Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Criptosistema de McEliece

McEliece Cryptosystem

Marta García Luis

La Laguna, *29* de *mayo* de 2021

Dña. **Pino Teresa Caballero Gil**, con N.I.F. 45.534.310-Z Catedrática de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

C E R T I F I C A

Que la presente memoria titulada:

“Criptosistema de McEliece”

ha sido realizada bajo su dirección por Dña. **Marta García Luis**,

con N.I.F. 46.246.397-K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de junio de 2021

Agradecimientos

Agradecer tanto a la tutora Pino Caballero Gil, como a las profesoras Jezabel Molina Gil e Irene Márquez Corbella por sus aportaciones y ayuda durante el desarrollo del proyecto. Así como a mi familia y amistades por el apoyo a lo largo de toda la carrera.

Licencia

Resumen

La llegada de los ordenadores cuánticos implica un cambio de paradigma. A diferencia de los ordenadores clásicos, en estos se emplean qubits en lugar de bits, haciendo uso de las propiedades de la mecánica cuántica, lo que permitiría resolver problemas que hasta ahora eran irresolubles, es decir, resolver problemas matemáticos de gran complejidad que no podrían ser resueltos en tiempo polinomial con ordenadores clásicos. Muchas empresas como IBM, Microsoft, Google o Amazon están trabajando para ser los primeros en conseguir elaborar este tipo de tecnología y ofrecer el servicio a sus clientes.

Entre las implicaciones que estos ordenadores tendrán podemos destacar su uso en campos como el financiero, o la informática, cobrando este último gran importancia ya que los algoritmos criptográficos de clave pública que se emplean en la actualidad pasarán a ser vulnerables. Por ello se investigan algoritmos post-cuánticos que sean resistentes tanto a los ordenadores clásicos como a los cuánticos.

En este trabajo se plantea, concretamente, el estudio del sistema de McEliece junto con su versión mejorada, el sistema de Niederreiter, pues ambos son ejemplos de uno de los tipos de criptosistemas post-cuánticos con mejores expectativas de futuro.

Palabras clave: criptografía, computación cuántica, McEliece, Niederreiter

Abstract

The arrival of quantum computers implies a paradigm shift. Unlike classical computers, these use qubits instead of bits, making use of the properties of quantum mechanics, which would allow solving problems that until now were unsolvable, that is, solving highly complex mathematical problems that could not be solved. be solved in polynomial time with classical computers. Many companies such as IBM, Microsoft, Google or Amazon are working to be the first to develop this type of technology and offer the service to their customers.

Among the implications that these computers will have, we can highlight their use in fields such as finance, or computing, the latter being of great importance since the public key cryptographic algorithms used today will become vulnerable. For this reason, post-quantum algorithms that are resistant to both classical and quantum computers are being investigated.

In this work, the study of the McEliece system together with its improved version, the Niederreiter system, is specifically proposed, since both are examples of one of the types of post-quantum cryptosystems with better future prospects.

Keywords: cryptography, quantum computing, McEliece, Niederreiter

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1. Introducción al trabajo

1.1. Motivación

La criptografía post-cuántica resulta un ámbito de gran interés debido no solo a la importancia que tiene la seguridad informática sino a los continuos progresos que tienen lugar en el campo de la computación cuántica.

Uno de los sistemas criptográficos mas utilizados en la actualidad es el RSA [1], desarrollado en 1977 por Rivest, Shamir y Adleman. Su seguridad radica en el problema 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 del producto es algo demasiado costoso para los ordenadores clásicos. Además el tamaño de los dos números primos se hace mayor a medida que aumente 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 [2] se podrá factorizar un número entero en tiempo polinomial. Todo esto hace necesario que se desarrollen nuevos sistemas criptográficos capaces de hacer frente a la computación cuántica, lo que fomenta el interés en el estudio de este ámbito.

En este contexto, este Trabajo de Fin de Grado muestra el desarrollo de la implementación del sistema de McEliece, posible alternativa al ser resistente tanto a la computación clásica como a la cuántica. A su vez se estudia su versión mejorada, el sistema de Niederreiter, así como algunos ataques.

1.2. Estado del arte

En la actualidad, el Instituto Nacional de Estándares y Tecnología [3] (NIST, National Institute of Standards and Technology) se encuentra llevando a cabo un concurso para seleccionar un estándar de criptografía post-cuántica con el fin de resolver la problemática planteada anteriormente. Dicho concurso fue anunciado en el PQCrypto 2016 [4], y se presentaron 23 esquemas de firma y 59 esquemas de cifrado con distintas aproximaciones, cada uno con sus ventajas y desventajas [5].

El 22 de julio de 2020 [6] tuvo lugar la tercera ronda de las cuatro planificadas. En cada una de estas rondas se descartan o se estudian con detalle los algoritmos propuestos. En esta ocasión se anunciaron siete finalistas entre los cuales se encuentra el criptosistema de McEliece que puede resistir ataques cuánticos a pesar de sus desventajas de tamaño de clave. Por otro lado, tenemos el criptosistema de Niederreiter que es una variante del sistema de McEliece con la particularidad de que también permite firmar mensajes y cifrar hasta diez veces más rápido. Se espera publicar en 2024 los documentos de estandarización resultantes del concurso del NIST.

1.3. Objetivos

El objetivo fundamental de este trabajo es la comprensión del sistema de McEliece, incluyendo sus ventajas, desventajas así como implementar tanto el cifrado como el descifrado. Se ha utilizado el lenguaje de programación SageMath [7] para su implementación. Por tanto, además de iniciarse en el uso de dicho lenguaje, ha sido necesaria la comprensión de muchos conceptos de la teoría de códigos correctores de errores.

1.4. SageMath

Esta herramienta es un sistema algebraico computacional que se ha utilizado en el desarrollo de este proyecto ya que destaca por estar construido sobre paquetes matemáticos como Numpy [8], Sympy [9], PARI/GP [10] o Maxima [11] y por acceder a sus potencias combinadas a través de un lenguaje común basado en Python [12]. Esto facilita la implementación de los distintos algoritmos. Para ser exactos se hizo uso de la versión online llamada cocalc [13] que nos permite ejecutar los programas de Sage en la nube. Con esta herramienta también se puede trabajar de forma colaborativa. Existen otras alternativas para este tipo de cálculos como son Magma [14], Maple [15], Mathematica [16], y MATLAB [17], pero el objetivo de Sage (escrito en Python y Cython [18]) es integrar todo el software de código abierto sobre matemática ya existente en una interfaz común, haciendo necesario únicamente la comprensión del lenguaje de programación Python.

Por otro lado, es conveniente destacar que dentro de la implementación del sistema McEliece ha sido necesario implementar los códigos de Goppa por no estar entre los códigos que sí se encuentran implementados en la herramienta. Entre los códigos que ya están implementados total o parcialmente se encuentran los códigos Reed-Solomon o Reed-Solomon generalizados [19] entre otros. Dichos códigos no son empleados para la construcción de este criptosistema por ser vulnerables, ya que se han podido criptoanalizar sin necesidad de emplear ordenadores cuánticos.

Como se comentaba, Sage es un software dotado de gran cantidad de paquetes matemáticos permitiendo abordar distintos aspectos matemáticos como son el álgebra o cálculo numérico teniendo en cuenta el cuerpo sobre el que se está trabajando, pudiendo emplear cuerpos finitos, los cuales son de gran utilidad en el desarrollo de las distintas partes del trabajo.

1.5. Fases del desarrollo

Las fases del desarrollo del presente Trabajo Fin de Grado se pueden clasificar en seis.

Una primera etapa fue dedicada al estudio y análisis del estado en el que se encuentra la

computación cuántica en la actualidad, así como sus efectos en cuestiones de seguridad informática con las posibles alternativas.

En lo que respecta a la segunda etapa, se procedió a investigar algunas de las propuestas presentadas al concurso del NIST como el criptosistema de McEliece, cobrando gran importancia al ser uno de los finalistas del concurso.

Procediendo a la tercera etapa, se llevó a cabo un análisis de las posibles herramientas alternativas para desempeñar el desarrollo del proyecto. Finalmente se decidió desempeñar dicho trabajo en el lenguaje de programación Sage, lenguaje basado en Python, por los motivos que se describieron en el apartado anterior.

En la cuarta etapa, ya teniendo el entendimiento de la herramienta escogida y de los criptosistemas se procedió a la realización de la implementación, adquiriendo un mejor entendimiento del mismo.

En la quinta etapa, una vez realizadas las implementaciones se estudiaron algunos de los posibles criptoanálisis que se podrían realizar, así como los distintos códigos criptográficos con los que se podría realizar.

Finalmente, alcanzando la sexta y última etapa, se realizó la implementación del ataque seleccionado, resolviendo los inconvenientes encontrados por el camino.

1.6. Estructura de la memoria

La estructura del presente documento se compone de 6 capítulos. En el primero, de introducción, se mencionan las motivaciones, objetivos, fases de desarrollo del trabajo, estructura de la memoria y el estado del arte, definiendo la problemática a resolver, así como el estado actual de dicha tecnología.

El segundo capítulo abarca la introducción al sistema criptográfico de McEliece, desde la explicación de conceptos claves para su entendimiento hasta la implementación llevada a cabo. A su vez se comenta la versión mejorada del criptosistema, el sistema de Niederreiter.

El tercer capítulo incluye menciones a distintos criptoanálisis, así como a algunos de los distintos códigos que se pueden emplear. Además se hace especial mención al ataque estándar, ataque que resulta eficaz si se empleasen códigos Reed-Solomon generalizados. Se incluye también la implementación correspondiente.

En el cuarto capítulo se incluye un presupuesto estimado de lo que conlleva el desarrollo e implementación de este Trabajo Fin de Grado en un contexto empresarial.

Finalmente, en el quinto y sexto capítulo de esta memoria se mencionan algunas de las conclusiones y posibles líneas futuras tanto en español como en inglés.

Capítulo 2. Análisis e implementación del criptosistema de McEliece

2.1. Introducción

El criptosistema de McEliece es un algoritmo de cifrado asimétrico desarrollado en 1978 por Robert McEliece [20] . Fue el primer esquema de este tipo en utilizar la aleatorización en el proceso de cifrado. Dicho criptosistema tuvo poca aceptación debido principalmente a los tamaños de las claves que necesita. Sin embargo, actualmente ha ganado importancia al ser resistente a ataques con ordenadores cuánticos y, como se ha mencionado con anterioridad, uno de los finalistas del concurso del NIST.

Antes de explicar el funcionamiento del sistema, es conveniente introducir algunos conceptos para facilitar su entendimiento [21].

2.1.1. Teoría de códigos

La teoría de códigos es una especialidad matemática que trata de las leyes de la codificación de la información. A grandes rasgos, codificar es transformar una información en una señal convenida para su comunicación. Decodificar sería el proceso inverso y complementario del anterior por el cual la señal comunicada es transformada en la información original [22].

Un cuerpo finito o campo de Galois es un cuerpo definido sobre un conjunto finito de elementos. Si $q = p^m$ es una potencia de un primo, al cuerpo finito con q elementos se le denota $GF(p^m)$ o $F_{(p^m)}$, y al correspondiente grupo multiplicativo se le denota F_q^k.

Un código C para un alfabeto fuente S y un alfabeto código A es una aplicación inyectiva, $C: S \rightarrow A^n$. La imagen de la aplicación C se denomina conjunto de palabras de código.

Los códigos de corrección de errores permiten, además de detectar un error, corregirlo sin necesidad de repetir la transmisión. De forma general, se puede decir que la técnica más empleada para la corrección consiste en identificar como combinación correcta, a la palabra de código que sea más parecida a la errónea recibida.

La distancia de Hamming es el número de bits en que difieren dos palabras de código. Cuanto mayor sea esta diferencia, menor es la posibilidad de que una palabra de código válida se transforme en otra palabra de código válida tras una serie de errores.

Si dos palabras de código tienen distancia de Hamming d, se necesitan d errores para convertir una en la otra.

Por otro lado, un código lineal es un código de corrección de errores para los que cualquier combinación lineal de palabras de código es también una palabra de código.

2.1.2. Código de Goppa binarios

Los códigos de Goppa binarios [23] son unos códigos correctores de errores lineales construidos sobre el cuerpo finito $GF(2) = \{0, 1\}$.

Estos fueron elegidos por McEliece debido a algunas de sus propiedades, ya que son difíciles de distinguir de otros códigos lineales aleatorios, se conocen algoritmos tanto de codificación como de descodificación para dichos códigos y son relativamente sencillos de construir. En 1978 introdujo su criptosistema con estos códigos.

Un código de Goppa se encuentra definido a partir de los elementos de $GF(p^m)$, cuerpo de Galois extensión finita de $GF(p)$. Para construir un código de Goppa, hacen falta un polinomio generador de grado t, $g(x)$, con coeficientes en $GF(p^m)$ llamado polinomio de Goppa y un subconjunto L de elementos de $GF(p^m)$ sin raíces de $g(x)$.

Un código de Goppa binario es un código de Goppa $\Gamma(L, g(x))$ sobre el cuerpo finito $GF(2^m)$ definido a partir de un polinomio $g(x)$ de grado t con coeficientes en $GF(2^m)$ y un subconjunto L de elementos de $GF(2^m)$ sin raíces de $g(x)$.

Se define el código de Goppa respecto de L y $g(x)$ como:

$\Gamma(L, g(x)) = \{c = (c_1, \ldots, c_n) \in GF(q^n) \,/\, \sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} \equiv 0 \; mod \; g(x)$ siendo $L = \{\alpha_1, \ldots, \alpha_n\}\}$

es decir, el código de Goppa denotado por $\Gamma(L, g(x))$ es un código corrector que está formado por todos los vectores o palabras $c = (c_1, \ldots, c_n)$ tales que $\sum_{i=1}^{n} \frac{c_i}{x-\alpha_i} \equiv 0 \; mod \; g(x)$.

Se trata por tanto de un caso particular de código de Goppa sobre el cuerpo finito$GF(2^m)$, es decir, con $p = 2$.

Un ejemplo sería el siguiente:
Dado el cuerpo finito $GF(2^4)$ construido a partir del polinomio binario irreducible $x^4 + x + 1$, se escoge el polinomio de grado 2 $g(x) = (x + \alpha^{13}) \cdot (x + \alpha^{14}) = x^2 + \alpha^2 x + \alpha^{12}$, obteniendo como resultado los elementos del conjunto $L = \{\alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7, \alpha^8, \alpha^9, \alpha^{10}, \alpha^{11}, \alpha^{12}\}$.

Un código de Goppa binario irreducible es aquel cuyo $g(x)$ es un polinomio irreducible. Un código de este estilo permite generar algoritmos de decodificación eficientes.

Un código de Goppa binario irreducible, $\Gamma(L,\ g(x))$, tiene distancia mínima de Hamming, d , mayor o igual a $2t\ +\ 1$, es decir, $d\ \geq\ 2t\ +\ 1$.

Dado que cualquier código con distancia mínima de Hamming d puede detectar como mucho $d\ -\ 1$ errores y corregir como mucho $\frac{d-1}{2}$ errores, un código de Goppa binario irreducible puede corregir un máximo de $\frac{(2t+1)-1}{2}$ errores.

Conocer el polinomio generador permite una corrección de errores eficiente. Aún no se han encontrado algoritmos de corrección de errores eficientes sin el conocimiento de ese polinomio generador.

2.1.3. Matriz de control de paridad

Una matriz de control o de chequeo de paridad de un código es una matriz asociada a un sistema de ecuaciones lineales y homogéneas, es decir, ecuaciones lineales con constantes nulas, cuyas soluciones son las palabras del código. Los elementos de esta matriz son los elementos p_{ij} tales que para cada palabra de código $(c_1, c_2,..., c_n)$ se tiene que $\sum_{i=1}^{n} c_i p_{ij} = 0,\ para\ 1\ \leq j \leq t.$

Como una palabra es del código solo si cumple esta igualdad, la matriz de paridad H satisface $cH^T = 0$. Así pues:

$$H = \begin{pmatrix} p_{11} & \cdots & p_{n1} \\ \vdots & \ddots & \cdots \\ p_{1t} & \cdots & p_{nt} \end{pmatrix}$$

2.1.4. Matriz generatriz

Llamamos matriz generatriz o generadora de un código lineal C a la matriz de una aplicación lineal biyectiva, $C: F_q^k \rightarrow C \subset F_q^n$, cuyas filas son una base del código C. Una base es un conjunto B del espacio vectorial V que cumple que todos los elementos de B pertenecen al espacio vectorial V, los elementos de B forman un sistema linealmente independiente y todo elemento de V se puede escribir como combinación lineal de los elementos de la base B.

Dada una base $\{g_1, \ldots, g_n\}$ de un código C, una matriz generatriz será:

$$G = \begin{vmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n-1} & g_{1,n} \\ g_{2,1} & g_{1,2} & \cdots & g_{1,n-1} & g_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ g_{k-1,1} & g_{k-1,2} & \cdots & g_{k-1,n-1} & g_{k-1,n} \\ g_{k,1} & g_{k,2} & \cdots & g_{k,n-1} & g_{k,n} \end{vmatrix}$$

Cualquier palabra del código puede obtenerse por tanto multiplicando por la matriz G el mensaje a codificar. Conociendo una matriz generatriz y un bloque del mensaje entrante es inmediato calcular la palabra código asociada a ese bloque. Basta pues almacenar la matriz generatriz en lugar de todas las palabras del código junto con su bloque relacionado, evitando así almacenar tanta información. Esto es especialmente importante cuando se trabaja con códigos cuyos valores de n y k sean realmente altos.

Una vez se ha calculado la matriz H, se puede obtener G a partir de la igualdad $GH^T = 0$. Esto se consigue resolviendo un sistema de ecuaciones. Se usa la notación $C[n, k]$ para denotar a un código lineal de longitud n y dimensión k con matriz generatriz G.

2.1.5. Matriz S

Matriz aleatoria invertible, es decir, que su determinante no valga cero y de orden $k \times k$.

2.1.6. Matriz P

Matriz cuadrada de tamaño n permutada con solo un uno por fila y columna.

2.1.7. Matriz G'

Es una matriz generadora de un código lineal tal que no existe un algoritmo rápido que corrija los errores con este código. Este código tiene la misma información y mínima distancia que el código generado por G. Se calcula multiplicando tres matrices, $G' = SGP$. Este parámetro será público junto con t.

Dada H matriz de control de paridad de un código lineal $C[n, k]$, se llama síndrome de un vector y al vector yH^T.

2.1.8. Algoritmo de Patterson

A la hora de corregir los errores de una palabra recibida tenemos que aplicar un algoritmo de descodificación rápida como es el algoritmo de Patterson [24] [25]. Dados $r \leq t$ errores para un polinomio irreducible $g(x)$ sobre $GF(2^m)$, consiste primero en calcular el síndrome del vector y a partir del síndrome, calculamos el polinomio localizador de errores

$L(x)$. Los pasos a seguir en el caso de un código de Goppa son los siguientes:

1. Sea $y = y_1, \dots, y_n$ una palabra codificada dada. Calculamos la función

2. Calculamos el polinomio $s(x) = \sum_{i=1}^{n} \frac{y_i}{x - \alpha_i} \ (mod\ g(x))$ y obtenemos el de localización de errores en cuatro pasos.

   a) Encontrar $h(x)$ tal que $s(x)h(x) \equiv 1\ (mod\ g(x))$. Si $h(x) = x$ entonces hemos terminado y la solución es $L(x) = x$.

   b) Calcular $d(x)$ tal que $d^2(x) \equiv h(x) + x\ (mod\ g(x))$.

   c) Encontrar $a(x)$ y $b(x)$ de grado menor tales que $d(x)b(x) \equiv a(x)(mod\ g(x))$.

   d) Definir el polinomio de localización de errores como $L(x) = a^2(x) + b^2(x)x$.

3. Una vez que tenemos el polinomio de localización de errores definido, lo usamos para determinar el conjunto de posiciones de error $E = \{i \mid e_i \neq 0\}$.

4. Definimos el vector de errores $e = (e_1, \dots, e_n)$ como $e_i = 1$ si $i \in E$ y $e_i = 0$ en caso contrario.

5. Definimos la palabra codificada $c = y - e$.

2.2. Definición del sistema

El sistema de McEliece es un criptosistema de clave pública que se basa en códigos lineales para introducir errores. Este algoritmo consta de los siguientes elementos:

- Código binario (C) que permite corregir t errores
- Matriz generadora de C (G) cuyas filas constituyen una base del código.
- Una matriz aleatoria invertible (S)
- Una matriz de permutación aleatoria (P)
- Una matriz G' que se calcula a partir de S, G, y P. ($G' = S \cdot G \cdot P$)
- Un vector de errores (e) que se insertan en el cifrado
- Un algoritmo de decodificación D que permita eliminar los errores insertados.

En cuanto a la clave pública está compuesta por G' y t. Por otra parte, la clave privada se encuentra compuesta por S,G, P y D.

A la hora de cifrar el mensaje se hace uso de la matriz G' y el vector de errores generando el cifrado C del mensaje m de la siguiente manera:

$$C = mG' \oplus e$$

Para recuperar el mensaje m se elimina P multiplicando C por la inversa de esta matriz quedando que $CP^{-1} = (mS)G \oplus eP^{-1}$. Tras aplicar un algoritmo de decodificación eficiente para el código C, se eliminan los errores devolviendo el vector mS del cual se obtiene el mensaje multiplicándolo por la inversa de S. Por tanto para descifrar el mensaje es

necesario calcular la inversa de P, aplicar un algoritmo de decodificación y calcular la inversa de S, además de realizar algunas multiplicaciones de matrices.

2.3. Implementación del sistema

2.3.1. Códigos de Goppa binarios

En el desarrollo de este proyecto se ha utilizado la herramienta SageMath. Antes que nada, es necesario construir los códigos de Goppa binarios. Estos códigos han tenido que ser construidos desde cero completamente, ya que Sage no tiene actualmente implementada ninguna función acerca de ellos. Lo primero que necesitamos es definir el código de Goppa binario.

N indica la longitud deseada del código de Goppa que vamos a construir y m la extensión del cuerpo F_2 que queremos emplear en este código. Definimos todos los elementos de componen el código de Goppa, ya mencionados previamente, como el polinomio de Goppa (g) o el subconjunto L.

```
m = 4;
N = 2^m
K_.<a> = GF(2)
F.<a> = GF(2^m)

PR = PolynomialRing(F,'X')
X = PR.gen()
g = X^3+X+1
L = [a^i for i in range(N)]
```

Figura 2.1: Construcción del código de Goppa

2.3.4. Matriz H

La matriz H es la matriz de control de paridad, para construirla se calcula con la multiplicación de las tres matrices T, V y D definidas en las siguientes subsecciones ($H = TVD$).

```
H = T*V*D
print ("Matriz H: ")
show(H)
```

Matriz H:

$$\begin{pmatrix} 1 & a^2+1 & a & a^3+a^2+a & a^2 & a^2+a+1 & a^3+a+1 & a^2+a+1 & a+1 & a^3+a^2+1 & a^2+a & a^2+a & a^3+1 & a^2+a+1 & a^2+a & 1 \\ 1 & a^3+a & a^3 & a^3+1 & a^3+a^2 & 1 & a^3+a^2+1 & a^2 & a^3+a^2+a+1 & a^3+a+1 & 1 & a & a^3+a^2+a & a^2+1 & a+1 & 1 \\ 0 & a & a^2 & a^3+a & a+1 & 1 & a^3 & a^3+a^2+1 & a^2+1 & a^3+a^2+a+1 & 1 & a^3+1 & a^3+a^2 & a^3+a+1 & a^3+a^2+a & 0 \end{pmatrix}$$

Figura 2.2: Implementación de la matriz H

2.3.4.1. Matriz T

Para construir esta matriz se recorren los elementos insertando ceros por encima de la diagonal, unos en la diagonal y los coeficientes correspondientes del polinomio de Goppa por debajo de la diagonal. Para esto último se hace uso de la función list().

```
T = matrix(F,rango,rango)
for i in range(rango):
    count = rango - i
    for j in range(rango):
        if i > j:
            T[i,j]=g.list()[count]
            count = count + 1
        if i < j:
            T[i,j] = 0
        if i == j:
            T[i,j] = 1
```

Figura 2.3: Implementación de la matriz T

Como se muestra a continuación, el polinomio de Goppa es el definido previamente y la función g.list() nos devuelve sus coeficientes.

```
print ("Matriz T: ")
show(T)
show(F)
show(g.list())
show(g)
```

Matriz T:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$\mathbb{F}_{2^4}$$

$$[1, 1, 0, 1]$$

$$X^3 + X + 1$$

Figura 2.4: Visualización de la matriz T

2.3.4.2. Matriz V

Esta matriz contiene elementos del cuerpo sobre el que estamos trabajando en concreto cada fila tiene una copia de dicho cuerpo pero con los elementos elevados al número de la fila en la que se encuentra.

```
V = matrix([[L[j]^i for j in range(N)] for i in range(rango)])
print ("Matriz V: ")
show(V)
```

Matriz V:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & a & a^2 & a^3 & a+1 & a^2+a & a^3+a^2 & a^3+a+1 & a^2+1 & a^3+a & a^2+a+1 & a^3+a^2+a & a^3+a^2+a+1 & a^3+a^2+1 & a^3+1 \\ 1 & a^2 & a+1 & a^3+a^2 & a^2+1 & a^2+a+1 & a^3+a^2+a+1 & a^3+1 & a & a^3 & a^2+a & a^3+a+1 & a^3+a & a^3+a^2+a & a^3+a^2+1 \end{pmatrix}$$

Figura 2.5: Matriz V

En la primera fila el cuerpo estaría elevado a cero, en la segunda a uno con lo que se

muestra el cuerpo tal cual y en la tercera estaría elevado cada elemento al cuadrado.

2.3.4.3. Matriz D

Es una matriz diagonal que contiene la inversa de cada uno de los elementos del cuerpo con el que estamos trabajando.

```
D = diagonal_matrix([1/g(L[i]) for i in range(N)])
print ("Matriz D: ")
show(D)
```

```
Matriz D:
```

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & a^2+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & a^3+a^2+a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & a^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & a^2+a+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & a^3+a+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & a^2+a+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^3+a^2+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^2+a & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^2+a & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^3+1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^2+a+1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a^2+a & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Figura 2.6: Matriz D

2.3.5. Matriz G’

En este caso, se realiza la multiplicación correspondiente, $G' = SGP$.

```
Matriz G'

In [28]: G_prima = S*G*P

print ("Matriz G_prima: ")
show(G_prima)

Matriz G_prima:
```

$$\begin{pmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Figura 2.7: Matriz G’

2.3.6. Matriz S

Recordamos que S es una matriz aleatoria invertible de orden $k \times k$. Para generarla se calculan matrices aleatorias mientras el determinante sea cero hasta obtener una matriz adecuada. Esta matriz se multiplica por G y P para calcular G'.

```
S = random_matrix(GF(2), N-m*rango)
while (S.determinant()==0):
    S = random_matrix(GF(2), N-m*rango)

print ("Matriz S: ")
show(S)

Matriz S:
```

$$\begin{pmatrix}
1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1
\end{pmatrix}$$

Figura 2.8: Matriz S

2.3.7. Matriz P

Matriz de permutaciones con solo un uno por fila y columna que se utiliza para calcular G' multiplicándose con S y G.

```
rng = range(N)
P = matrix(GF(2),N);

for i in range(N):
    p = floor(len(rng)*random());
    P[i,rng[p]]=1;
    rng=[*rng[:p], *rng[p+1:]];

#print ("Matriz P: ")
show(P)
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figura 2.9: Matriz P

2.3.8. Algoritmo de Patterson

El algoritmo de Patterson se emplea para descodificar un mensaje como se ha comentado con anterioridad. Para su desarrollo se emplean funciones adicionales para descomponer cómodamente los polinomios o emplear el algoritmo de Euclides Extendido.

2.3.8.1. Función descomponer polinomios

La función *descomponer_polinomio(p)* recibe por parámetros un polinomio, y devuelve su descomposición. Se utiliza para definir el polinomio de localización de errores dentro del algoritmo de Patterson.

Descompone o separa un polinomio dado en dos partes que nos permitirán definir el polinomio de localización de errores como $a^2(x) + b^2(x)x$.

```
def descomponer_polinomio(p):

    Phi1 = p.parent()

    p0 = Phi1([sqrt(c) for c in p.list()[0::2]])
    p1 = Phi1([sqrt(c) for c in p.list()[1::2]])
    return (p0,p1)
```

Figura 2.10: Función para descomponer polinomios

2.3.8.2. Algoritmo de Euclides Extendido

El algoritmo de Euclides Extendido calcula el máximo común divisor de dos números enteros a y b. En este caso se emplea para expresarlo como la mínima combinación lineal de esos números, es decir, encontrar números enteros s y t tales que $mcd(a,b) = as + bt$.

```
def algoritmo_euclides_extendido(self, other):
    delta = self.degree()

    if other.is_zero():
        ring = self.parent()
        return self, R.one(), R.zero()

    ring = self.parent()
    a = self
    b = other

    s = ring.one()
    t = ring.zero()

    resto0 = a
    resto1 = b

    while true:
        cociente,resto_auxiliar = resto0.quo_rem(resto1)
        resto0 = resto1
        resto1 = resto_auxiliar

        s = t
        t = s - t*cociente

        if resto1.degree() <= floor((delta-1)/2) and resto0.degree() <= floor((delta)/2):
            break

    V = (resto0-a*s)//b
    coeficiente_lider = resto0.leading_coefficient()

    return resto0/coeficiente_lider, s/coeficiente_lider, V/coeficiente_lider
```

Figura 2.11: Algoritmo de Euclides Extendido

2.3.8.3. Inversa de un polinomio

La función *inversa_(p,g)* utiliza la versión del algoritmo de Euclides extendido implementada en Sage. Como parámetros recibe un polinomio y su divisor, devolviendo el polinomio de grado menor.

Se necesita para calcular $d(x)$ tal que $d^2(x) \equiv h(x) + x \,(mod\, g(x))$, en el segundo paso del algoritmo de Patterson.

```
def inversa_g(p,g):
    (d,u,v) = xgcd(p,g)
    return u.mod(g)
```

Figura 2.12: Inversa de un polinomio

En cuanto a la implementación del algoritmo de Patterson, este recibe el código a

decodificar haciendo uso de las funciones anteriores. Además se muestran las posiciones donde se han encontrado dichos errores antes de ser corregidos.

```
def decodePatterson(y):

    alpha = vector(H*y)

    polinomioS = PR(0)
    for i in range(len(alpha)):
        polinomioS = polinomioS + alpha[i]*(X^(len(alpha)-i-1))

    vector_g = descomponer_polinomio(g)
    w = ((vector_g[0])*inversa_g(vector_g[1],g)).mod(g)
    vector_t = descomponer_polinomio(inversa_g(polinomioS,g) + X)

    R = (vector_t[0]+(w)*(vector_t[1])).mod(g)

    (a11,b11,c11) = algoritmo_euclides_extendido(g,R)


    sigma = a11^2+X*(c11^2)

    for i in range(N):
        if (sigma(a^i)==0):
            print ("Error encontrado en la posición: " + str(i))
            y[i] = y[i] + 1
    return y
```

Figura 2.13: Algoritmo de Patterson

2.3.9. Cifrado

A la hora de cifrar se genera un mensaje aleatorio con la ayuda de la función randint.

```
u = vector(K_,[randint(0,1) for _ in range(G_prima.nrows())])
print("Vector u");
show(u)
```

```
Vector u
```

$$(1, 1, 0, 0)$$

Figura 2.14: Mensaje aleatorio

Ese mensaje se cifra multiplicando por la matriz G', ya calculada previamente, y se crea un vector de errores e.

```
c = u*G_prima
print("Vector c");
show(c)
```

```
Vector c
```

$$(1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0)$$

Figura 2.15: Generando el cifrado

```
e = vector(K_,N)
e[8] = 1
e[9] = 1
print("Vector de errores e");
show(e)
```

```
Vector de errores e
```

$$(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)$$

Figura 2.16: Generando el vector de errores

Tras insertar el vector de errores se obtiene el mensaje final, que es el que enviaremos.

```
y = c + e
print("Vector codificado y");
show(y)
```

```
Vector codificado y
```

$$(1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0)$$

Figura 2.17: Insertando el vector de errores

2.3.10. Descifrado

```
yP = y*(P.inverse())
yd = decodePatterson(yP)
corregido = (G.transpose()\yd)*S.inverse()
show(corregido)
```

```
Error encontrado en la posición: 7
Error encontrado en la posición: 13
```

$$(1, 1, 0, 0)$$

Figura 2.18: Descifrando el mensaje

Para descifrar el mensaje es necesario calcular la inversa de P, aplicar un algoritmo de decodificación y calcular la inversa de S. Al sustituir G' $(G' = SGP)$ en nuestro cifrado $(C = mG' \oplus e)$ queda que $C = mSGP \oplus e$. Se elimina P aplicando la inversa de esta matriz quedando que $cP^{-1} = (mS)G \oplus eP^{-1}$. Tras aplicar el algoritmo de decodificación de Patterson este elimina los errores devolviendo el vector mS del cual se obtiene el mensaje aplicando la inversa de S.

2.4. Ejemplo criptosistema de McEliece

Para ejemplificar este algoritmo imaginemos dos amigos, Alice y Bob, que quieren enviarse un mensaje secreto.

El receptor construye un código de Goppa binario fácilmente resoluble con matriz generadora de orden $k \times n$, denotada como G, y capacidad de corrección de t errores como el siguiente:

$$\mathbb{F}_2$$

$$\mathbb{F}_{2^4}$$

$$\mathbb{F}_{2^4}[X]$$

$$X^3 + X + 1$$

$$[1, a, a^2, a^3, a+1, a^2+a, a^3+a^2, a^3+a+1, a^2+1, a^3+a, a^2+a+1, a^3+a^2+a, a^3+a^2+a+1, a^3+a^2+1, a^3+1, 1]$$

Figura 2.19: Código de Goppa binario

Matriz H:

$$\begin{pmatrix} 1 & a^2+1 & a & a^3+a^2+a & a^2 & a^2+a+1 & a^3+a+1 & a^2+a+1 & a+1 & a^3+a^2+1 & a^2+a & a^2+a & a^3+1 & a^2+a+1 & a^2+a & 1 \\ 1 & a^3+a & a^3 & a^3+1 & a^3+a^2 & 1 & a^3+a^2+1 & a^2 & a^3+a^2+a+1 & a^3+a+1 & 1 & a & a^3+a^2+a & a^2+1 & a+1 & 1 \\ 0 & a & a^2 & a^3+a & a+1 & 1 & a^3 & a^3+a^2+1 & a^2+1 & a^3+a^2+a+1 & 1 & a^3+1 & a^3+a^2 & a^3+a+1 & a^3+a^2+a & 0 \end{pmatrix}$$

Figura 2.20: Matriz de control de paridad (imágenes de $G(\alpha^n)$)

$$\begin{pmatrix} 0&0&0&1&0&0&1&0&0&1&0&0&1&0&0&0 \\ 0&1&0&1&1&1&0&1&0&1&1&1&0&1&1&0 \\ 0&0&1&1&0&1&1&1&1&0&1&1&0&1&1&0 \\ 1&1&0&0&0&1&1&1&1&1&0&0&1&1&0&1 \\ 0&1&1&1&1&0&1&0&1&1&0&0&1&0&0&0 \\ 0&0&0&0&1&0&1&1&1&0&0&0&1&1&0&0 \\ 0&1&0&0&0&0&0&0&1&1&0&1&1&0&1&0 \\ 1&0&0&1&0&1&1&0&1&1&1&0&0&1&1&1 \\ 0&0&0&1&0&0&1&1&0&1&0&1&1&1&1&0 \\ 0&0&1&0&0&0&0&1&1&1&0&0&1&0&1&0 \\ 0&1&0&1&1&0&0&0&0&1&0&0&0&1&1&0 \\ 0&0&0&0&1&1&0&1&1&1&1&1&0&1&0&0 \end{pmatrix}$$

Figura 2.21: Matriz de control de paridad (expresada en forma binario)

Matriz G:

$$\begin{pmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&1&0&0&1&1&1&0&0&1&0&1&0&0&1&0 \\ 0&0&1&0&0&0&1&0&1&1&1&1&0&0&1&1 \\ 0&0&0&1&0&0&1&1&0&1&0&1&1&1&1&1 \end{pmatrix}$$

Figura 2.22: Matriz generatriz

Esta matriz generadora G se calcula fácilmente a partir de la matriz de chequeo de paridad.

La matriz G se trasforma en G' mediante la expresión $G' = SGP$, donde S es una matriz aleatoria invertible de orden $k \times k$, y P es una matriz de permutaciones de orden $n \times n$. Se escogen matrices como estas:

Matriz P

$$\begin{pmatrix} 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \end{pmatrix}$$

Figura 2.23: Matriz P

Matriz S

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Figura 2.24: Matriz S

Matriz G'

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Figura 2.25: Matriz G'

La matriz G' es una matriz generadora para un código lineal tal que no existe un algoritmo rápido que corrija los errores con este código.

G' se publica como clave de cifrado. El emisor cifra un vector de mensaje m de k bits convirtiéndolo en un vector de mensaje cifrado C de n bits mediante $C = mG' \oplus e$, donde e en un vector de errores de n bits de peso t elegido por el remitente.

$$(0, 1, 1, 1)$$

Figura 2.26: Mensaje

```
c = u*G_prima
print("Vector c"); show(c)
```

```
Vector c
```

$$(0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0)$$

Figura 2.27: Cifrando el mensaje

```
Vector de errores e
```

$$(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0)$$

Figura 2.28: Vector de errores

Se envía Y = mG' + e

```
y = c + e
print("Vector codificado y"); show(y)
```

```
Vector codificado y
```

$$(0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0)$$

Figura 2.29: Insertando errores

El receptor, sabiendo que $C = mG' \oplus e = mSGP \oplus e$, calcula $CP^{-1} = mSG \oplus eP^{-1}$ y usa un algoritmo de decodificación para el código original que elimina el vector de errores eP^{-1} y recupera así el vector mS. El mensaje del remitente se encuentra fácilmente mediante

$m = (mS)S^{-1}$.

```
yP = y*(P.inverse())                                                    0.018 seconds 60
show(yP)
yd = decodePatterson(yP)
show(yd)
corregido = (G.transpose()\yd)*S.inverse()
show(corregido)
```

$$(1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0)$$

```
Error encontrado en la posición: 7
Error encontrado en la posición: 14
```

$$(1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0)$$

$$(0, 1, 1, 1)$$

Figura 2.30: Descifrando el mensaje

2.5. Criptosistema de Niederreiter

El criptosistema de Niederreiter es una variación del criptosistema de McEliece desarrollado en 1986 por Harald Niederreiter [26]. Aplica la misma idea a la matriz de chequeo de paridad, H, de un código lineal. Es equivalente a McEliece desde el punto de vista de la seguridad, utiliza un síndrome como texto cifrado y el mensaje es un patrón de error, pero el cifrado de Niederreiter es unas diez veces más rápido que el cifrado de McEliece. Niederreiter se puede utilizar para construir un esquema de firma digital.

Este sistema consta de los siguientes elementos:

- Código de Goppa binario que permite corregir t errores
- Matriz de control (H)
- Una matriz aleatoria invertible (S)
- Una matriz de permutación aleatoria (P)
- Una matriz H' que se calcula a partir de S, H, y P ($H' = SHP$).
- Un algoritmo de decodificación que nos permita eliminar los errores insertados.

En este caso la matriz generatriz coincide con la matriz de chequeo de paridad. En cuanto a la clave pública está compuesta por H' y t, siendo t el número de errores que se pueden corregir. Mientras que la clave privada se encuentra compuesta por S, H y P.

A la hora de cifrar el mensaje se hace uso de la matriz H' multiplicándola por el vector del mensaje, es decir, $c = H'm^T$.

Para recuperar el mensaje se sustituye H' ($H' = SHP$) en nuestro cifrado ($c = H'm$) quedaría que $c = SHPm^T$. Se elimina S aplicando la inversa de esta matriz quedando que $S^{-1}c = HPm^T = H(mP^T)^T$. Tras aplicar un algoritmo de decodificación este elimina los errores devolviendo el vector mP^T del cual se obtiene el mensaje aplicando la traspuesta de la inversa de P, $(P^{-1})^T$. Por tanto para descifrar el mensaje es necesario calcular las inversas y traspuestas de S y P, así como aplicar un algoritmo de decodificación.

Capítulo 3. Introducción al criptoanálisis del sistema de McEliece

Entre la multitud de ataques que existen en criptografía destacan los siguientes:

· Ataque por fuerza bruta: es un tipo de ataque en el cual un enemigo o atacante intenta todas las posibles claves hasta dar con la correcta, la que descifraría el mensaje. Estos intentos son aleatorios y por tanto, conllevan una gran cantidad de computación para poder realizarse, pero ningún esfuerzo. En el caso de darse este ataque, cuanto mayor sea la longitud de nuestra clave, mayor será el tiempo que necesite este atacante hasta dar con ella, dado que tiene que ir probando con todas las claves posibles.

· Ataques contra la estructura del código o ataque estructural: consiste en un ataque más directo o un ataque a un mensaje en concreto, intenta decodificar un mensaje sin necesidad de resolver el criptosistema por completo. El objetivo de un atacante sería recomponer la estructura del mensaje (o al menos una parte) del mensaje original utilizando la clave pública que también es conocida por el atacante.

· Decodificación por conjuntos de información: es el ataque más efectivo contra el criptosistema McEliece. En Junio de 2008 se realizó el primer ataque con éxito de este tipo de ataque sobre el criptosistema de McEliece. En dicho ataque se utilizaron 200 ordenadores con un total de 300 núcleos. Los cálculos llevaron unos 90 días en completarse (finalizaron a principios de octubre). En este primer ataque se fueron afinando el valor de los parámetros y ya en los siguientes ataques se partió de los más óptimos. Estos parámetros permitían disminuir el coste de cálculo de una manera considerable.

Por otro lado comentar la importancia del tipo de código que se utiliza en este sistema.

Existen distintos tipos de código:

· Códigos lineales: son códigos correctores de errores para los cuales cualquier combinación lineal de palabras de código es también una palabra de código. Estos presentan algunas ventajas que facilitan el cómputo como por ejemplo que su distancia mínima se determina fácilmente analizando la matriz de chequeo de paridad.

· Códigos Reed-Solomon (RS) y Reed-Solomon generalizados (GRS): son un tipo especial de códigos cíclicos diseñados por Irving S. Reed y Gustave Solomon en 1960. Resultan especialmente útiles en la corrección de errores a ráfagas que afectan a bloques contiguos. Son muy utilizados en telecomunicaciones siendo algunos ejemplos de sus aplicaciones la tecnología DSL y variantes como la ADSL y VDSL, distintas unidades de almacenamiento como discos duros, CD, DVD, BlueRay o los códigos de barras y radiodifusión digital actual como DVB y sus variantes.

· Códigos Reed-Muller: son una familia infinita de códigos lineales, que toman su nombre de los dos matemáticos que los propusieron en el año 1954, prácticamente al mismo tiempo, en dos trabajos independientes: Irving Stoy Reed y David Eugene Muller. Ambos se ocuparon de introducir estos en el caso binario. Hoy se sabe que el primero en realizar la primera de las construcciones para estos códigos en su forma binaria fue Muller, mientras que su estudio en detalle y la sencilla decodificación por la que son tan conocidos e importantes en este caso binario es obra de Reed.

Los códigos de Reed-Muller tienen una gran importancia en la historia. Su estudio en la década de los años 50 fue fundamental para que en los años posteriores se hiciesen grandes avances en la exploración espacial. Así, desde 1969 hasta 1977, todas las naves espaciales de la NASA iban equipadas con un código de Reed-Muller binario de longitud 32, dimensión 6 y distancia mínima 16. Se escogió dicho código dado que el cociente entre la dimensión del mismo y su longitud es (relativamente) pequeño para la amplia distancia mínima que posee. Por esta razón podemos decir que nos encontramos ante un código de bajo coste y buenas capacidades para corregir errores.

Algunos de estos códigos se pueden encontrar implementados en los paquetes de Sage sin embargos estos no han sido empleados en la implementación ya que se ha descubierto que al utilizar códigos como los códigos Reed-Solomon, Reed-Solomon generalizados o Reed-Muller el sistema se vuelve vulnerable.

Entre los ataques que se pueden aplicar al utilizar estos códigos se encuentran el de Sidelnikov y Shestakov [27] o el ataque por filtración [28]. En el caso del primero, consiste en recuperar la estructura de cualquier código Reed-Solomon generalizado a partir de la obtención del par de vectores a y b que definen el código Reed-Solomon generalizado. Mientras que el ataque por filtración es una versión en la cual no es necesario calcular las palabras del código de peso mínimo. En la forma estándar de este ataque [29] se recuperan los vectores a y b de un código GRS_k a partir de las matrices generatrices del código de dimensión k y del código de dimensión $k-1$, es decir, este ataque se basa en que a partir de los códigos $GRS_k(a,b)$ y $GRS_{k-1}(a,b)$, se puede calcular el código $GRS_{k-2}(a,b)$, donde a y b son elementos de F_q^n.

Teniendo en cuenta que el producto estrella de dos elementos $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ de F_q^n, se define como el producto escalar de las componente de los dos vectores, es decir, $x \star y = (x_1y_1, \ldots, x_ny_n)$. Y la potencia i-ésima del vector x es el producto i veces, $x^{\star i} = x \star x \star \ldots \star x$.

Aplicando este operador a los códigos tenemos que, dados dos códigos, C_1 y C_2, de igual longitud el código resultante del producto estrella es el espacio vectorial obtenido a partir de todos los productos por componentes donde el primer vector es un elemento de C_1 y el segundo vector es un elemento de C_2, es decir, $C_1 \star C_2 = \{x \star y \mid x \in C_1,\ y \in C_2\}$.

Para calcular el vector b primero se calcula una base del código $GRS_{k-1}(a,b)^{\star 2}$. Una vez calculada , e resuelve el sistema construido a partir de las ecuaciones que tienen la forma $g_i \star h_j$, donde g_i hace referencia a la fila i-ésima de una matriz generatriz de $GRS_k(a,b)$ y h_j hace referencia a la fila j-ésima de una matriz de control de paridad, H, de $GRS_{k-1}(a,\ b)^{\star 2}$. Esto se debe a que se cumple que $(c \star g_i) \in GRS_{k-1}(a,\ b)^{\star 2}$ si y solo si $(c \star g_i) \cdot H^{\ T} = 0$, es decir, $(c \star g_i) \cdot h_j = 0$. Además se verifica que $(c \star g_i) \cdot h_j = c \cdot (g_i \star h_j)$.

Esto se repite con los dos códigos Reed-Solomon generalizados de dimensión más baja hasta obtener el código con dimensión uno. Como $GRS_1(a,\ b) = \{\lambda b \mid \lambda \in F_q\} = < b >$ debido a que sus elementos se obtienen evaluando en constantes y multiplicándolas por el elemento b, permitiendo obtenerlo. A este proceso se le conoce como filtración.

Para obtener el vector a se utiliza el código de dimensión dos calculado en el proceso anterior.

A partir de estos dos códigos, se calcula una matriz generatriz como la siguiente, siendo n el tamaño de los elementos codificados:

$$G = \begin{pmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \end{pmatrix}$$

y la matriz generatriz de $GRS_2(a,\ b)$:

$$G' = \begin{pmatrix} 1 & 0 & \frac{(a_2-a_3)b_3}{(a_2-a_1)b1} & \cdots & \frac{(a_2-a_n)b_n}{(a_2-a_1)b1} \\ 0 & 1 & \frac{(a_3-a_1)b_3}{(a_2-a_1)b_2} & \cdots & \frac{(a_n-a_1)b_n}{(a_2-a_1)b_2} \end{pmatrix}$$

Como la matriz generatriz en forma estándar es única, ambas matrices generatrices del mismo código deben ser equivalentes a la misma matriz generatriz en forma estándar. Por tanto, si cogemos nuestro vector b y la primera fila de la matriz G, al igualar se obtienen los valores del vector a que buscamos. Quedaría que:

$$\frac{(a_2-a_i)b_i}{(a_2-a_1)b_1} = g_{1,i}$$

donde i es el número de la fila correspondiente.

Despejando la variable a tenemos que:

$$a_i = \frac{a_2 b_i - (a_2-a_1)b_1 g_{1i}}{b_i}$$

Para obtener el vector se fijan los dos primeros valores, $a_1 = 1$ y $a_2 = 2$. Con esto se tendría determinado por completo el código Reed-Solomon generalizado y se concluiría con éxito el ataque.

3.1. Implementación del ataque estándar

A la hora de implementar el ataque se hace uso de varias funciones.

La primera función desarrollada es *codigos_traza(C, C1)* la cual va obteniendo los códigos Reed-Solomon generalizados con dimensiones menores a la del código C1. Primero se obtiene el código cuadrado del código C1. Una vez obtenido el código cuadrado de C1, también se valida que su dimensión cumpla con las propiedades correspondientes. Por último se resuelve el correspondiente sistema de ecuaciones, devolviendo los dos códigos menores obtenidos.

```
def codigos_traza(C,C1):

    baseC1cuadrado=[];
    n=C1.generator_matrix().row(0).length();

    for i in range(C1.generator_matrix().nrows()):
        for j in range(i,C1.generator_matrix().nrows()):
            baseC1cuadrado+=[vector(F,[C1.generator_matrix().row(i)[h]*C1.generator_matrix().row(j)[h] for h in range(n)])];
    V = VectorSpace(F,n)

    subV = V.subspace(baseC1cuadrado)
    Gcuadrada = subV.basis_matrix()
    C1cuadrado = codes.LinearCode(Gcuadrada);

    if C1cuadrado.dimension() != (2*C1.dimension()-1):
        raise TypeError('Deben emplearse códigos GRS para realizar el ataque')

    n=len(C1.random_element());
    lista=[]
    for i in range(C.generator_matrix().nrows()):
        for j in range(C1cuadrado.parity_check_matrix().nrows()):
            lista+=[list(vector(F,[F(C.generator_matrix().row(i)[h]*C1cuadrado.parity_check_matrix().row(j)[h]) for h in range(n)]))]

    AAA=matrix(F,lista)
    matrizG_codigo=AAA.right_kernel().basis_matrix()
    C2=codes.LinearCode(matrizG_codigo)

    return [C2, C1]
```

Figura 3.1: Función códigos

A continuación esta función es llamada por la función *reducir_codigos_traza(C, C1)*, repitiendo el proceso anterior con los códigos de menor dimensión que se obtengan hasta obtener un código de dimensión 1. Esta función devuelve dichos códigos.

```
def reducir_codigos_traza(C,C1):

    while(C1.dimension() > 1):
        [C1,C] = codigos_traza(C,C1);

    return [C1,C]
```

Figura 3.2: Función reducir códigos

Por último, la función *ataque_estandar(CCC, CCC1)* recibe por parámetros los códigos GRS de dimensiones k y $k - 1$, devolviendo los parámetros a y b para obtener el código de dimensión $k - 2$.

En primer lugar llama a la función *reducir_codigos* la cual devuelve los dos códigos de menor dimensión. Dado que $GRS_k(a,b) = GRS_k(a,\lambda b)$ para obtener ese vector λb que se corresponde con la matriz generatriz del código de dimensión 1. Se calcula la matriz generatriz del código de dimensión 2 y se realiza la resolución del sistema. Para ello, se fijan las dos primeras posiciones del vector a con los valores $a_1 = 1$ y $a_2 = 2$, permitiendo recuperar el vector a, como se expuso en la teoría.

```
def ataque_estandar(CCC,CCC1):
    [bb,G]=reducir_codigos_traza(CCC,CCC1)
    bb = bb.generator_matrix()
    G = G.systematic_generator_matrix()

    F = CCC.base_ring()

    g = vector(F,[G[0,i] for i in range(G.ncols())])
    b = vector(F,[bb[0,i] for i in range(bb.ncols())])
    a = vector(F,[0 for i in range(bb.ncols())])

    a[0] = 1
    a[1] = 2

    for i in list(range(2,len(g))):
        print("i = " + str(i))
        num = a[1]*b[i]-(a[1]-a[0])*g[i]
        den = b[i]

        a[i] = (num/den)

    return [a,b];
```

Figura 3.3: Función del ataque estándar

3.2. Ejemplo de ataque estándar

Imaginemos que de los códigos $GRS_4(a, b)$ y $GRS_3(a, b)$, con valores $a = (1, 2, 9, 3, 10, 8, 5, 4) \in F_{11}$ y $b = (3, 10, 5, 9, 9, 10, 5, 2) \in F_{11}$, únicamente se conocen una matriz generatriz de cada código.

```
F=GF(11);
a=[F(i) for i in [1,2,9,3,10,8,5,4]];#3,9.....
b=[F(i) for i in [3,10,5,9,9,10,5,2]];
k=5
C=codes.GeneralizedReedSolomonCode(a, k, b);
show(C)
show(C.generator_matrix())
```

$[8, 5, 4]$ Generalized Reed-Solomon Code over $\mathbb{F}_{11}$

$$\begin{pmatrix} 3 & 10 & 5 & 9 & 9 & 10 & 5 & 2 \\ 3 & 9 & 1 & 5 & 2 & 3 & 3 & 8 \\ 3 & 7 & 9 & 4 & 9 & 2 & 4 & 10 \\ 3 & 3 & 4 & 1 & 2 & 5 & 9 & 7 \\ 3 & 6 & 3 & 3 & 9 & 7 & 1 & 6 \end{pmatrix}$$

Figura 3.4: Código Reed-Solomon de dimensión k

```
C1=codes.GeneralizedReedSolomonCode(a, k-1, b);
show(C1)
show(C1.generator_matrix())
```

$[8, 4, 5]$ Generalized Reed-Solomon Code over $\mathbb{F}_{11}$

$$\begin{pmatrix} 3 & 10 & 5 & 9 & 9 & 10 & 5 & 2 \\ 3 & 9 & 1 & 5 & 2 & 3 & 3 & 8 \\ 3 & 7 & 9 & 4 & 9 & 2 & 4 & 10 \\ 3 & 3 & 4 & 1 & 2 & 5 & 9 & 7 \end{pmatrix}$$

Figura 3.5: Código Reed-Solomon de dimensión $k - 1$

Ahora se emplea la función *ataque_estandar* desarrollada para obtener los valores de los vectores a y b, u otros valores que también permiten reconstruir el código de partida:

```
parametros=ataque_estandar(C,C1)
parametros
```

```
[(1, 2, 9, 3, 10, 8, 5, 4), (1, 7, 9, 3, 3, 7, 9, 8)]
```

Figura 3.6: Salida del ataque estándar (vectores a y b)

En este ejemplo se recupera el mismo vector a, aunque no tiene porqué ocurrir. En cuanto al vector b en este caso es 4b. Entonces, aunque en este caso se ve claramente, al tener el mismo vector a que los valores obtenidos van a generar el mismo código del que se parte ya que se sabe que $GRS_k(a, b) = GRS_k(a, \lambda b)$. Se comprueba que la matriz generatriz estándar del código de partida y el obtenido son iguales:

```
show(C)
show(C.systematic_generator_matrix())
```

$[8, 5, 4]$ Generalized Reed-Solomon Code over $\mathbb{F}_{11}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 2 & 2 & 7 \\ 0 & 1 & 0 & 0 & 0 & 7 & 8 & 4 \\ 0 & 0 & 1 & 0 & 0 & 8 & 9 & 9 \\ 0 & 0 & 0 & 1 & 0 & 6 & 7 & 1 \\ 0 & 0 & 0 & 0 & 1 & 3 & 1 & 7 \end{pmatrix}$$

Figura 3.7: Matriz generatriz del código Reed-Solomon inicial

```
CCCC = codes.GeneralizedReedSolomonCode([F(a1) for a1 in parametros[0]],
k,[ F(b1) for b1 in parametros[1]]);
show(CCCC)
show(CCCC.systematic_generator_matrix())
```

$[8, 5, 4]$ Generalized Reed-Solomon Code over $\mathbb{F}_{11}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 2 & 2 & 7 \\ 0 & 1 & 0 & 0 & 0 & 7 & 8 & 4 \\ 0 & 0 & 1 & 0 & 0 & 8 & 9 & 9 \\ 0 & 0 & 0 & 1 & 0 & 6 & 7 & 1 \\ 0 & 0 & 0 & 0 & 1 & 3 & 1 & 7 \end{pmatrix}$$

Figura 3.8: Código Reed-Solomon a partir de la salida del ataque estándar

```
CCCC.systematic_generator_matrix() == C.systematic_generator_matrix()
```

```
True
```

Figura 3.9: Comparación de las matrices generatrices - ataque estándar

Efectivamente, se puede comprobar que dichas matrices son iguales.

Capítulo 4. Presupuestos

A continuación se ofrece una visión aproximada de los costes que tendría tanto la implementación como la investigación que requiere este proyecto. Analizando con detalle cada uno de los aspectos a tener en cuenta.

4.1. Personal

Actividad a realizar	**Días**	**Horas**	**Coste por hora***	**Subtotal**
Investigación	40	200	10,94 €	2.188,00 €
Desarrollo de la implementación (senior)	29	145	16,26 €	2.357,70 €
Gestión del proyecto	47	235	19,51 €	4.584,85 €
Total	116	580	-	9.130,55 €

Tabla 4.1: Sueldo del personal

*Importes extraídos del salario bruto medio en España [30], de desarrollador/a senior, gestor/a de proyectos y personal investigador, con una jornada laboral de 40 horas semanales.

4.2. Planificación de ejecución

Con el objetivo de aportar mayor compresión sobre la planificación de actividades a realizar, se adjunta un diagrama de Gantt, el cual ha sido creado gracias a la plataforma Tomplanner [31].

Figura 4.1: Planificación del proyecto

4.3. Material

En este apartado se tratan los recursos materiales necesarios para la realización del proyecto. En la siguiente tabla podemos visualizar los gastos mensuales de algunas de algunos de los productos, debajo se concretan los gastos cuyos costes no son mensuales.

Recurso	Especificaciones	Meses	Precio Mensual	Total
Cocalc	Licencia para un proyecto	4	4,10 €	16,40 €
Oficina trabajo	Espacio de coworking	5	350 €	1.750 €
Total	-	-	-	1.766,40 €

Tabla 4.2: Coste de los materiales mensuales

Para poder realizar este proyecto es necesario un ordenador en el que poder desarrollarlo como por ejemplo el portátil acer aspire 5 con un valor de 569,99 €. En cuanto al software necesario, a parte de la licencia del entorno virtual para poder trabajar con el lenguaje de programación Sage. El resto de aplicaciones utilizadas, como el editor de texto, consisten en herramientas gratuitas con lo que no suponen un incremento del presupuesto.

4.4. Coste total

Atendiendo a los importes calculados en los apartados anteriores, entre personal y material necesario, llevar a cabo el proyecto tendría un coste de 11.466,94€.

Item	**Recursos Materiales**	**Recursos Humanos**	**Coste del proyecto**
Total	1.766,40 €	9.130,55 €	11.466,94€

Tabla 4.3: Resumen de costes

A este precio habría que añadirle un 10% de coste de contingencia para cubrir cualquier imprevisto, con lo que el coste total quedaría en 12.613,63 €.

Capítulo 5. Conclusiones y líneas futuras

5.1. Conclusiones

Este proyecto se ha centrado en el estudio del criptosistema de McEliece, brindando la oportunidad de aprender sobre la teoría de corrección de errores. Además se ha estudiado la seguridad del sistema con la realización de un posible ataque construido a partir de los códigos Reed-Solomon generalizados.

A su vez se ha podido profundizar en conocimientos algebraicos que no se habían adquirido a lo largo de la carrera como son los de la teoría de códigos. Por otro lado, se ha ahondado en materias propias del grado como son la seguridad informática y el manejo de distintos lenguajes de programación, fomentando el interés en este ámbito.

Este trabajo 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 demostrado un alto nivel de seguridad. También se ha podido apreciar la importancia de seleccionar correctamente el tipo de código a utilizar en nuestro criptosistema, ya que esto puede marcar la diferencia en cuanto a la vulnerabilidad.

Para concluir, mencionar el gran impacto que tendría si mañana se anunciase la llegada de un ordenador cuántico lo suficientemente desarrollado como para romper el cifrado RSA tan utilizado en la actualidad. Es más que evidente que la seguridad de la información de las comunicaciones tiene una gran importancia pues cualquier persona realiza multitud de comunicaciones diarias empleando dispositivos electrónicos, las cuales contienen en ocasiones información muy personal. Esto muestra la importancia de ir pensando en cambiar o desarrollar algoritmos criptográficos resistentes a estos nuevos ordenadores cuánticos aunque puedan tener ciertos puntos débiles en cuanto a la eficiencia como es el caso del criptosistema de McEliece.

5.2. Trabajos futuros

Como se ha comentado, el camino de la criptografía y la computación cuántica no ha hecho más que comenzar. Se trata de un campo en pleno desarrollo y que sin duda alguna dará que hablar en un futuro no muy lejano.

Es importante que las personas que investigan en este campo sean capaces de encontrar soluciones, al menos momentáneas ante la posibilidad de que surjan ataques cuánticos, dado que la potencia de éstos sería capaz de romper muchos algoritmos de cifrado cuyo uso está muy extendido en nuestra vida como el algoritmo RSA.

En este sentido, en este proyecto se ha estudiado que el criptosistema de tipo McEliece

es un fuerte candidato para solucionar este problema. Sin embargo, sería interesante plantearse cuánto tiempo será capaz de resistir. Con los recursos que existen en la actualidad parece que aguantaría, pero parece cuestión de tiempo que los atacantes tengan los medios necesarios para encontrar un fallo de seguridad.

Otra de las opciones que habría que barajar una vez entendido el problema que existe y las posibles soluciones con las que contamos a día de hoy sería entrar de lleno en el estudio de algoritmos de cifrado cuánticos, es decir, una nueva serie de algoritmos de cifrado que utilicen la computación cuántica.

Capítulo 6. Conclusions and future works

6.1. Conclusions

This project has focused on the study of McEliece's cryptosystem, providing an opportunity to learn about error correction theory. In addition, the security of the system has been studied with the realization of a possible attack built from the generalized Reed-Solomon codes.

At the same time, it has been possible to delve into algebraic knowledge that had not been acquired throughout the career, such as code theory. On the other hand, it has delved into subjects of the degree such as computer security and the use of different programming languages, fostering interest in this area.

This work has served to increase awareness of the great potential of the McEliece system, a system that has been studied for many years and has always demonstrated a high level of safety. The importance of correctly selecting the type of code to use in our cryptosystem has also been appreciated, since this can make a difference in terms of vulnerability.

To conclude, mention the great impact it would have if the arrival of a quantum computer sufficiently developed to break the RSA encryption so widely used today is announced tomorrow. It is more than evident that the security of the information of the communications is of great importance since any person carries out a multitude of daily communications using electronic devices, which sometimes contain very personal information. This shows the importance of thinking about changing or developing cryptographic algorithms that are resistant to these new quantum computers, although they may have certain weak points in terms of efficiency, such as the McEliece cryptosystem.

6.2. Future works

As has been commented, the path of cryptography and quantum computing has only just begun. It is a field in full development and one that will undoubtedly be talked about in the not too distant future.

It is important that people who research in this field are able to find solutions, at least momentary to the possibility of quantum attacks, since their power would be capable of breaking many encryption algorithms whose use is very widespread in our lives. like the RSA algorithm.

In this sense, this project has studied that the McEliece-type cryptosystem is a strong candidate to solve this problem. However, it would be interesting to consider how long it will be able to resist. With the resources that exist today it seems that it would hold, but it

seems a matter of time before attackers have the means to find a security breach.

Another option that should be considered once the problem that exists and the possible solutions we have today is understood would be to enter fully into the study of quantum encryption algorithms, that is, a new series of encryption algorithms that use quantum computing.

Apéndice A

A.1. Manual

Para la implementación se ha hecho uso del entorno virtual cocalc, entorno que permite trabajar fácilmente con el lenguaje Sage, además de poder compartir cómodamente los ficheros.

Cuando se abra el enlace compartido [32] se nos mostrará una ventana como la siguiente, en dicha ventana se nos indica el contenido de la carpeta McEliece la cual podemos abrir haciendo clic en el botón que así lo indica.

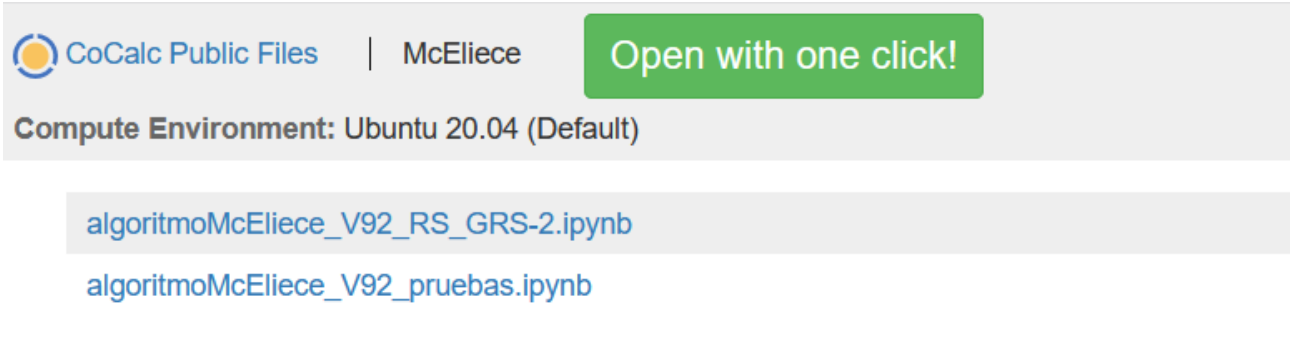

Figura A.1: Carpeta compartida en Cocalc

Cuando se nos cargue el entorno podremos abrir cualquiera de los ficheros que se nos muestran.

En el primer fichero “algoritmoMcEliece_V92_RS_GRS-2.ipynb” podemos encontrar la implementación relacionada con el criptoanálisis, mientras que en el otro fichero encontraremos la implementación del sistema de McElice.

Una vez seleccionado el fichero que deseamos visualizar en la parte superior se nos muestra una serie de opciones pudiendo seleccionar entre ejecutar celda a celda o todo en su conjunto.

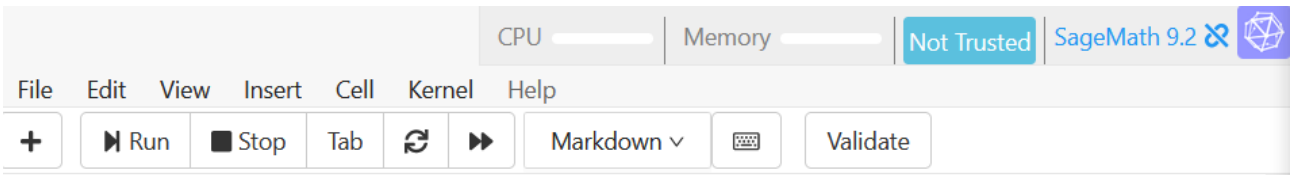

Figura A.2: Encabezado del entorno virtual

Para ejecutar paso por paso bastará con seleccionar el botón “Run”. Al mantener el cursor sobre el número de ejecución de la celda nos aparecerá una serie de opciones que podemos aplicar sobre esta. Esta opción nos da la posibilidad de ejecutar las celdas en el

orden que deseemos.

Para ejecutar todo el código a la vez debemos presionar el botón ⏭ se nos mostrarán dos alternativas, que la ejecución pare cuando se encuentre algún error o seguir ejecutando a pesar de que esto ocurra.

Se nos mostrará un menú como el siguiente.

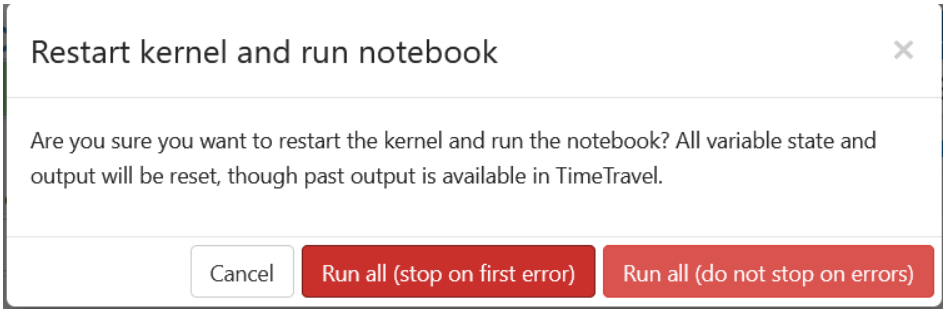

Figura A.3: Panel de ejecución

Cuando ejecutemos una celda se nos mostrará a la izquierda el número de ejecución y justo debajo del código la salida que mostremos.

Figura A.4: Ejemplo de celda del entorno

A la hora de cerrar el archivo bastará con cerrar la pestaña que se nos muestra justo encima del encabezado o en su defecto dirigirse al botón “Files” que se encuentra el primero a la derecha.

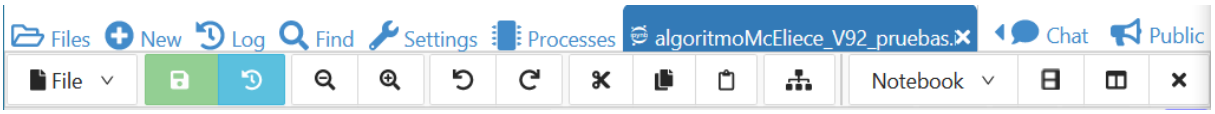

Figura A.5: Menú de opciones

Para concluir este apartado recordar que el orden de ejecución es importante ya que no podemos utilizar una función sin antes estar cargada. Por otro lado, dicha implementación también se encuentra alojada en la plataforma github en el enlace que podemos encontrar en la bibliografía [33].

Bibliografía

[1] E. W. Weisstein, RSA-640 Factored, MathWorld Headline New, 200.
URL: http://mathworld.wolfram.com/news/2005-11-08/rsa-640/

[2] P. Shor, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Compute, AT&T Research, 1996.

[3] NIST, National Institute of Standards and Technology.
URL: https://www.nist.gov/

[4] PQCrypto 2016
URL: https://pqcrypto2016.jp/

[5] SAFEcrypto. "Post-Quantum Crypto Lounge - SAFEcrypto”.
URL: https://www.safecrypto.eu/pqclounge/

[6] NIST Post-Quantum Cryptography
URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

[7] SageMath, the Sage Mathematics Software System
URL: http://www.sagemath.org.

[8] NumPy, The fundamental package for scientific computing with Python
URL: https://numpy.org/

[9] SymPy, Python library for symbolic mathematics
URL: https://www.sympy.org/en/index.html

[10] PARI/GP, Computer algebra system designed for fast computations in number theory
URL: https://pari.math.u-bordeaux.fr/

[11] Maxima, Sistema de álgebra computacional
URL: https://maxima.sourceforge.io/es/index.html

[12] Python, Lenguaje de programación
URL: https://www.python.org/

[13] CoCalc, Collaborative Calculation and Data Science
URL: https://cocalc.com

[14] Magma, Computational Algebra System
URL: http://magma.maths.usyd.edu.au/magma/

[15] Maplesoft, Software for Mathematics, online learning and engineering
URL: https://www.maplesoft.com/

[16] Wolfram Mathematica, Modern Technical Computing
URL: https://www.wolfram.com/mathematica/

[17] MATLAB, Plataforma de programación y cálculo numérico
URL: https://www.mathworks.com/products/matlab.html

[18] Cython, Lenguaje de programación
URL: https://cython.org/

[19] SageMath, Reed-Solomon codes and Generalized Reed-Solomon codes
URL: https://doc.sagemath.org/html/en/reference/coding/sage/coding/grs_code.html

[20] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory,1978. Jet Propulsion Laboratory DSN Progress Report 42–44.

[21] Pino Caballero Gil. Introducción a la Criptografía, edición 2, ilustrada, Editor RA-MA S.A. Editorial y Publicaciones, 2002

[22] Shoup, Victor, A Computational Introduction to Number Theory and Algebra, Cambridge University 2008

[23] Berlekamp. E, Goppa Codes, IEEE Transactions on Information Theory 1975.

[24] N. J. Patterson. The algebraic decoding of Goppa codes, IEEE Transactions on Information Theory21, 203207, 1975

[25] Ashley Valentijn. “Goppa Codes and Their Use in the McEliece Cryptosystems", Syracuse Universitym, SURFACE Honors Program Project. Spring 5-1-2005

[26] Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Problems of Control and Information Theory 15, 19–34 (1986)

[27] V. M. Sidelnikov y S. O. Shestakov, On the insecurity of cryptosystems based on generalized Reed-Solomon codes, Discrete Math. Appl 1992.

[28] A. Couvreur, P. Gaborit, V. Gauthier-umaña, A. Otmani, J-P. Tillich, Distinguisher-Based Attacks on Public-Key Cryptosystems Using Reed-Solomon Codes, 2014.

[29] A. Couvreur, A. Otmani, J-P. Tillich, Polynomial time attack onwild McEliece over quadratic extensions, IEEE Transactions on Infor-mation Theory 2016

[30] Indeed, Bolsa de trabajo
URL: https://es.indeed.com/

[31] Tom’s Planner, Planificador de proyectos | Diagrama de Gantt
URL: https://www.tomsplanner.es/

[32] CoCalc, Proyecto sistema de Mc Eliece
URL:
https://cocalc.com/share/6b926e4eff03a1161df03368c4abe85e36c60443/McEliece/?viewer=share

[33] Github, Repositorio del proyecto sobre el criptosistema de McEliece
URL: https://github.com/Martadosptocero/McEliece