementos del vector son independientes entre sí y eso permite ejecutarlos en hilos independientes al mismo tiempo.

```
bin_matrix make_matrix(int rows, int cols, unsigned short* vec, int x)
```

```
{
    bin_matrix mat = mat_init(rows, cols);
    set_matrix_row(mat, 0, vec);
    int i;

    for(i = 1; i < rows; i++)
    {
      vec = shift(vec, x, cols);
      set_matrix_row(mat, i, vec);
    }
    return mat;
}
```

Código 52

- *Splice*: esta función recibe como parámetros un valor mínimo, un valor máximo y una fila. Procede a devolver una fila formada por los datos de la fila pasada como parámetro y limitada por los valores máximo y mínimo. Esta función se puede paralelizar ya que los elementos de los vectores son independientes entre sí y no tienen relación entre ellos con lo cual todos los elementos se pueden ejecutar en hilos independientes a la vez.

```
unsigned short* splice(unsigned short* row, int min, int max)
{
    unsigned short* temp = (unsigned short*)calloc(max - min,
sizeof(unsigned short));
    int i;

    for(i = min; i < max; i++)
    {
      temp[i - min] = row[i];
    }
    return temp;
}
```

Código 53

- *Parity_check_matrix*: esta función recibe como parámetro un código MDPC y procede a crear una matriz de paridad. El único sitio donde se podría hacer una paralelización es en el *for* donde se obtiene la versión final de la matriz H. Se ahorra mucho tiempo si se paraleliza esta sección del código ya que en vez de crear una matriz M y concatenarla con H por cada vuelta del bucle, se pueden ir creando en hilos separados de forma independiente y concatenarlas con H. Así la matriz H final se crea de manera más eficiente.

```
bin_matrix parity_check_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
```

```
    bin_matrix H = make_matrix(code->p, code->p, splice(code->row, 0,
code->p), 1);
    int i;

    for(i = 1; i < code->n0; i++)
    {
      bin_matrix M = make_matrix(code->p, code->p, splice(code->row, i
* code->p, (i + 1) * code->p), 1);
      H = concat_horizontal(H, M);
    }
    end = clock();
    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for H: %f\n", cpu_time_used);
    return H;
}
```

Código 54

- *Generator_matrix*: esta función recibe como parámetro un código MDPC y procede a crear la matriz generadora. El único sitio donde se podría hacer una paralelización es en el *for* donde se obtiene la matriz Q. Se ahorra mucho tiempo si se paraleliza esta sección del código ya que en vez de crear una matriz M, transponerla con H_inv y concatenarla con Q en cada vuelta, se pueden ir creando en hilos separados de forma independiente. Así la matriz Q final se crea de manera más eficiente.

```
bin_matrix generator_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    bin_matrix H = parity_check_matrix(code);



    printf("Construction of G started...\n");
    bin_matrix H_inv = circ_matrix_inverse(make_matrix(code->p, code-
>p, splice(code->row, (code->n0 - 1) * code->p, code->n), 1));
    bin_matrix H_0 = make_matrix(code->p, code->p, splice(code->row,
0, code->p), 1);
    bin_matrix Q = transpose(matrix_mult(H_inv,  H_0));

    bin_matrix M;
    int i;

    for(i = 1; i < code->n0 - 1; i++)
    {
      M = make_matrix(code->p, code->p, splice(code->row, i * code->p,
(i + 1) * code->p), 1);
      M = transpose(matrix_mult(H_inv, M));
```

```
      Q = concat_vertical(Q, M);
    }
    bin_matrix I = mat_init(code->k, code->k);
    make_identity(I);
    bin_matrix G = concat_horizontal(I, Q);


    end = clock();
    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for G: %f\n", cpu_time_used);
    printf("Generator matrix generated....\n");
    return G;
}
```

Código 55

- *Get_max*: esta función recibe como parámetros un vector y una longitud y procede a devolver el máximo elemento del vector. Al ser un vector pequeño donde los elementos son independientes entre sí, esta función se puede paralelizar para acelerar el proceso de encontrar el mayor elemento dentro del vector buscando a la vez en todos los elementos que se están ejecutando en hilos separados.

```
int get_max(int* vec, int len)
{
    int max = vec[0];
    int i;

    for(i = 1; i < len; i++)
    {
        if(vec[i] > max)
        {
            max = vec[i];
        }
    }
    return max;
}
```

Código 56

- *Encode*: esta función recibe como parámetros una matriz binaria y un código MDPC y procede a devolver la matriz codificada como código MDPC.

```
bin_matrix encode(bin_matrix vec, mdpc code)
{
  bin_matrix G = generator_matrix(code);
  bin_matrix msg = matrix_mult(vec, G);
  return msg;
}
```

Código 57

- *Decode*: esta función recibe como parámetros una matriz binaria y un código MDPC y procede a devolver la matriz decodificada. En esta función la paralelización se puede producir a distintos niveles ya que el bucle principal se puede paralelizar ya que los elementos del mensaje son independientes entre sí y se pueden ejecutar en hilos distintos. Ya dentro del bucle principal hay otros tres bucles y estos también se pueden paralelizar. El primero se puede paralelizar sin ningún problema ya que simplemente se inicializan todos los elementos del vector *unsetesfied* al mismo tiempo. El segundo bucle es un doble bucle donde el segundo bucle interno tiene dependencia del bucle externo con lo cual si se paraleliza el bucle externo el bucle interno debe tener una paralización secuencial debido a la dependencia del bucle externo. El ultimo bucle se puede paralelizar ya que los elementos del vector *unsitiesfied* y los elementos de la matriz *syn* son independientes entre sí y se pueden ejecutar en hilos separados.

```
bin_matrix decode(bin_matrix word, mdpc code)
{
  bin_matrix H = parity_check_matrix(code);
  bin_matrix syn  = matrix_mult(H, transpose(word));
  int limit = 10;
  int delta = 5;
  int i,j,k,x;

    for(i = 0; i < limit; i++)
    {

        int unsatisfied[word->cols];
        for(x = 0; x < word->cols; x++)
        {
            unsatisfied[x] = 0;
        }
        for(j = 0; j < word->cols; j++)
        {
            for(k = 0; k < H->rows; k++)
            {
                if(get_matrix_element(H, k, j) == 1)
                {
                    if(get_matrix_element(syn, k, 0) == 1)
                    {
                        unsatisfied[j] = unsatisfied[j] + 1;
                    }
                }
            }
        }

        int b = get_max(unsatisfied, word->cols) - delta;
        for(j = 0; j < word->cols; j++)
        {
            if(unsatisfied[j] >= b)
```

```
            {
                set_matrix_element(word, 0, j,
(get_matrix_element(word, 0, j) ^ 1));
                syn = add_matrix(syn, mat_splice(H, 0, H->rows - 1, j,
j));
            }
        }

        if(is_zero_matrix(syn))
        {
            return word;
        }
    }
    printf("Decoding failure...\n");
    exit(0);
}
```

Código 58

3.4.9. utility.h y utility.c

Este archivo se encarga de asegurar que la reserva en memoria se realiza de forma segura y no sobrepasa los límites máximos indicados, además de asegurar que no se produzca ningún tipo de error o violación de segmento.

```
#include <stddef.h>

#ifndef UTIL
#define UTIL

void* safe_malloc(size_t n);

#endif


#include "utility.h"
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>

void* safe_malloc(size_t n)
{
    void* p = malloc(n);
    if (!p)
    {
        fprintf(stderr, "Out of memory(%lu bytes)\n",(size_t)n);
        exit(EXIT_FAILURE);
    }
    return p;
}
```

Código 59

4.Implementación realizada

En este capítulo abordaremos las diferentes mejoras sobre el código, realizadas en dos "rondas". Ambas secciones incluyen aquellas funciones mejoradas, el porqué de su mejora, además de su código correspondiente. Para facilitar la lectura y compresión de este capítulo repetiremos información que ya sea ha presentado en el capítulo 3.

4.1. Primera ronda de mejoras

La implementación del algoritmo Classic McEliece que fue elegida permite la opción de una mejora de rendimiento a la hora de ejecutar el proceso completo de cifrado y descifrado mediante el uso del lenguaje de programación OpenACC. Con este lenguaje se puede realizar una paralelización del código original que permita compilar y ejecutar las partes del código modificadas y mejoradas en un emulador GPU y poder así operar con las matrices indicadas en paralelo. Este proceso se conoce como paralelización y se encarga de dividir una tarea o un programa en partes más pequeñas y ejecutarlas de manera simultánea en múltiples hilos de ejecución o procesadores, con el objetivo de mejorar el rendimiento y aprovechar al máximo los recursos disponibles en sistemas multiprocesador o multi-hilo. Hay que tener en cuenta que para poder emplear la paralelización los datos tienen que ser independientes ya que si existe algún tipo de dependencia habría que esperar a que un dato se termine de ejecutar para poder lanzar el siguiente. El ejemplo más sencillo es un doble bucle for, básicamente el doble bucle se colapsa en uno único formando un bus en el que se ejecutan las operaciones sobre todos los elementos de ese bus al mismo tiempo gracias al uso de hilos que se crean dentro del bus, permitiendo poner un elemento de la matriz dentro de un hilo y ejecutarlo de forma independiente del resto de hilos que se ejecutan al mismo tiempo. Una vez que finaliza la ejecución el espacio de memoria reservado para los hilos se libera. Este método es muy eficaz a la hora de mejorar el rendimiento si se quieren cifrar y descifrar mensajes de gran cantidad de caracteres.

```
//Inicialización del array
for(i= 0; i < SIZE; i++){
      for(j = 0; j < SIZE; j++){
            array[i][j] = i + j;
      }
}
//Paralelización del bucle for con OpenACC
#pragma acc parallel loop
for(i= 0; i < SIZE; i++){
      for(j = 0; j < SIZE; j++){
            array[i][j] *= 2;
      }
}
```

Código 60

A continuación, se muestran y describen las mejoras realizadas sobre el código explicando qué acción realiza cada método y cuáles son los cambios que se han realizado sobre el código usando el lenguaje de programación OpenACC.

4.1.1. matrix.c

- *Set_matrix_row*: esta función recibe por parámetros una matriz binaria, un índice de posición de fila, un índice de posición de columna y un vector de valores de la matriz auxiliar. Se encarga de insertar el valor pasado por parámetro dentro de la matriz binaria empleando los índices de posición de la fila y de la columna. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
void set_matrix_row(bin_matrix A, int row, unsigned short* vec)
{
  if(row < 0 || row >= A->rows)
  {
    printf("Row index out of range\n");
    exit(0);
  }
  #pragma acc parallel loop independent
  for(int i = 0; i < A->cols; i++)
  {
    set_matrix_element(A, row, i, vec[i]);
  }}
```

Código 61

- *Transpose*: esta función recibe por parámetro una matriz binaria y devuelve la matriz traspuesta de la matriz binaria. El bucle exterior se ejecuta en paralelo y puede ser dividido en múltiples hilos de ejecución. Por otro lado, el bucle interno se ejecuta en serie lo que garantiza que cada hilo de ejecución acceda a los elementos de la matriz de manera secuencial y sin conflictos.

```
bin_matrix transpose(bin_matrix A)
{
  bin_matrix B;
  B = mat_init(A->cols, A->rows);
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
      set_matrix_element(B, j, i, mat_element(A, i, j));
    }
  }
```

```
  return B;
}
```

Código 62

- *Add_rows*: esta función recibe por parámetros una matriz binaria y dos filas. Se encarga de insertar una fila dentro de otra fila de la matriz binaria y procede a devolverla al finalizar la operación. El bucle se ejecuta en paralelo y puede ser dividido en múltiples hilos de ejecución. Cada hilo de ejecución se encarga de actualizar un subconjunto de los elementos de la fila *row2*.

```
bin_matrix add_rows(bin_matrix A,int row1, int row2)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->rows)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  #pragma acc parallel loop independent
  for(int i = 0; i < A->cols; i++)
  {
    mat_element(A, row2, i) = (mat_element(A, row1, i) ^
mat_element(A, row2, i));
  }
  return A;
}
```

Código 63

- *Add_matrix*: esta función recibe por parámetros dos matrices binarias, procede a insertar una de las matrices dentro de la otra y devuelve el resultado. El bucle exterior se ejecuta en paralelo y puede ser dividido en múltiples hilos de ejecución. Por otro lado, el bucle interno se ejecuta en serie lo que garantiza que cada hilo de ejecución acceda a los elementos de la matriz de manera secuencial y sin conflictos.

```
bin_matrix add_matrix(bin_matrix A, bin_matrix B)
{
  if(A->rows != B->rows || A->cols != B->cols)
  {
    printf("Incompatible dimenions for matrix addition.\n");
    exit(0);
  }
  bin_matrix temp = mat_init(A->rows, A->cols);
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
```

```
      set_matrix_element(temp, i, j, (mat_element(A, i, j) ^
mat_element(B, i, j)));
    }
  }
  return temp;
}
```

Código 64

- *Swap*: esta función recibe por parámetros una matriz binaria y dos filas. Se encarga de intercambiar dos filas de una matriz. El bucle se ejecuta en paralelo y puede ser dividido en múltiples hilos de ejecución. Cada hilo de ejecución se encarga de intercambiar un subconjunto de los elementos de las dos filas, lo que permite una ejecución más rápida en sistemas paralelos.

```
void swap(bin_matrix A, int row1, int row2)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->rows)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  int temp;
  #pragma acc paralleel loop independent
  for(int i = 0; i < A->cols; i++)
  {
    temp = mat_element(A, row1, i);
    mat_element(A, row1, i) = mat_element(A, row2, i);
    mat_element(A, row2, i) = temp;
  }}
```

Código 65

- *Matrix_rref*: esta función recibe por parámetro una matriz binaria. La función se encarga de obtener una versión echlon reducida de la matriz binaria y devolver ese resultado. Cada uno de los bucles de la función contiene un pragma para poder ejecutarlos en paralelo, así puede ser dividido en múltiples hilos de ejecución.

```
bin_matrix matrix_rref(bin_matrix A)
{
  int lead = 0;
  int row_count = A->rows;
  int col_count = A->cols;
  bin_matrix temp = mat_init(row_count, col_count);
  temp = mat_copy(A);

  int r = 0;
  while(r < row_count)
  {
    if(mat_element(temp, r, r) == 0)
```

```
    {
      int i;
      #pragma acc parallel loop independent
      for(i = r + 1; i < temp->rows; i++)
      {
        if(mat_element(temp, i, r) == 1)
        {
          swap(temp, r, i);
          break;
        }
      }
      if(i == row_count)
      {
        printf("Matix cannot be transformed into row echlon form...");
        exit(1);
      }
    }
    else
    {
      #pragma acc parallel loop independent
      for(int i = 0; i < row_count; i++)
      {
        if(mat_element(temp, i, r) == 1 && i != r)
        {
          add_rows(temp, r, i);
        }
      }
      r++;
    }
  }
  return temp;
}
```

Código 66

- *Matrix_mult*: esta función recibe por parámetros dos matrices binarias. Se encarga de multiplicar las dos matrices que fueron pasadas por parámetro y devuelve el resultado. El código está utilizando la directiva #pragma acc para paralelizar los bucles externos e internos, indicando que se pueden ejecutar en paralelo independientemente. Además, utiliza la directiva *loop seq* para indicar que el bucle más interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bin_matrix matrix_mult(bin_matrix A, bin_matrix B)
{
  if (A->cols != B->rows)
  {
    printf("Matrices are incompatible, check dimensions...\n");
    exit(0);
  }
```

```
  bin_matrix C;
  C = mat_init(A->rows, B->cols);
  bin_matrix B_temp = transpose(B);

  #pragma acc paralleel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc paralleel loop independent
    for(int j = 0  ; j < B->cols; j++)
    {
      unsigned short val = 0;
      #pragma acc loop seq
      for(int k = 0; k < B->rows; k++)
      {
        val = (val ^ (mat_element(A, i, k) & mat_element(B_temp, j,
k)));
      }
      mat_element(C, i, j) = val;
    }
  }

  return C;
}
```

Código 67

- *Make_identity*: esta función recibe por parámetros una matriz binaria y procede a establecerla como una matriz de identidad. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva *loop seq* para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
void make_indentity(bin_matrix A)
{
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
      if(i == j)
      {
        mat_element(A, i, j) = 1;
      }
      else
      {
        mat_element(A, i, j) = 0;
      }
```

```
    }
  }
}
```

Código 68

- *Is_identity*: esta función recibe por parámetro una matriz binaria y procede a comprobar si es una matriz de identidad devolviendo una señal para indicar si es verdadero o falso. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bool is_identity(bin_matrix A)
{
  bool flag = true;
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
      if(i == j)
      {
        if(mat_element(A, i, j) == 0)
        {
          flag = false;
          return flag;
        }
      }
      else
      {
        if(mat_element(A, i, j) == 1)
        {
          flag = false;
          return flag;
        }
      }
    }
  }
  return flag;
}
```

Código 69

- *Is_zero_matrix*: esta función recibe por parámetro una matriz binaria y procede a comprobar si es una matriz de ceros devolviendo una señal para indicar si es verdadero o falso. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
int is_zero_matrix(bin_matrix A)
{
  int flag = 1;

  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
      if(mat_element(A, i, j) != 0)
      {
        flag = 0;
        return flag;
      }
    }
  }
  return flag;
}
```

Código 70

- *Mat_is_equal*: esta función recibe por parámetros dos matrices binarias y procede a comprobar si esas matrices son iguales devolviendo una señal para indicar si es verdadero o falso. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
int mat_is_equal(bin_matrix A, bin_matrix B)
{
  int flag = 1;
  if(A->rows != B->rows || A->cols != B->cols)
  {
    flag = 0;
    return flag;
  }
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < A->cols; j++)
    {
      if(mat_element(A, i, j) != mat_element(B, i, j))
      {
        flag = 0;
        return flag;
      }
    }
  }
```

```
  return flag;
}
```

Código 71

- *Add_rows_new*: esta función recibe por parámetros una matriz binaria, dos filas y un rango. Se encarga de insertar los elementos de la primera fila a la segunda fila en el límite del rango y procede a devolver la matriz. El bucle de la función se puede ejecutar en paralelo permitiendo dividirlo en múltiplos hilos para realizar la ejecución.

```
bin_matrix add_rows_new(bin_matrix A,int row1, int row2, int a, int b)
{
  if(row1 < 0 || row1 >= A->rows || row2 < 0 || row2 >= A->cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  #pragma acc parallel loop independent
  for(int i = a; i < b; i++)
  {
    mat_element(A, row2, i) = (mat_element(A, row1, i) ^
mat_element(A, row2, i));
  }
  return A;
}
```

Código 72

- *Add_cols*: esta función recibe por parámetros una matriz binaria, dos columnas y un rango. Se encarga de insertar los elementos de la primera fila a la segunda fila en el límite del rango y procede a devolver la matriz. El bucle de la función se puede ejecutar en paralelo permitiendo dividirlo en múltiples hilos para realizar la ejecución.

```
bin_matrix add_cols(bin_matrix A,int col1, int col2, int a, int b)
{
  if(col1 < 0 || col1 >= A->cols || col2 < 0 || col2 >= A->cols)
  {
    printf("Matrix index out of range\n");
    exit(0);
  }
  #pragma acc parallel loop independent
  for(int i = a; i < b; i++)
  {
    mat_element(A, i, col2) = (mat_element(A, i, col1) ^
mat_element(A, i, col2));
  }
  return A;
}
```

Código 73

- *Circ_matrix_inverse*: esta función recibe por parámetros una matriz binaria, y procede a devolver la matriz inversa de la matriz dada. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bin_matrix circ_matrix_inverse(bin_matrix A)
{
  if(A->rows != A->cols)
  {
    printf("Inverse not possible...\n");
    exit(0);
  }

  if(is_identity(A))
  {
    return A;
  }

  bin_matrix B;
  B = mat_init(A->rows, A->cols);
  make_indentity(B);


  int i;
  int flag, prev_flag = 0;

  #pragma acc parallel loop independent
  for(i = 0; i < A->cols; i++)
  {
    if(mat_element(A, i, i) == 1)
    {
      #pragma acc loop seq
      for(int j = 0; j <  A->rows; j++)
      {
        if(i != j && mat_element(A, j, i) == 1)
        {
          add_rows_new(B, i, j, 0, A->cols);
          add_rows_new(A, i, j, i, A->cols);
        }
      }
    }
    else
    {
      int k;
      #pragma acc loop seq
      for(k = i + 1; k < A->rows; k++)
      {
```

```
        if(mat_element(A, k, i) == 1)
        {
          add_rows(B, k, i);
          add_rows(A, k, i);
          i = i - 1;
          break;
        }
      }
    }
  }

  if(!is_identity(A))
  {
    printf("Could not find inverse, exiting...\n");
    exit(-1);
  }


  return B;
}
```

Código 74

- *Mat_splice*: esta función recibe por parámetros una matriz binaria, dos filas y dos columnas. Se encarga de devolver una matriz binaria que está formada por los datos de la matriz binaria pasada por parámetro limitándola a un tamaño formado por las filas y columnas que fueron pasadas como parámetros. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bin_matrix mat_splice(bin_matrix A, int row1, int row2, int col1, int
col2)
{
  int row_count = row2 - row1 + 1;
  int col_count = col2 - col1 + 1;
  int idx1, idx2;

  bin_matrix t = mat_init(row_count, col_count);

  #pragma acc parallel loop independent
  for(int i = 0; i < row_count; i++)
  {
    idx1 = row1 + i;
    #pragma acc loop seq
    for(int j = 0; j < col_count; j++)
    {
      idx2 = col1 + j;
      set_matrix_element(t, i, j, mat_element(A, idx1, idx2));
```

```
    }
  }
  return t;
}
```

Código 75

- *Mat_kernel*: esta función recibe por parámetro una matriz binaria y se encarga de devolver una matriz donde se encontró su base del espacio del kernel. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos. En la segunda parte de la función, se realiza la misma operación ejecutando en paralelo el primer bucle de forma independiente y el bucle interno se ejecuta de forma secuencial porque los datos dependen del primer bucle.

```
bin_matrix mat_kernel(bin_matrix A)
{
  int row_count = A->rows;
  int col_count = A->cols;

  bin_matrix temp = mat_init(col_count, row_count + col_count);

  bin_matrix ans = mat_init(col_count, col_count - row_count);
  #pragma acc parallel loop independent
  for(int i = 0; i < temp->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < row_count; j++)
    {
      set_matrix_element(temp, i, j, mat_element(A, j, i));
    }
  }

  #pragma acc parallel loop independent
  for(int i = 0; i < col_count; i++)
  {
    set_matrix_element(temp, i, i + row_count, 1);
  }

  int r = 0;
  #pragma acc parallel loop independent
  while(r < row_count)
  {
    if(mat_element(temp, r, r) == 0)
    {
      int i;
      #pragma acc loop seq
      for(i = r + 1; i < temp->rows; i++)
```

```
      {
        if(mat_element(temp, i, r))
        {
          swap(temp, r, i);
          break;
        }
      }
      if(i == temp->rows)
      {
        ans = mat_splice(temp, row_count, col_count - 1, row_count,
row_count + col_count - 1);
        return (matrix_rref(ans));
      }
    }
    else
    {
      #pragma acc loop seq
      for(int i = 0; i < temp->rows; i++)
      {
        if(mat_element(temp, i, r) && i != r)
        {
          add_rows(temp, r, i);
        }
      }
      r++;
    }
  }
  ans = mat_splice(temp, row_count, col_count - 1, row_count,
row_count + col_count - 1);
  return (matrix_rref(ans));
}
```

Código 76

- *Concat_horizontal*: esta función recibe por parámetros dos matrices binarias. Se encarga de devolver la matriz obtenida al realizar la concatenación horizontal de las matrices pasadas como parámetros. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bin_matrix concat_horizontal(bin_matrix A, bin_matrix B)
{
  if(A->rows != B->rows)
  {
    printf("Incompatible dimensions of the two matrices. Number of
rows should be same.\n");
    exit(0);
  }
```

```
  bin_matrix temp = mat_init(A->rows, A->cols + B->cols);

  #pragma acc parallel loop independent
  for(int i = 0; i < temp->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < temp->cols; j++)
    {
      if(j < A->cols)
      {
        set_matrix_element(temp, i, j, mat_element(A, i, j));
      }
      else
      {
        set_matrix_element(temp, i, j, mat_element(B, i, j - A-
>cols));
      }
    }
  }
  return temp;
}
```

Código 77

- *Concat_vertical*: esta función recibe por parámetros dos matrices binarias. Se encarga de devolver la matriz obtenida al realizar la concatenación vertical de las matrices pasadas como parámetros. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
bin_matrix concat_vertical(bin_matrix A, bin_matrix B)
{
  if(A->cols != B->cols)
  {
    printf("Incompatible dimensions of the two matrices. Number of
rows should be same.\n");
    exit(0);
  }
  bin_matrix temp = mat_init(A->rows + B->rows, A->cols);

  #pragma acc parallel loop independent
  for(int i = 0; i < temp->rows; i++)
  {
    #pragma acc loop seq
    for(int j = 0; j < temp->cols; j++)
    {
      if(i < A->rows)
      {
```

```
        set_matrix_element(temp, i, j, mat_element(A, i, j));
      }
      else
      {
        set_matrix_element(temp, i, j, mat_element(B, i - A->rows,
j));
      }
    }
  }
  return temp;
}
```

Código 78

- *Print_matrix*: esta función recibe por parámetro una matriz binaria y procede a imprimir esa matriz por pantalla en formato binario. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
void print_matrix(bin_matrix A)
{
  #pragma acc parallel loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for (int j = 0; j < A->cols; j++)
    {
      printf("%hu ", mat_element(A, i, j));
    }
    printf("\n");
  }
}
```

Código 79

- *Print_matrix_char*: esta función recibe por parámetros una matriz binaria y un output y procede a imprimir esa matriz por pantalla en formato texto. El código está utilizando la directiva #pragma acc para paralelizar el bucle externo, indicando que se puede ejecutar en paralelo independientemente. Además, utiliza la directiva loop seq para indicar que el bucle interno no puede ejecutarse en paralelo debido a la dependencia de datos.

```
void print_matrix_char(bin_matrix A, char* output)
{
  #pragma acc paralll loop independent
  for(int i = 0; i < A->rows; i++)
  {
    #pragma acc loop seq
    for (int j = 0; j < A->cols; j++)
```

```
    {
      char num[2];
      sprintf(num,"%hu",mat_element(A, i, j));
      strcat(output,num);
    }
    printf("\n");
  }
}
```

Código 80

4.1.2. qc_mdpc.c

- *Get_row_weight*: esta función recibe como parámetros un valor mínimo, un valor máximo y una fila. Procede a devolver el peso de la fila en el rango formado por los valores máximo y mínimo. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
int get_row_weight(unsigned short* row, int min, int max)
{
    int weight = 0;
    int i;
    #pragma acc parallel loop independent
    for(i = min; i < max + 1; i++)
    {
        if(row[i] == 1)
        {
            weight++;
        }
    }
    return weight;
}
```

Código 81

- *Reset_row*: esta función recibe como parámetros un valor mínimo, un valor máximo y una fila. Procede a restablecer todas las posiciones de la fila a 0 en el rango formado por los valores máximo y mínimo. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
void reset_row(unsigned short* row, int min, int max)
{
    int i;
  #pragma acc parallel loop independent
    for(i = min; i < max + 1; i++)
    {
      row[i] = 0;
    }
```

```
}
```

Código 82

- *Shift*: esta función recibe como parámetros una longitud, número de posiciones y una fila. Procede a devolver una fila donde un número de posiciones han sido rotadas a la derecha. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
unsigned short* shift(unsigned short* row, int x, int len)
{
    unsigned short* temp = (unsigned short*)calloc(len,
sizeof(unsigned short));
    int i;
  #pragma acc parallel loop independent
    for(i = 0; i < len; i++)
    {
      temp[(i + x) % len] = row[i];
    }
    return temp;
}
```

Código 83

- *Make_matrix*: esta función recibe como parámetros filas, columnas, un vector y un número de posiciones. Procede a devolver una matriz binaria circular. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
bin_matrix make_matrix(int rows, int cols, unsigned short* vec, int x)
{
    bin_matrix mat = mat_init(rows, cols);
    set_matrix_row(mat, 0, vec);
    int i;
  #pragma acc parallel loop independent
    for(i = 1; i < rows; i++)
    {
      vec = shift(vec, x, cols);
      set_matrix_row(mat, i, vec);
    }
    return mat;
}
```

Código 84

- *Splice*: esta función recibe como parámetros un valor mínimo, un valor máximo y una fila. Procede a devolver una fila formada por los datos de la fila pasada como parámetro y limitada por los valores máximo y mínimo. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se

pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
unsigned short* splice(unsigned short* row, int min, int max)
{
    unsigned short* temp = (unsigned short*)calloc(max - min,
sizeof(unsigned short));
    int i;
  #pragma acc parallel loop independent
    for(i = min; i < max; i++)
    {
      temp[i - min] = row[i];
    }
    return temp;
}
```

Código 85

- *Parity_check_matrix*: esta función recibe como parámetro un código MDPC y procede a crear la matriz de paridad. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
bin_matrix parity_check_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    bin_matrix H = make_matrix(code->p, code->p, splice(code->row, 0,
code->p), 1);
    int i;
  #pragma acc parallel loop independent
    for(i = 1; i < code->n0; i++)
    {
      bin_matrix M = make_matrix(code->p, code->p, splice(code->row, i
* code->p, (i + 1) * code->p), 1);
      H = concat_horizontal(H, M);
    }
    end = clock();
    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for H: %f\n", cpu_time_used);

    return H;
}
```

Código 86

- *Generator_matrix*: esta función recibe como parámetro un código MDPC y procede a crear la matriz generadora. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes, se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
bin_matrix generator_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    bin_matrix H = parity_check_matrix(code);



    printf("Construction of G started...\n");
    bin_matrix H_inv = circ_matrix_inverse(make_matrix(code->p, code-
>p, splice(code->row, (code->n0 - 1) * code->p, code->n), 1));

    bin_matrix H_0 = make_matrix(code->p, code->p, splice(code->row,
0, code->p), 1);
    bin_matrix Q = transpose(matrix_mult(H_inv,  H_0));

    bin_matrix M;
    int i;
  #pragma acc parallel loop independent
    for(i = 1; i < code->n0 - 1; i++)
    {
      M = make_matrix(code->p, code->p, splice(code->row, i * code->p,
(i + 1) * code->p), 1);
      M = transpose(matrix_mult(H_inv, M));
      Q = concat_vertical(Q, M);
    }
    bin_matrix I = mat_init(code->k, code->k);
    make_indentity(I);
    bin_matrix G = concat_horizontal(I, Q);

    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for G: %f\n", cpu_time_used);
    printf("Generator matrix generated....\n");
    return G;
}
```

Código 87

- *Get_max*: esta función recibe como parámetros un vector y una longitud y procede a devolver elemento del vector con el máximo valor. Se paraleliza el bucle de la función porque al ser los elementos de la matriz independientes se pueden dividir en múltiples hilos de ejecución y cada uno puede trabajar en secciones diferentes del bucle.

```
int get_max(int* vec, int len)
{
    int max = vec[0];
    int i;
  #pragma acc parallel loop independent
```

```
    for(i = 1; i < len; i++)
    {
        if(vec[i] > max)
        {
            max = vec[i];
        }
    }
    return max;
}
```

Código 88

- *Decode*: esta función recibe como parámetros una matriz binaria y un código MDPC y procede a devolver la matriz decodificada. Los tres primeros bucles se ejecutan en paralelo de forma independiente permitiendo dividir los datos en múltiples hilos permitiendo hacer la ejecución de forma más rápida. El cuarto bucle se ejecuta de forma secuencial ya que los datos dependen del tercer bucle. El quinto bucle se ejecuta en paralelo de forma independiente permitiendo dividir los datos en múltiples hilos para hacer la ejecución de forma más rápida.

```
bin_matrix decode(bin_matrix word, mdpc code)
{
  bin_matrix H = parity_check_matrix(code);
  bin_matrix syn  = matrix_mult(H, transpose(word));
  int limit = 10;
  int delta = 5;
  int i,j,k,x;

  #pragma acc parallel loop independent
    for(i = 0; i < limit; i++)
    {

        int unsatisfied[word->cols];
    #pragma acc parallel loop independent
        for(x = 0; x < word->cols; x++)
        {
            unsatisfied[x] = 0;
        }
    #pragma acc parallel loop independent
        for(j = 0; j < word->cols; j++)
        {
      #pragma acc loop seq
            for(k = 0; k < H->rows; k++)
            {
                if(get_matrix_element(H, k, j) == 1)
                {
                    if(get_matrix_element(syn, k, 0) == 1)
                    {
                        unsatisfied[j] = unsatisfied[j] + 1;
                    }
```

```
                }
            }
        }
        int b = get_max(unsatisfied, word->cols) - delta;
    #pragma acc parallel loop independent
        for(j = 0; j < word->cols; j++)
        {
            if(unsatisfied[j] >= b)
            {
                set_matrix_element(word, 0, j,
(get_matrix_element(word, 0, j) ^ 1));
                syn = add_matrix(syn, mat_splice(H, 0, H->rows - 1, j,
j));
            }
        }
        if(is_zero_matrix(syn))
        {
            return word;
        }
    }
    printf("Decoding failure...\n");
    exit(0);
}
```

Código 89

4.2. Segunda ronda de mejoras

Después de realizar las primeras mejoras sobre la paralelización del código del Classic McEliece se ha realizado una segunda ronda para intentar mejorar la eficiencia del código y mejorar el rendimiento de la ejecución del código.

Las funciones que han sido mejoradas son las siguientes:

4.2.1. matrix.c

- *Set_matrix_row*: la mejora realizada en esta función consiste en añadir una nueva directiva por encima. Esta es 'data present' que asegura que los datos de las matrices A y vec estén presentes y accesibles en la región paralela. Esto garantiza que los datos necesarios estén disponibles para la ejecución paralela.

```
void set_matrix_row(bin_matrix A, int row, unsigned short* vec)
{
  if(row < 0 || row >= A->rows)
  {
    printf("Row index out of range\n");
    exit(0);
  }

  #pragma acc data present(A, vec)
```

```
  {
    #pragma acc parallel loop independent
    for (int i = 0; i < A->cols; i++)
    {
      set_matrix_element(A, row, i, vec[i]);
    }
  }
}
```

Código 90

- *Matrix_rref*: en esta función se ha añadido una directiva 'data present' que realiza la misma operación que la directiva utilizada en set_matrix_row.

```
bin_matrix matrix_rref(bin_matrix A)
{
  int lead = 0;
  int row_count = A->rows;
  int col_count = A->cols;
  bin_matrix temp = mat_init(row_count, col_count);
  temp = mat_copy(A);

  int r = 0;

  #pragma acc data present(temp)
  {
    while(r < row_count)
    {
      if(mat_element(temp, r, r) == 0)
      {
        int i;
        #pragma acc parallel loop independent
        for(i = r + 1; i < temp->rows; i++)
        {
          if(mat_element(temp, i, r) == 1)
          {
            swap(temp, r, i);
            break;
          }
        }
        if(i == row_count)
        {
          printf("Matix cannot be transformed into row echlon
form...");
          exit(1);
        }
      }
      else
      {
        #pragma acc parallel loop independent
```

```
        for(int i = 0; i < row_count; i++)
        {
          if(mat_element(temp, i, r) == 1 && i != r)
          {
            add_rows(temp, r, i);
          }
        }
        r++;
      }
    }
  }
  return temp;
}
```

Código 91

4.2.2. qc_mdpc.c

- *qc_mdpc_init*: en esta función se ha añadido una directiva simple de parallel loop que paraleliza el bucle while y crea múltiples hilos o unidades de procesamiento para ejecutar las iteraciones en paralelo.

```
mdpc qc_mdpc_init(int n0, int p, int w, int t)
{
    mdpc code;
    code = (mdpc)safe_malloc(sizeof(struct qc_mdpc));
    code->n0 = n0;
    code->p = p;
    code->w = w;
    code->t = t;
    code->n = n0 * p;
    code->r = p;
    code->k = (n0 - 1) * p;
    unsigned seed;
    code->row = (unsigned short*)calloc(n0 * p, sizeof(unsigned short));
    printf("Input seed or -1 to use default seed: ");
    scanf("%u", &seed);
    time_t tx;
    if(seed == -1)
    {
      srand((unsigned) time(&tx));
    }
    else
    {
        srand(seed);
    }

    while(1)
    {
        int flag = 0;
```

```
        int idx;
        #pragma acc parallel loop
        while(flag < w)
        {
            idx = random_val(0, (n0 * p) - 1, seed);
            if(!code->row[idx])
            {
              code->row[idx] = 1;
              flag = flag + 1;
            }
        }
        if((get_row_weight(code->row, (n0 - 1) * p, (n0 * p)-1)) % 2 ==
1)
        {
            break;
        }
        reset_row(code->row, 0, n0 * p);
    }
    printf("MDPC code generated....\n");
    return code;
}
```

Código 92

- *get_row_weight*: en esta función se ha modificado la directiva anterior. Se ha eliminado la parte de 'independent' y se ha añadido la parte de 'gang vector'. Esta parte de la directiva indica que cada "gang" se divide en "vector" unidades, que son básicamente los hilos individuales o unidades de procesamiento que realizan las operaciones vectoriales.

```
int get_row_weight(unsigned short* row, int min, int max)
{
    int weight = 0;
    int i;

    #pragma acc parallel loop gang vector
    for(i = min; i < max + 1; i++)
    {
        if(row[i] == 1)
        {
            weight++;
        }
    }
    return weight;
}
```

Código 93

- *reset_row*: en esta función se han realizado los mismos cambios que en la función "get_row_weight".

```
void reset_row(unsigned short* row, int min, int max)
```

```
{
    int i;

    #pragma acc parallel loop gang vector
    for(i = min; i < max + 1; i++)
    {
      row[i] = 0;
    }
}
```

Código 94

- *parity_check_matrix*: en esta función se ha añadido una nueva directiva llamada 'data'. Esta directiva asegura que la matriz H esté disponible para la copia de salida después de que finalice la región paralela. También crea una variable i en el dispositivo (GPU) para que esté disponible en la región paralela.

```
bin_matrix parity_check_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    bin_matrix H = make_matrix(code->p, code->p, splice(code->row, 0,
code->p), 1);
    int i;

    #pragma acc data copyout(H) create(i)
    {
        #pragma acc parallel loop independent reduction(concat:H)
        for (i = 1; i < code->n0; i++)
        {
            bin_matrix M = make_matrix(code->p, code->p, splice(code-
>row, i * code->p, (i + 1) * code->p), 1);
            H = concat_horizontal(H, M);
        }
    }


    end = clock();
    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for H: %f\n", cpu_time_used);

    return H;
}
```

Código 95

- *generator_matrix*: en esta función se ha añadido una nueva cláusula en la directiva que tiene. La cláusula 'reduction (concat: Q)' especifica que la variable Q se reducirá (concatenará) de manera segura en paralelo. Esto significa que cada hilo calcula una parte del resultado final y, al final del bucle, los resultados

parciales se combinan para formar el resultado global de la concatenación de matrices.

```
bin_matrix generator_matrix(mdpc code)
{
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    bin_matrix H = parity_check_matrix(code);

    printf("Construction of G started...\n");
    bin_matrix H_inv = circ_matrix_inverse(make_matrix(code->p, code->p,
splice(code->row, (code->n0 - 1) * code->p, code->n), 1));
    bin_matrix H_0 = make_matrix(code->p, code->p, splice(code->row, 0,
code->p), 1);
    bin_matrix Q = transpose(matrix_mult(H_inv,  H_0));

    bin_matrix M;
    int i;

    #pragma acc parallel loop independent reduction(concat:Q)
    for(i = 1; i < code->n0 - 1; i++)
    {
      M = make_matrix(code->p, code->p, splice(code->row, i * code->p, (i
+ 1) * code->p), 1);
      M = transpose(matrix_mult(H_inv, M));
      Q = concat_vertical(Q, M);
    }
    bin_matrix I = mat_init(code->k, code->k);
    make_indentity(I);
    bin_matrix G = concat_horizontal(I, Q);

    end = clock();
    cpu_time_used = ((double) (end - start))/ CLOCKS_PER_SEC;
    printf("Time for G: %f\n", cpu_time_used);
    printf("Generator matrix generated....\n");
    return G;
}
```

Código 96

- *get_max*: en esta función se han realizado las mismas mejoras que en la función "generator_matrix".

```
int get_max(int* vec, int len)
{
    int max = vec[0];
    int i;

    #pragma acc parallel loop independent reduction(max)
    for(i = 1; i < len; i++)
```

```
    {
        if(vec[i] > max)
        {
            max = vec[i];
        }
    }
    return max;
}
```

Código 97

- *Decode*: en esta función se han realizado mejoras en la directiva del segundo for y son las mismas mejoras de la función "generator_matrix". Se ha eliminado la directiva del tercer for porque con la mejora añadida a la directiva del segundo for la directiva del tercer for ya no es necesaria. Además, se ha eliminado la directiva del primer for.

```
bin_matrix decode(bin_matrix word, mdpc code)
{
  bin_matrix H = parity_check_matrix(code);
  bin_matrix syn  = matrix_mult(H, transpose(word));
  int limit = 10;
  int delta = 5;
  int i,j,k,x;


    for(i = 0; i < limit; i++)
    {
        int unsatisfied[word->cols];
        #pragma acc parallel loop independent
        for(x = 0; x < word->cols; x++)
        {
            unsatisfied[x] = 0;
        }

        #pragma acc parallel loop independent
reduction(+:unsatisfied[:word->cols])
        for(j = 0; j < word->cols; j++)
        {
            for(k = 0; k < H->rows; k++)
            {
                if(get_matrix_element(H, k, j) == 1)
                {
                    if(get_matrix_element(syn, k, 0) == 1)
                    {
                        unsatisfied[j] = unsatisfied[j] + 1;
                    }
                }
            }
        }
```

```
        int b = get_max(unsatisfied, word->cols) - delta;
        #pragma acc parallel loop independent
        for(j = 0; j < word->cols; j++)
        {
            if(unsatisfied[j] >= b)
            {
                set_matrix_element(word, 0, j, (get_matrix_element(word,
0, j) ^ 1));
                syn = add_matrix(syn, mat_splice(H, 0, H->rows - 1, j,
j));
            }
        }

        if(is_zero_matrix(syn))
        {
            return word;
        }
    }
    printf("Decoding failure...\n");
    exit(0);
}
```

Código 98

5. Resultados

5.1 Resultados obtenidos con las distintas versiones

Uno de los objetivos principales de este trabajo era realizar una modificación del código original de la implementación del algoritmo McEliece de tal manera que se pudiese compilar y ejecutar el proceso de cifrado y descifrado de manera más rápida y eficiente. Gracias a la utilización del lenguaje de programación de OpenACC se ha obtenido una mejora en el rendimiento muy significativa realizando una prueba en la versión original y en la versión mejorada utilizando los mismos parámetros y la misma cadena de texto. Pero no todo fue sencillo ya que muchas de las funciones que había que paralelizar tenían un código específico que no permitía el uso de código OpenACC por lo tanto había que modificar gran parte del código de algunas de las funciones para poder adaptarlas al uso de código OpenACC y eso retrasó las pruebas porque había que localizar exactamente cuáles eran las funciones que daban problema y luego modificarlas.

Las pruebas se realizaron en el emulador de GPU que proporciona Google Colab. Google Colab es una herramienta proporcionada por Google Research que permite a cualquier usuario escribir y ejecutar código en el navegador, es decir, todo trabajo realizado se guarda en la nube de Google drive permitiendo trabajar desde cualquier dispositivo con conexión a internet. Por otro lado, tiene funciones de configuración que permiten cambiar el entorno de ejecución, como usar un emulador de GPU que permite conectarse de forma remota con una GPU donde se puede ejecutar el código y realizar las pruebas al usuario. En general es una herramienta muy útil para la creación y ejecución de código de forma rápida y eficiente. En nuestro caso utilizamos un emulador de GPU de Google Colab que tiene 3 GB de memoria RAM y emplea un procesador AMD Ryzen 7 3700U de cuatro núcleos a una velocidad de 2,3 GHz.

Para poder realizar la prueba hay que seguir los siguientes pasos:

1. Primero hay que ejecutar el código necesario para poder cargar los archivos .c y .h al entorno de drive.

   ```
   from google.colab import files
   uploaded = files.upload()
   ```

2. Para poder compilar el código de la versión original hay que introducir el siguiente comando:

   ```
   !gcc -o test test.c BinaryText.c matrix.c mceliece.c
   qc_mdpc.c TextBinary.c utility.c
   ```

3. Cuando haya finalizado la compilación para iniciar la prueba hay que ejecutar el siguiente comando:

   ```
   !./test
   ```

4. Al comenzar la ejecución el usuario introduce su mensaje y puede elegir si introduce un valor a la semilla o si usa el valor de –1, que es el que se usa por defecto. Después simplemente hay que esperar a que finalice la ejecución.

5. Por otro lado, para poder compilar el código de la versión paralelizada hay que introducir el siguiente comando:

```
!gcc -g -O3 -B -fopenacc -o test test.c BinaryText.c
BinaryText.h mceliece.c mceliece.h qc_mdpc.c qc_mdpc.h
matrix.c matrix.h TextBinary.c TextBinary.h utility.c
utility.h
```

6. Cuando haya finalizado la compilación para iniciar la prueba hay que ejecutar el comando usado en el punto 3.

7. Al comenzar la ejecución hay que realizar los pasos utilizados previamente del punto 4.

En relación con el cálculo de los tiempos, los ficheros de la implementación descargada utilizaban la función clock de la biblioteca time.h para medir el tiempo de ejecución total y del cifrado y descifrado. Y en cuanto al porcentaje aproximado de la cantidad de código paralelizado estimamos que ha sido alrededor de un 73%.

Podemos observar en la Tabla 4 cómo obtenemos una mejora del 92% respecto al código sin paralelización en comparación con el de la primera versión de la implementación paralelizada. Eso es un resultado que mejora significativamente la versión original haciendo la primera versión mejorada más rápida y eficiente. La mejora que se obtiene entre la versión original y la segunda versión de la implementación paralelizada es del 98%. El tiempo que se obtiene después de ejecutar la segunda versión paralelizada es 15% menor que el obtenido tras ejecutar la primera versión paralelizada.

En la siguiente tabla se muestra una comparación entre la versión original y la versión mejorada ofreciendo datos como la longitud de texto a cifrar, los tiempos de realizar el cifrado y descifrado y por último el tiempo total que se ha tardado en realizar el proceso.

	Con paralelización OpenACC (implementación segunda versión)	**Con paralelización OpenACC (implementación primera versión)**	**Sin paralelización**
Longitud del texto a cifrar	1320 caracteres	1320 caracteres	1320 caracteres
Tamaño de las claves	1000 bits clave pública y 2000 bits clave privada	1000 bits clave pública y 2000 bits clave privada	1000 bits clave pública y 2000 bits clave privada

Tiempo de crear la matriz H y realizar el cifrado	0.021589 segundos	0.029149 segundos	0.239253 segundos
Tiempo de crear la matriz G y realizar el descifrado	0.643661 segundos	0.732339 segundos	86.78445 segundos
Tiempo total de la ejecución del proceso	0.698995 segundos	0.822698 segundos	87.023751 segundos

Tabla 4: Comparación tiempos con/sin paralelización.

En nuestra opinión, la mejora del rendimiento del nuevo código es tan elevada debido a la gran cantidad de código que hemos paralelizado (74%), ya que paralelizar consiste en dividir una tarea en varias subtareas que se ejecutan al mismo tiempo en diferentes procesadores, permitiendo que al paralelizar tanto código se ahorre mucho tiempo de ejecución.

5.2. Otras investigaciones

Para poder confirmar que los resultados obtenidos en las dos versiones de paralelización de la implementación del QC-MDPC McEliece explicadas en el capítulo 4 tienen sentido, hemos comparado estos con otros resultados obtenidos de otras investigaciones y pruebas realizadas sobre el mismo algoritmo cuántico.

En total hemos hecho comparaciones con cuatro investigaciones. Dos de ellas son muy similares a la nuestra, utilizan los mismos mecanismos y sistemas a la hora de realizar los procesos de creación de las claves, ejecución del proceso de cifrado y descifrado. Las diferencias respecto a nuestra investigación son las siguientes:

1. Se han utilizado distintos parámetros iniciales, en nuestro caso los parámetros iniciales son n0 = 2, t = 20, p = 1000 y w = 45. Los parámetros iniciales utilizados por las otras implementaciones son n0 = 2, t = 84, p = 4801 y w = 90. Lo único común que tienen las tres implementaciones son los dos bloques circulares fijos. La matriz de paridad en nuestra implementación tiene un peso de fila w = 45 y en las otras dos se duplica ese peso. El texto plano en nuestra implementación es de 1000 bits y se crea un texto cifrado de 2000 bits al cual se le añaden t = 20 errores, en cambio en las otras dos implementaciones el texto plano es de 4801 bits lo cual permite crear un texto cifrado de 9602 bits al cual se le añaden t = 84 errores. En general los tamaños de los parámetros iniciales utilizados en nuestra implementación son más pequeños respecto a los tamaños de los parámetros iniciales utilizados en las otras dos implementaciones.
2. Otra diferencia entre las implementaciones son los dispositivos de ejecución. En nuestra implementación se utiliza un emulador de GPU de Google Colab que tiene 3 GB de memoria RAM y emplea un AMD Ryzen 7 3700U de cuatro núcleos a una velocidad de 2,3 GHz. Los dispositivos utilizados por las otras implementaciones son dos FPGAs Xilinx Spartan 6 y Virtex 6 que están equipadas con doble memoria de BRAM de capacidad de 18/36 Kb. La diferencia en los

dispositivos es importante ya que las FPGAs están hechas para realizar pruebas de este tipo en comparación con un ordenador y emulador cualquiera.

El objetivo de las tres implementaciones es mejorar el rendimiento de las ejecuciones de los procesos de cifrado y descifrado. Por nuestra parte se ha realizado una versión mejorada mediante código OpenACC realizando una paralelización del código en GPU. Las otras implementaciones han aprovechado las ventajas de las FPGAs para poder ejecutar los procesos de manera más eficiente. Teniendo en cuenta las diferencias entre las implementaciones y el tamaño de los textos usados para realizar las pruebas, se han obtenido los siguientes resultados [18] [19].

	Con paralelización GPU (nuestro)	**Virtex-6 XC6VLX240T**	**Spartan-6 XC6SLX4**
Longitud del texto a cifrar	1320 caracteres (La máquina no acepta más de cierta cantidad de caracteres por falta de espacio de memoria)	4800 caracteres	4801 caracteres
Tamaño de las claves	1000 bits clave pública y 2000 bits clave privada	4801 bits clave pública y 9602 bits clave privada	4801 bits clave pública y 9602 bits clave privada
Tiempo de crear la matriz H y realizar el cifrado	22 ms	2.2 ms	3.4 ms
Tiempo de crear la matriz G y realizar el descifrado	644 ms	13.4 ms	23 ms
Tiempo total de la ejecución del proceso	698 ms	18 ms	30 ms

Otra de las investigaciones [20] sobre el Classic McEliece que hemos buscado para comparar con nuestra implementación tiene las siguientes diferencias respecto a nuestra implementación:

1. Utiliza una FPGA con un procesador ARM Cortex-A53 con una velocidad de 1.1 GHz con un coprocesador A14 con una interfaz lite. El sistema completo ha sido implementado sobre una FPGA Xilinx Zynq UltraScale+. Esta implementación es capaz de cifrar y descifrar textos de 8192 caracteres en 47,39 ms y utiliza claves de 2 millones de bytes.
2. Utiliza dos algoritmos distintos a los de nuestra implementación, el de Patterson y una versión alternativa al algoritmo de Euclides. El algoritmo de Patterson se usa para localizar y computar el error principal de una clave con

errores y el algoritmo de Euclides se encarga de calcular el máximo común divisor entre dos polinomios.

El objetivo de nuestra implementación es mejorar la eficiencia al realizar una paralelización en GPU y la otra implementación se centra en mejorar la eficiencia mejorando la seguridad del cifrado. En la siguiente tabla se mostrarán los resultados entre las dos implementaciones.

	Con paralelización GPU (nuestro)	**Zynq UltraScale+**
Longitud del texto a cifrar	512 caracteres	512 caracteres
Tamaño de las claves	1000 bits clave pública y 2000 bits clave privada	1046739 bytes clave pública y 2093478 bytes clave privada
Tiempo de crear la matriz H y realizar el cifrado	2961 ms	1.5 ms
Tiempo de crear la matriz G y realizar el descifrado	44873 ms	1002.1 ms
Tiempo total de la ejecución del proceso	46490 ms	1010 ms

La última investigación [21] utilizada para comparar con nuestra implementación tiene una modificación híbrida distribuida en dos partes. La primera parte aumenta la tasa de información añadiendo datos en el patrón de errores y la segunda parte disminuye el tamaño de la clave pública utilizando una matriz generatriz que tiene una fila en forma echelon. Al ser un sistema híbrido utiliza distintos métodos para poder generar las claves y para poder realizar los procesos de cifrado y descifrado. En nuestra implementación tenemos dos parámetros iniciales para crear el QC-MDPC McEliece, en cambio en la otra implementación se usan solo dos parámetros aleatorios iniciales para generar las claves. Las pruebas de la implementación híbrida se han realizado sobre un procesador Intel Core 2 y un compilador de Intel con un sistema operativo de 32 bits. El objetivo de la implementación híbrida es mejorar la seguridad frente a los ataques e intentar disminuir el tamaño de las claves para que sea más eficiente. El objetivo es similar al de nuestra implementación que intenta mejorar la eficiencia realizando una paralelización en GPU. En la siguiente tabla se muestran una comparación entre las dos implementaciones.

	Con paralelización GPU (nuestro)	Intel Core 2
Longitud del texto a cifrar	1320 caracteres	No se sabe
Tamaño de las claves	1000 bits clave pública y 2000 bits clave privada	32000 bytes clave pública y 64000 bytes clave privada
Tiempo de crear la matriz H y realizar el cifrado	22 ms	2.19 ms
Tiempo de crear la matriz G y realizar el descifrado	644 ms	75 ms
Tiempo total de la ejecución del proceso	698 ms	79 ms

Las razones por las que creemos que este algoritmo presenta este rendimiento tan bueno son las siguientes:

- Uso de códigos QC-MDPC, permiten realizar cifrados utilizando implementaciones con una huella de recursos más pequeña de lo normal.
- Permite reducir el tamaño de las claves públicas, lo que permite crear diseños más pequeños.
- Uso de hardware, FPGAs y implementaciones.

Como se puede observar los resultados que hemos obtenido en nuestra implementación se asemejan a los resultados que se han obtenido en otras implementaciones usando otras técnicas y métodos para obtenerlos, ya sea con otro hardware, usando otros algoritmos o usando otros parámetros de mayor tamaño.

5.3. Comparaciones adicionales

En este aparatado abordaremos una comparación entre nuestra implementación con y sin paralelizar y los algoritmos tradicionales RSA y DH, para ver cuánto oscilan los tiempos de ejecución partiendo de un mismo texto de 100 caracteres. Para ello hemos utilizado el siguiente texto: Hoy es lunes. y mi casa está a media hora de Madrid, por lo que tendremos que coger el bus para ir.

Para nuestras implementaciones hemos realizado una ejecución siguiendo las instrucciones descritas n en la sección 5.1 de este documento, para ello introduciremos dicho texto y un valor de semilla igual a 1. Para comprobar que nuestro código paralelizado funciona tenemos que comprobar que al ejecutarlo se obtiene el mismo texto cifrado que con el algoritmo McEliece original, para ello se ha realizado un volcado

a mano en un fichero binario. Posteriormente han sido comparados con un programa de comparación de ficheros (DiffMerge).

En el caso de RSA y DH se ha utilizado una máquina virtual de Debian (32-bit). Con esta máquina virtual se ha empleado OpenSSL para la creación de las claves y cifrado y descifrado del texto. En el caso de DH se muestra el tiempo que tarda únicamente en crear las claves necesarias para llevar a cabo el proceso con y sin generación de parámetros, ya que este algoritmo no realiza cifrado/descifrado. Para realizar estas comparativas y ver los tiempos resultantes hemos realizado los siguientes scripts:

- RSA

```
start=`date +%s%N`
openssl pkeyutl -encrypt -pubin -inkey rsapubkey.pem -in texto.txt -out enc.bin
openssl pkeyutl -decrypt -inkey rsakey.pem -in enc.bin -out plainagain.txt
end=`date +%s%N`
echo `expr $end - $start`
```

  En este script, primero se inicia la cuenta del reloj. Luego, se crea la clave privada y la clave pública a partir de la privada. Y por último se muestra el tiempo.

```
start=`date +%s%N`
openssl genpkey -algorithm rsa -out rsakey.pem
openssl pkey -pubout -in rsakey.pem -out rsapubkey.pem
end=`date +%s%N`
echo `expr $end - $start`
```

  En este script, primero se inicia la cuenta del reloj. Luego se cifra y se descifra. Y por último se muestra el tiempo.

- DH (generando parámetros)

```
start=`date +%s%N`
openssl genpkey -genparam -algorithm dh -out dhparam.pem
openssl genpkey -paramfile dhparam.pem -out dhkey1.pem
openssl pkey -pubout -in dhkey1.pem -out dhpubkey1.pem
openssl genpkey -paramfile dhparam.pem -out dhkey2.pem
openssl pkey -pubout -in dhkey2.pem -out dhpubkey2.pem
openssl pkeyutl -derive -inkey dhkey1.pem -peerkey dhpubkey2.pem -out secret1
openssl pkeyutl -derive -inkey dhkey2.pem -peerkey dhpubkey1.pem -out secret2
cmp secret1 secret2
end=`date +%s%N`
echo `expr $end - $start`
```

  En este script, primero se inicia la cuenta del reloj. Luego, se crean los parámetros que compartirán ambos usuarios, las claves privadas a partir de los parámetros y las claves públicas a partir de las claves privadas. Después se genera la clave simétrica que usarán ambos usuarios para cifrar. Y por último, se muestra el tiempo.

- DH (parámetros ya generados)

```
start=`date +%s%N`
openssl genpkey -paramfile dhparam.pem -out dhkey1.pem
openssl pkey -pubout -in dhkey1.pem -out dhpubkey1.pem
openssl genpkey -paramfile dhparam.pem -out dhkey2.pem
openssl pkey -pubout -in dhkey2.pem -out dhpubkey2.pem
openssl pkeyutl -derive -inkey dhkey1.pem -peerkey dhpubkey2.pem -out secret1
openssl pkeyutl -derive -inkey dhkey2.pem -peerkey dhpubkey1.pem -out secret2
cmp secret1 secret2
end=`date +%s%N`
echo `expr $end - $start`
```

  En este script, primero se inicia la cuenta del reloj. Luego se crean las claves privadas a partir de los parámetros (se supone que ya han sido creados) y las claves públicas a partir de las privadas. Después se crea la clave simétrica que usarán ambos usuarios para cifrar. Y, por último, se muestra el tiempo.

ALGORITMO	NUESTRO NO PARALELIZADO	NUESTRO PARALELIZADO	RSA	DH
TIEMPO Generación Claves (s)	3.71763	0.218176	0,62763	385,24985 (con generación de parámetros) 0,21538 (sin generación de parámetros)
TIEMPO CIFRADO Y DESCIFRADO (suma)	0.10131	0.02961	0,05563	NO CIFRA, USADO PARA INTERCAMBIO DE CLAVES

Tabla 5: tiempos diferentes McEliece, RSA y DH

En la Tabla 5, podemos observar que nuestra implementación de McEliece es mucho más rápida que la del algoritmo RSA en la generación de claves y muy parecida a la de Diffie-Hellman. Además, nuestra implementación comparada con la RSA a la hora de cifrar/descifrar es más rápida.

6.Aportaciones y Conclusiones

6.1. Aportaciones

Las aportaciones realizadas por Petar Ivanov durante el desarrollo del Trabajo Fin de Grado han sido la búsqueda, análisis, comparación y pruebas de las implementaciones sobre el algoritmo post-cuántico de Classic McEliece. Este proceso implicó identificar versiones funcionales del algoritmo y comprender a fondo su funcionamiento. Primero realizó una búsqueda exhaustiva para encontrar distintas versiones válidas del algoritmo con instrucciones sobre su funcionamiento. Posteriormente realizó numerosas pruebas entre las distintas implementaciones para comparar los resultados entre ellas y observar qué implementación ofrecía el resultado más eficiente. Una vez escogida la mejor implementación explicó por qué se había escogido esa implementación para realizar las modificaciones sobre ella. Petar procedió a realizar una mejora sobre el código aplicando el lenguaje de programación OpenACC. Después de completar las modificaciones sobre el código, cargó todos los archivos en el emulador de prueba de Google Colab. A través de una serie de comandos logró cargar los ficheros necesarios, compilar el código para la versión paralelizada (optimizada) y la versión original y posteriormente ejecutar el código. Una vez completadas las pruebas necesarias, Petar analizó los tiempos obtenidos para ambas versiones (la optimizada y la original), identificando claramente la mejora de rendimiento lograda en la versión mejorada.

Posteriormente, Petar se enfocó en mejorar aún más el código paralelizado en la versión optimizada de la implementación. Mediante la realización de pruebas y análisis, demostró una vez más una notable mejora en el tiempo respecto a las otras dos versiones y explicó por qué se había producido esa mejora en la segunda versión respecto a las otras.

En una etapa posterior, Petar realizó una detallada comparación entre la implementación mejorada y otras investigaciones similares. Esta comparación sirvió para validar que los resultados obtenidos en la segunda versión mejorada estaban en la misma línea con los logros de otras investigaciones. Además, delineó de manera clara las diferencias entre la implementación mejorada y las otras investigaciones tanto en hardware como en software.

Petar también contribuyó en la explicación de los métodos y funciones que se encuentran en el código de la implementación. Además, creó diagramas que ilustran el flujo de funcionamiento y las llamadas que realiza cada método en el código. También proporcionó una comprensión más profunda de las mejoras realizadas en el código original, incluyendo el funcionamiento del código paralelizado con lenguaje OpenACC.

Finalmente, Petar contribuyó activamente en la búsqueda de información sobre la última tanda de algoritmos del concurso organizado por NIST. Además, se encargó de la creación del resumen, la explicación de los objetivos y supervisó de cerca la planificación

durante todo el período de desarrollo del Trabajo Fin de Grado. También elaboró una tabla detallada que resume las actividades realizadas en cada período de trabajo.

Entre las aportaciones realizadas por Víctor Moreno durante el desarrollo del Trabajo Fin de Grado se encuentran la creación del resumen, la introducción, la explicación de los objetivos, el estado del arte y supervisó la planificación durante todo el período de desarrollo del Trabajo Fin de Grado mediante la elaboración de una tabla detallada que resume las actividades realizadas en cada período de trabajo. Además, se encargó de la búsqueda y análisis de los algoritmos NIST para la redacción de cada uno de ellos y posteriormente poder elegir el algoritmo en el que centrarnos. Se encargó activamente de la corrección, revisión y mejora constante de la redacción y traducción de la memoria.

Asimismo, se encargó de la búsqueda, análisis, comparación y pruebas de las implementaciones sobre el algoritmo post-cuántico de Classic McEliece. Este proceso consistió como hemos dicho en el párrafor anterior en identificar versiones funcionales del algoritmo y comprender a fondo su funcionamiento. Primero realizó una búsqueda exhaustiva para encontrar distintas versiones del algoritmo que funcionasen y que tuvieran instrucciones sobre cómo funcionan esas implementaciones. Posteriormente realizó numerosas pruebas entre las distintas implementaciones para comparar los resultados entre ellas y observar que implementación ofrecía el resultado más eficiente. Una vez escogida la mejor implementación explicó porque se había escogido esa implementación para realizar las mejoras sobre ella. Asistió a Petar en algunas partes de la paralelización y llevó a cabo junto a Petar la toma de los tiempos obtenidos para ambas versiones (la mejorada y la original), identificando claramente la mejora de rendimiento lograda en la versión mejorada, tanto la primera como la segunda.

En una etapa posterior, Víctor realizó una detallada comparación entre la implementación descargada, la mejorada, RSA y DH. Para ello, tuvo que descargarse una máquina virtual en la que realizó scripts de bash para poder medir el tiempo empleado por los algoritmos RSA y DH. Por otro lado, ejecutó las implementaciones en Google Collab y mediante un texto de igual tamaño identificó las diferencias en los tiempos de generación de claves y cifrado que surgían entre ellas.

Finalmente, Víctor también contribuyó en la redacción de los métodos y funciones que se encuentran en el código de la implementación. También proporcionó una comprensión más profunda de las mejoras realizadas en el código original, incluyendo el funcionamiento del código paralelizado con lenguaje OpenACC.

6.2. Conclusiones

Este Trabajo de Fin de Grado se ha centrado en el estudio de la importancia de la computación cuántica, los distintos criptosistemas del programa lanzado por NIST y especialmente en el Classic McEliece.

Hemos logrado implementar de forma paralela uno de los algoritmos finalistas de la convocatoria del National Institute of Standars and Technology de Estados Unidos (NIST) y modificar el código para que tome mensajes por consola y los convierta a binario para

su posterior cifrado y descifrado (en la implementación original eran mensajes predefinidos en binario de un tamaño estándar). De estas optimizaciones hemos conseguido sacar una mejora de rendimiento muy buena.

Además, hemos sido capaces de analizar y comparar el rendimiento de las distintas implementaciones buscadas y estudiadas anteriormente. Asimismo, las hemos analizado y comparado con otras investigaciones y con RSA y DH. Este trabajo de lectura y análisis ha servido de utilidad para ser más conscientes del gran potencial que tiene el sistema de McEliece, sistema que se ha estudiado durante muchos años y siempre ha obtenido buenos resultados a la hora de proporcionar seguridad.

Para concluir, la seguridad de la información de las comunicaciones tiene una gran importancia ya que cualquier persona realiza multitud de comunicaciones diarias las cuales contienen en ocasiones información sensible y/o privada. La amenaza que supone la computación cuántica va siendo cada vez más cercana debido al crecimiento cada vez mayor del rendimiento de los ordenadores. Por ello, la criptografía post-cuántica como futura solución debería cobrar más importancia, es decir, se deberían dedicar más recursos en la búsqueda y/o mejora de algoritmos para poder lograr una mayor dificultad a la hora de poder romperlos.

6.3. Objetivos futuros

Como hemos dicho, la criptografía no ha hecho nada más que comenzar. Se trata de un terreno en desarrollo al que le queda recorrido.

Nosotros hemos planteado estudiar el McEliece del que hemos sacado buenas conclusiones. Sin embargo, es posible que haya algunos mejores y convenga estudiarlos o que después de la entrega de este Trabajo de Fin de Grado, salgan o se barajeen mejores opciones.

Por lo que, habría que seguir estudiando aquellos candidatos del NIST y mejorarlos para ver cuál podría llegar a ser su verdadero potencial.

6. Contributions and Conclusions

6.1. Contributions

The contributions made by Petar Ivanov during the development of the Bachelor's Thesis involved the search, analysis, comparison, and testing of implementations for the post-quantum Classic McEliece algorithm. This process entailed identifying functional versions of the algorithm and gaining a deep understanding of its operation. Petar initially conducted an exhaustive search to find different versions of the algorithm that were functional and had instructions on how these implementations worked. Subsequently, he carried out numerous tests among the different implementations to compare the results and determine which implementation provided the most efficient outcome. Once the best implementation was chosen, he explained why it was selected for further improvements. Petar then proceeded to enhance the code using the OpenACC programming language. After completing the code modifications, he uploaded all the files to the Google Colab emulator. Through a series of commands, he managed to load the necessary files, compile the code for the parallelized (improved) version and the original version, and subsequently run the code. After completing the necessary tests, Petar analyzed the times obtained for both versions (the optimized and the original), clearly identifying the performance improvement achieved in the enhanced version.

Subsequently, Petar focused on further enhancing the parallelized code in the improved version of the implementation. Through testing and analysis, he once again demonstrated a significant improvement in the time compared to the other two versions and explained why this improvement had occurred in the second version compared to the others.

In a later stage, Petar conducted a detailed comparison between the improved implementation and similar research efforts. This comparison served to validate that the results obtained in the second improved version were in line with the achievements of other research endeavors. Furthermore, he clearly outlined the distinctions between the improved implementation and the other research efforts, both in terms of hardware and software.

Petar also contributed to explaining the methods and functions found in the implementation code. Additionally, he created diagrams illustrating the flow of operation and the calls made by each method in the code. He also provided a deeper understanding of the enhancements made to the original code, including the operation of the parallelized code using OpenACC language.

Finally, Petar actively contributed to the search for information about the latest batch of algorithms in the competition organized by NIST. Additionally, he was responsible for creating the summary, explaining the objectives, and closely overseeing the planning

throughout the entire development period of the Bachelor's Thesis. He also compiled a detailed table summarizing the activities performed in each work period.

Among the contributions made by Víctor Moreno during the development of the Bachelor's Thesis, we find the creation of the abstract, introduction, explanation of the objectives, the state of the art, and the supervision of the planning throughout the entire development period of the Bachelor's Thesis by creating a detailed table summarizing the activities carried out in each work period. Additionally, he was responsible for the search and analysis of NIST algorithms for drafting each of them and subsequently being able to choose the algorithm to focus on. He actively took care of the correction, review, and constant improvement of the writing and translation of the report.

Furthermore, he was in charge of the search, analysis, comparison, and testing of implementations of the post-quantum Classic McEliece algorithm. This process, as mentioned in the previous paragraph, involved identifying functional versions of the algorithm and thoroughly understanding its operation. He first conducted an exhaustive search to find different versions of the algorithm that worked and had instructions on how these implementations functioned. He then conducted numerous tests among the different implementations to compare the results between them and observe which implementation offered the most efficient result. Once the best implementation was chosen, he explained why that implementation had been chosen for further improvements. He assisted Petar in some parts of parallelization and together with Petar, recorded the times obtained for both versions (the improved and the original), clearly identifying the performance improvement achieved in the improved version, both the first and the second.

In a later stage, Víctor conducted a detailed comparison between the downloaded implementation, the improved one, RSA, and DH. To do this, he had to download a virtual machine in which he ran bash scripts to measure the time used by the RSA and DH algorithms. On the other hand, he executed the implementations in Google Colab and, using a text of the same size, identified the differences in key generation and encryption times that arose between them.

Finally, Víctor also contributed to the writing of the methods and functions found in the implementation code. He provided a deeper understanding of the improvements made in the original code, including the operation of the parallelized code using the OpenACC language.

6.2. Conclusions

This Bachelor's Thesis has focused on the study of the importance of quantum computing, the different cryptosystems from the program launched by the National Institute of Standards and Technology (NIST), and especially on Classic McEliece.

We have successfully implemented one of the finalist algorithms from the call by the National Institute of Standards and Technology (NIST) in a parallel manner. Additionally, we modified the code to accept console input messages and convert them to binary for

subsequent encryption and decryption (in the original implementation, messages were predefined in binary of a standard size). These enhancements have resulted in a significant performance improvement.

Furthermore, we have been able to analyze and compare the performance of the various implementations that were researched and studied earlier. Additionally, we evaluated them in comparison to other researches and alongside RSA and DH. This process of reading and analysis has been instrumental in making us more aware of the tremendous potential that the McEliece system holds, a system that has been studied for many years and has consistently demonstrated strong security capabilities.

In conclusion, the security of communication information holds great importance, as individuals engage in numerous daily communications that often contain sensitive and/or private information. The threat posed by quantum computing is becoming increasingly imminent due to the ever-growing performance of computers. Therefore, post-quantum cryptography, as a future solution, should be given greater emphasis. This means dedicating more resources to the search and/or improvement of algorithms to achieve greater difficulty in breaking them.

6.3. Future Objectives

As we have mentioned, cryptography is just getting started. It's a field in development with a long way to go.

We've chosen to study McEliece and have drawn positive conclusions from it. However, there might be even better options that are worth exploring. It's possible that after the completion of this Bachelor's Thesis, new and better alternatives emerge or are considered.

Therefore, it would be essential to continue studying the NIST candidates and refine them to understand their true potential.

Bibliografía

[1] «vanguardia,» [En línea]. Available: https://www.lavanguardia.com/tecnologia/20220805/8449725/rompen-algoritmo-cifrado-prueba-computadoras-cuanticas-ordenador-barato-pmv.html.

[2] «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/DSA.

[3] «nist,» [En línea]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography.

[4] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson y D. Smith-Tone, «NIST Technical Series Publications,» [En línea]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf.

[5] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, D. Urbanik, G. Pereira, K. Karabina y A. Hutchinson, «SIKE,» [En línea]. Available: https://sike.org/.

[6] N. Aragon, P. L. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, C. Aguilar Melchor, R. Misoczki, E. Persichetti, J. Richter-Brockmann, N. Sendrier, J.-P. Tillich, V. Vasseur y G. Zémor, «BIKE,» [En línea]. Available: https://bikesuite.org/.

[7] «hal,» [En línea]. Available: https://hal.science/hal-01671903/document.

[8] D. J. Bernstein, T. Chou, C. Cid, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, N.

Sendrier, J. Szefer, C. Jung Tjhai, M. Tomlinson y W. Wang, «Classic McEliece,» [En línea]. Available: https://classic.mceliece.org/.

[9] «wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/McEliece_cryptosystem.

[10] C. Aguilar Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J.-C. Deneuville, A. Dion, P. Gaborit, J. Lacan, E. Persichetti, J.-M. Robert, P. Véron y G. Zémor, «HQC,» [En línea]. Available: http://pqc-hqc.org/index.html.

[11] «IME,» [En línea]. Available: https://www.ime.usp.br/~tpaiva/papers/PaivaTerada_SAC2019_a_timing_attack_against_hqc.pdf.

[12] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, D. Stehle y J. Ding, «Kyber,» [En línea]. Available: https://pq-crystals.org/.

[13] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, Z. Zhang, T. Saito, T. Yamakawa y K. Xagawa, «NTRU,» [En línea]. Available: https://ntru.org/.

[14] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, F. Vercauteren, J. M. Bermudo Mera, M. Van Beirendonck y A. Basso, «SABER:LWR-based KEM,» [En línea]. Available: https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[15] «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n).

[16] «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/OpenACC.

[17] «hmong,» [En línea]. Available: https://hmong.es/wiki/Binary_Goppa_code.

[18] I. von Maurich y T. Guneysu, «Lightweight Code-based Cryptography: QC-MDPC».

[19] I. von Maurich, T. Oder y T. Guneysu, «Implementing QC-MDPC McEliece Encryption».

[20] M. López García y E. Cantó Navarro, «Hardware-Software Implementation of a McEliece».

[21] B. Biswas y N. Sendrier, «McEliece Cryptosystem Implementation: Theory».