UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Criptosistema de McEliece/Niederreiter con códigos skew Goppa

Autor
Gonzalo Sanz Guerrero

Director
Francisco Javier Lobillo Borrero

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, 6 de septiembre de 2024

Criptosistema de McEliece/Niederreiter con códigos skew Goppa

Autor
Gonzalo Sanz Guerrero

Director
Francisco Javier Lobillo Borrero

Departamento de Álgebra

—

Granada, 6 de septiembre de 2024

Criptosistema de McEliece/Niederreiter con códigos skew Goppa

Gonzalo Sanz Guerrero

Palabras clave: programación, criptografía, criptosistema, clave pública, McEliece, Niederreiter, skew Goppa, encapsular, desencapsular, python, cuerpo finito

Resumen

La criptografía, originada en tiempos antiguos como una técnica para proteger la información, ha evolucionado significativamente a lo largo de los siglos, adaptándose a los desafíos planteados por el avance de la tecnología. Desde los cifrados clásicos hasta la aparición de la criptografía moderna basada en clave pública, la necesidad de proteger la información confidencial ha impulsado el desarrollo de algoritmos cada vez más complejos y seguros.

En la era digital, estos sistemas han sido fundamentales para asegurar las comunicaciones y transacciones electrónicas. Sin embargo, con el desarrollo de la computación cuántica, las bases matemáticas de muchos de los criptosistemas actuales están siendo cuestionadas. Los algoritmos cuánticos amenazan con romper la seguridad de los sistemas criptográficos tradicionales, lo que podría poner en riesgo la confidencialidad y la integridad de los datos en la era post-cuántica.

Ante esta amenaza, surge la necesidad de desarrollar y adoptar criptosistemas post-cuánticos que sean resistentes a los ataques cuánticos. Entre ellos, los criptosistemas basados en clave pública, como el de McEliece y su variante de Niederreiter, han demostrado ser candidatos prometedores debido a su robustez y resistencia a las capacidades de computación cuántica.

Este trabajo se centra en la implementación del Criptosistema de McEliece/Niederreiter utilizando códigos skew Goppa en Python, como una respuesta a la necesidad de encontrar alternativas seguras para la criptografía en un futuro post-cuántico. A través de un análisis exhaustivo y una implementación práctica, se exploran las fortalezas y debilidades de este enfoque, ofreciendo una visión sobre cómo la criptografía debe evolucionar para mantenerse efectiva en un mundo donde la computación cuántica será una realidad

Criptosistema de McEliece/Niederreiter con códigos skew Goppa

Gonzalo Sanz Guerrero

Keywords: programming, cryptosystem, cryptography, public key, McEliece, Niederreiter, skew Goppa, encoding, decoding, Python, finite field

Abstract

Cryptography, originating in ancient times as a technique to protect information, has evolved significantly over the centuries, adapting to the challenges posed by advancing technology. From classical ciphers to the emergence of modern public-key cryptography, the need to protect confidential information has driven the development of increasingly complex and secure algorithms.

In the digital age, these systems have been fundamental in securing electronic communications and transactions. However, with the development of quantum computing, the mathematical foundations of many current cryptosystems are being challenged. Quantum algorithms threaten to break the security of traditional cryptographic systems, potentially compromising the confidentiality and integrity of data in the post-quantum era.

In response to this threat, the need arises to develop and adopt post-quantum cryptosystems that are resistant to quantum attacks. Among them, public-key cryptosystems such as McEliece and its Niederreiter variant have shown promise due to their robustness and resistance to quantum computing capabilities.

This work focuses on the implementation of the McEliece/Niederreiter Cryptosystem using skew Goppa codes in Python, as a response to the need to find secure alternatives for cryptography in a post-quantum future. Through a comprehensive analysis and practical implementation, the strengths and weaknesses of this approach are explored, providing insight into how cryptography must evolve to remain effective in a world where quantum computing will become a reality.

Yo, **Gonzalo Sanz Guerrero**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77147456M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Gonzalo Sanz Guerrero

Granada a 6 de septiembre de 2024 .

D. **Francisco Javier Lobillo Borrero**, Catedrático del Departamento de Álgebra de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Criptosistema de McEliece/Niederreiter con códigos skew Goppa***, ha sido realizado bajo su supervisión por **Gonzalo Sanz Guerrero**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 6 de septiembre de 2024.

El director:

Francisco Javier Lobillo Borrero

Agradecimientos

Agradecimientos

Quiero expresar mi más profundo agradecimiento a todas las personas que han hecho posible la realización de este Trabajo de Fin de Grado. Me gustaría agradecer de la manera más sincera posible a todas las personas que han contribuido a mejorar mi experiencia universitaria.

En primer lugar, agradezco de manera especial a mis padres, por haberme hecho ser quien soy y haberme dado cariño y apoyo en todos los pasos que he ido dando en mi vida. También a mis hermanos, que han sido otro gran apoyo e inspiración tanto en mi vida académica como personal.

A mis amigos, que han sido un pilar fundamental durante este proceso. Gracias por cada momento de felicidad, por la ayuda y por las risas. Sin ellas, estos 4 años no hubieran sido lo mismo.

Finalmente, quiero expresar un especial agradecimiento a mi tutor del TFG, Francisco Javier Lobillo Borrero. Su ayuda, paciencia y conocimientos han sido clave en la realización de este proyecto. Agradezco profundamente su dedicación y apoyo durante todo el desarrollo de este trabajo. Sin él, este trabajo no hubiera sido posible.

A todos vosotros, muchas gracias.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En este primer capítulo vamos a introducir el proyecto, por lo que hablaremos del contexto en el que se desarrolla el proyecto y la motivación para hacerlo, de los diferentes objetivos del proyecto, de la planificación del proyecto y finalmente de cómo está estructurada esta memoria.

1.1. Contexto y motivación

La criptografía es esencial para garantizar la confidencialidad y seguridad de la información. A día de hoy, hay diferentes algoritmos de cifrado y descifrado que nos permiten evitar que posibles atacantes se hagan con nuestra información. No obstante, la criptografía ahora se enfrenta a un nuevo problema con la futura llegada de los ordenadores cuánticos, pues se cree que estos serían capaces de vulnerar los algoritmos criptográficos actuales.

Debido a esto, se está preparando una transición desde la criptografía convencional a la criptografía post-cuántica, buscando nuevos algoritmos que sean capaces de resistir ataques realizados por ordenadores cuánticos. Por ejemplo, los ordenadores cuánticos serían capaces de ejecutar el algoritmo de Shor de forma eficiente, que factoriza números enteros grandes en sus componentes primos en tiempo polinomial, originando que algoritmos criptográficos que basan su seguridad en la alta dificultad de realizar esta tarea, como el RSA, dejen de ser seguros. Véase [5].

En este contexto, está ganando fuerza el criptosistema propuesto por McEliece en 1978 posteriormente modificado por Niederreiter en 1986, que se basa en códigos correctores de errores. Este criptosistema no tuvo mucha acogida inicial debido a que el tamaño de la clave secreto es muy grande, pero en estos últimos años se ha retomado el interés por él debido a que es resistente a ataques con ordenadores cuánticos. Véase [6].

Es por ello que la principal *motivación* para de este proyecto es realizar la implementación de un criptosistema capaz de resistir ataques de ordenadores cuánticos, como lo es el de McEliece/Niederreiter con códigos skew Goppa, contribuyendo así a la corriente de criptografía post-cuántica.

Además, hay una motivación personal, ya que fue el estudio de la ciberseguridad durante la etapa preuniversitaria lo que hizo que decidiera estudiar Ingeniería Informática. Por ello, buscaba realizar un proyecto que me ayudara a profundizar mis conocimientos en el ámbito de la ciberseguridad y la criptografía.

1.2. Objetivos

En este apartado se definen los principales *objetivos del proyecto*:

- Mejorar la habilidad de programación y la compresión del lenguaje Python.
- Análisis y estudio del criptosistema de McEliece/Niederreiter, comprendiendo su funcionamiento y sus características.
- Análisis y estudio de la propuesta de integración de códigos skew Goppa al criptosistema de McEliece/Niederreiter.
- Diseño de un sistema que implemente la propuesta de integración de códigos skew Goppa al criptosistema de McEliece/Niederreiter.
- Implementación del sistema que implementa la propuesta de integración de códigos skew Goppa al criptosistema de McEliece/Niederreiter.
- Realización de pruebas que corroboren la validez del sistema y la propuesta.

1.3. Planificación del proyecto

En este apartado, se presentará la planificación inicial del proyecto. Se puede observar un notable salto temporal desde octubre de 2023, cuando concerté el TFG con mi tutor, hasta febrero de 2024, cuando comencé la realización del mismo al inicio del segundo cuatrimestre:

1. **Planteamiento y aceptación (9 días)**: durante esta fase concerté la realización del proyecto con mi tutor y establecimos que el tema sería la implementación del criptosistema de McEliece/Niederreiter con códigos skew Goppa.

2. **Estudio y selección de las herramientas disponibles para la realización el proyecto (5 días)**: en esta segunda fase se investigó qué posibilidades había para implementar el proyecto, incluyendo lenguajes de programación, entornos de desarrollo, librerías... Una vez conocidas las posibilidades, se seleccionaron las herramientas que se usarían.

3. **Estudio y comprensión de los conceptos a implementar (50 días)**: durante esta fase estudié los conceptos que debía implementar posteriormente para completar el proyecto. Se divide en:

   - Estudio del criptosistema de McEliece/Niederreiter (15 días): en esta fase comprendí cómo funciona el criptosistema de McEliece/Niederreiter: la generación de claves, el encapsulado y el desencapsulado.
   - Estudio de los códigos Goppa y skew Goppa (17 días): recopilación, estudio y comprensión de información sobre los códigos Goppa y skew Goppa y su uso en criptografía.
   - Estudio de la propuesta de inclusión de códigos skew Goppa en el criptosistema de McEliece/Niederreiter (12 días): estudio de la propuesta de incluir códigos skew Goppa en el criptosistema de McEliece/Niederreiter.

4. **Diseño e implementación (92 días)**: durante esta fase diseñé el sistema e implementé el código. Se divide en:

   - Diseño del sistema que implementa el proyecto (16 días): recopilación de requisitos y diseño de la estructura del sistema.
   - Implementación y corrección de errores en el código (71 días): implementación del sistema previamente diseñado siguiendo las pautas de la propuesta y corrección de errores para garantizar un correcto funcionamiento.

5. **Pruebas de funcionamiento (10 días)**: en esta quinta fase se realizaron pruebas de funcionamiento para corroborar que el criptosistema está bien implementado

6. **Redacción y entrega de la memoria (37 días)**: en esta última fase se redactó la memoria que explica el desarrollo del proyecto y se preparó la entrega del proyecto. Se divide en:

   - Redacción de la memoria (37 días): redacción y revisión de la memoria explicativa del proyecto.
   - Preparación del proyecto para su entrega (5 días): preparación del código y de la memoria para su entrega.

En el siguiente diagrama de Gantt realizado con la herramienta GanttProject [7] se puede observar la planificación:

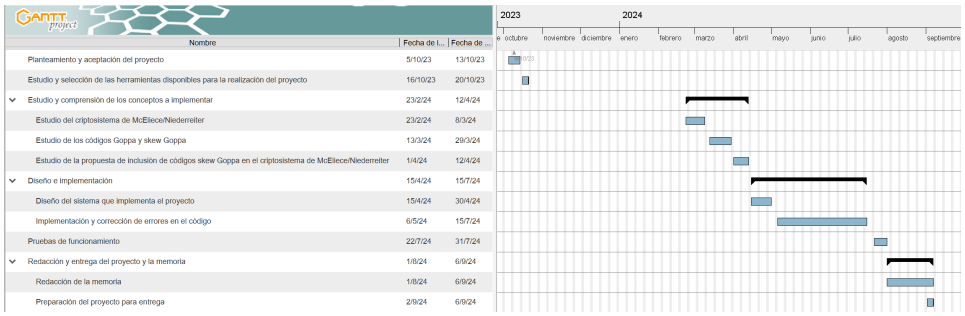

Figura 1.1: Diagrama de Gantt con la planificación inicial del proyecto

1.4. Estructura de la memoria

En esta sección se va a explicar cómo está estructurada esta memoria explicando brevemente cada capítulo:

1. **Introducción**: este primero capítulo introduce el proyecto y está compuesto por las siguientes secciones:
   - Contexto y motivación: explica el ámbito en el que se realiza el proyecto y el problema que se quiere resolver con él.
   - Objetivos: explica los diferentes objetivos que tiene el proyecto.
   - Planificación: explica la planificación temporal inicial del proyecto.
   - Estructura de la memoria: explica los capítulos que tiene esta memoria y su contenido
2. **Estado del arte**: este capítulo es una revisión de la literatura y los avances actuales de la temática del proyecto y está compuesto por las siguientes secciones:
   - Criptosistemas basados en clave pública: explica el uso de los diferentes criptosistemas de clave pública en la actualidad.
   - Criptografía post-cuántica: explica el estado actual de la criptografía post-cuántica y su proyección en el futuro.
   - Herramientas y tecnologías disponibles para la realización del proyecto: explica las diferentes herramientas disponibles para la realización del proyecto.

- Herramientas y tecnologías seleccionadas para la realización del proyecto: explica las herramientas seleccionadas para la realización del proyecto y los motivos de su selección.

3. **Fundamentos teóricos**: este tercer capítulo explica los conceptos teóricos sobre los que se basa el proyecto. Se divide en:

   - Criptografía: introducción al concepto de criptografía .
   - Cuerpos finitos: explica el concepto matemático de los cuerpos finitos y sus aplicaciones en criptografía.
   - Códigos Goppa y skew Goppa: explica el concepto de los códigos Goppa y su variante skew Goppa y su relación con la criptografía.
   - Criptosistema de McEliece/Niederreiter: explica cómo funciona el criptosistema de McEliece y su posterior variante de Niederreiter.
   - Inclusión de códigos skew Goppa en el criptosistema McEliece/Niederreiter: explica cómo incluir los códigos skew Goppa en el criptosistema de McEliece/Niederreiter.

4. **Diseño e implementación**: este cuarto capítulo expone el diseño del sistema y su posterior implementación. Se divide en:

   - Diseño de la solución: se explica el proceso seguido para el diseño del sistema.
   - Implementación de la solución: explicación de la implementación realizada y sus partes.

5. **Resultados**: el quinto capítulo muestra los resultados de las ejecuciones y realiza un análisis de ellos. Se divide en:

   - Resultados: muestra el resultado de diferentes ejecuciones del sistema.
   - Análisis de los resultados: realiza un análisis de los resultados de estas ejecuciones.

6. **Conclusiones**: este último capítulo expone las conclusiones a las que se han llegado tras la realización del proyecto. Se divide en:

   - Conclusiones del proyecto: se muestran algunas conclusiones a las que se han llegado una vez finalizado el proyecto.
   - Futuras líneas de investigación: se mencionan posibles caminos por los que seguir investigando o trabajando a partir de este proyecto.

- **Apéndice A: Repositorio del Proyecto en GitHub**: se describe el repositorio de GitHub donde se ha subido el código del proyecto y se explica su contenido y cómo descargarlo para su uso.
- **Apéndice B: Manual de usuario**: explica cómo utilizar el sistema creado en este proyecto.
- **Apéndice C: Instalación del entorno de desarrollo**: explica cómo instalar el entorno de desarrollo que se ha utilizado durante la realización del proyecto.
- **Apéndice D: Instalación de las librerías necesarias**: explica cómo instalar las librerías que se han utilizado en este proyecto.

Capítulo 2

Estado del Arte

En este segundo capítulo se va a hablar sobre las diferentes opciones que existen dentro de la criptografía de clave pública, así como de qué alternativas hay en el campo de la criptografía post-cuántica actualmente. Por último, explicaremos qué diferentes herramientas hay disponibles para la realización de este proyecto y cuáles fueron las elegidas finalmente para ello.

2.1. Criptosistemas basados en clave pública

La criptografía de clave pública o de clave asimétrica utiliza un algoritmo asimétrico para generar una clave pública y una privada. Este tipo de criptografía garantiza la confidencialidad, ya que la información se cifra usando la clave pública de una persona y solo puede descifrarse usando la clave privada de esa persona, por lo que solo la persona a la que está destinada el mensaje puede descifrarlo. Véase [8]

.

Entre los sistemas de cifrado de clave pública más utilizados podemos encontrar el RSA, los sistemas de curva elíptica (ECC) y el método Diffie-Hellman:

- Método Diffie-Hellman: este método criptográfico permite a 2 partes el intercambio de un secreto compartido por un canal público sin previo conocimiento entre sí. Se basa en la dificultad de calcular logaritmos discretos en un cuerpo finito, lo que proporciona un alto nivel de seguridad.

  Este sistema de cifrado permite a los usuarios establecer un canal de comunicación cifrado sin necesidad de exponer las claves a posibles atacantes. Hoy en día se utiliza en diversidad de aplicaciones y sistemas, desde el e-mail hasta el establecimiento de conexiones seguras en VPNs. Véase [9].

- RSA: el algoritmo de cifrado RSA es un algoritmo de clave pública basado en el problema matemático de la factorización de números enteros con factores primos grandes. Fue propuesto por Rivest, Shamir y Adleman en 1978. Permite el envío de mensajes sin tener que intercambiar la clave privada, así como realizar firmas digitales. Véase [10].

- Sistemas de cifrado de curva elíptica (ECC): este tipo de criptografía de clave pública se basa en la teoría de curvas elípticas. Permite intercambiar claves, firmar documentos digitales o cifrar de manera segura. La seguridad de la criptografía de curva elíptica se basa en la dificultad de resolver el problema del logaritmo discreto en el grupo asociado a una la curva elíptica.

  Además, la ECC presenta ventajas frente a RSA, como el requerimiento de menos recursos computacionales para la generación de claves, el cifrado y el descifrado. Como la longitud de clave es menor, las operaciones de ECC pueden realizarse más rápido que las de RSA. Véase [11].

2.2. Criptografía post-cuántica

La *criptografía post-cuántica (PQC)* se refiere a aquellos sistemas criptográficos que están diseñados para soportar ataques de ordenadores cuánticos. Con el auge de los ordenadores cuánticos que podrían vulnerar la seguridad de algoritmos de cifrado como el RSA, los criptosistemas post-cuánticos buscan reemplazarlos o complementarlos para evitar la amenaza de la computación cuántica. Véase [5].

Algunos algoritmos de cifrado en el ámbito post-cuántico que se están desarrollando son:

- Criptografía basada en retículo (LBC): se basa en la dificultad de resolver el problema del vector más corto. Son rápidos y usan claves pequeñas, pero aún no han sido sometidos a pruebas exhaustivas para probar su seguridad.

- Criptografía basada en códigos: esta basada en la dificultad de resolver un código lineal. Puede cifrar de manera muy rápida, pero se necesitan claves muy grandes.

- Criptografía basada en isogenias: esta basada en la dificultad de encontrar una isogenia (mapeo) entre dos curvas elípticas. Es óptima en términos de almacenamiento y transmisión porque tiene clave pequeña, pero aún no ha sido sometida a las suficientes pruebas como para comprobar su robustez.

- Criptografía basada en ecuaciones multivariadas: está basada en la dificultad de resolver sistemas de ecuaciones polinómicas con múltiples variables. Tiene claves privadas cortas, pero las públicas son excesivamente grandes.

- Criptografía basada en funciones Hash: está basada en las propiedades de las funciones unidireccionales (como SHA-3). Tiene claves muy pequeñas y esquemas muy seguros, pero es lento.

Además, el *NIST* (National Institute of Standards and Technology) inició en 2016 un concurso para seleccionar nuevos algoritmos de cifrado de clave pública que resistieran ataques de computación cuántica. El objetivo es que estos algoritmos acaben sustituyendo a los algoritmos de criptografía de clave pública convencionales (como RSA). El mecanismo de encapsulación de clave pública seleccionado fue el CRYSTALS-KYBER. Aún así el proceso para escoger un esquema de firma digital aún sigue abierto y se consideran 3 esquemas: CRYSTALS-Dilithium, FALCON y SPHINCS+. Véase [12, 13]

2.3. Herramientas y tecnologías disponibles y seleccionadas para la realización del proyecto

Para el desarrollo del proyecto tenemos muchas herramientas diferentes que nos van a permitir la implementación del criptosistema. Debemos escoger primero un lenguaje de programación. Una vez escogido, debemos elegir herramientas que nos permitan trabajar con cuerpos finitos, archivos, hashings...

La lista de candidatos está compuesta por C++, Java y Python (véase [14]):

- C++: es un lenguaje potente que permite escribir código eficiente de alto rendimiento. Además, cuenta con un gran número de bibliotecas estándar. Aún así, es un lenguaje más complejo, (por ejemplo tiene gestión manual de la memoria), por lo que se tarda más en escribir código C++.

- Java: en un lenguaje robusto y versátil, que permite la portabilidad de código entre plataformas y cuenta con una amplia gama de bibliotecas y herramientas. No obstante, Java se ejecuta más lento debido a que depende de una máquina virtual (JVM) que interpreta su código de bytes.

- Python: este lenguaje es conocido por su simplicidad, legibilidad y su facilidad de uso, lo que permite el desarrollo rápido. La gama de bibliotecas y herramientas de python es de las más completas. Aún así, es un lenguaje interpretado, por lo que puede ejecutarse más lento y no es capaz de lograr un paralelismo verdadero.

Finalmente, se seleccionó **python** ya que python es el lenguaje más simple de los 3, lo que nos permitía un desarrollo rápido del proyecto. Además, cuenta con una amplia gama de bibliotecas centradas en matemáticas, criptografía y álgebra.

Ahora, debemos seleccionar una herramienta que nos permita trabajar con cuerpos finitos en python, pues nos harán falta. Aquí encontramos dos alternativas:

- SageMath: SageMath es un sistema matemático muy potente que permite trabajar con cuerpos finitos y está basado en python. Véase [15].

- Galois: esta librería es una extensión de la librería numpy que permite trabajar con cuerpos finitos de manera más sencilla. Véase [16].

- PyFinite: biblioteca de Python diseñada para trabajar con campos finitos y realizar operaciones matemáticas relacionadas. También incluye un paquete genérico para hacer operaciones con matrices sobre campos genéricos. Véase [17].

Vamos a quedarnos con *Galois*, ya que aunque SageMath es mas poderosa y nos permitiría realizar todas las operaciones con cuerpos finitos, tiene demasiadas dependencias y es demasiado pesada para la funcionalidad que queremos implementar. Además, Galois tiene más funcionalidad desarrollada que PyFinite.

Además, entre las bibliotecas de python que vamos a necesitar usar, podemos encontrar:

- Numpy: permite realizar operaciones de computación científica en python, proporcionando soporte para arrays multi dimensionales o matrices y funciones matemáticas más complejas para operar con los datos. También ofrece varias formas para guardar arrays en archivos, lo que nos será de gran utilidad. Véase [18].
- Sys: da acceso a variables usadas o mantenidas por el intérprete y a funciones que interactúan fuertemente con el intérprete. Véase [19].
- Time: permite trabajar con el tiempo. En este caso utilizamos cronómetros para medir el tiempo de ejecución de los programas. Véase [20].
- Math: permite trabajar con diferentes funciones matemáticas. Véase [21].
- Random: permite generar números aleatorios y seleccionar elementos aleatorios de un conjunto. Véase [22].
- Hashlib: módulo estándar de python que permite trabajar con gran multitud de algoritmos de hashing en python, como el MD5 o el SHA-3. Véase [23].

Capítulo 3

Fundamentos teóricos

En este capítulo de la memoria se explicarán los fundamentos teóricos sobre los que se basa este proyecto. Entre ellos podemos encontrar la criptografía, donde destacamos la criptografía basada en clave pública, los cuerpos finitos, los códigos skew Goppa, el criptosistema de McEliece/Niederreiter y su inclusión en el criptosistema de McEliece/Niederreiter.

3.1. Criptografía

> La **Criptografía** es una práctica que consiste en proteger la información mediante el uso de algoritmos codificados, hashes y firmas. Véase [24].

La información protegida puede estar en reposo (un archivo ubicado en un disco duro), en tránsito (una comunicación electrónica intercambiada entre 2 o más partes) o en uso (se están ejecutando operaciones sobre estos datos).

La criptografía tiene 4 *objetivos* principales:

- Confidencialidad: poner la información únicamente a disposición de los usuarios autorizados.
- Integridad: asegurar que la información no se ha manipulado.
- Autenticación: confirmar la autenticidad de la información o de la identidad de un usuario.
- No repudio: evitar que un usuario deniegue compromisos o acciones previas.
  Véase [24].

La criptografía cumple un rol muy importante en la actualidad, ya que permite que los datos confidenciales (como mensajes o datos de una transacción online) permanezcan seguros y confidenciales. Además, la criptografía avanzada es crucial para mantener la seguridad nacional, debido a que protege la información clasificada frente a adversarios y posibles ataques.
Véase [25].

La criptografía tiene múltiples *usos*, entre los que se pueden destacar:

- Contraseñas: la criptografía se utiliza a menudo para la autenticación y para proteger las contraseñas almacenadas. Esto permite que los servicios de autenticación puedan comprobar la validez de las contraseñas sin usar un archivo de texto plano con las contraseñas, ya que estos son muy vulnerables frente a posibles ataques.

- Criptomonedas: algunas de las criptomonedas más famosas (como Bitcoin o Ethereum) están basadas en sistemas de cifrado complejos que requieren una gran potencia de cálculo para ser descifrados. Además, las criptomonedas hacen uso de la criptografía avanzada para proteger las criptocarteras y verificar las transacciones, evitando así posibles fraudes.

- Navegación web segura: la criptografía protege a los usuarios de ataques de intermediario (MitM) y de posibles escuchas mientras navegan en la web. Algunos protocolos como SSL (Secure Socket Layers) o TLS (Transport Layer Security) se basan en criptografía para proteger datos enviados entre servidor web y cliente en canales de comunicación seguros.

- Firmas electrónicas: las firmas electrónicas, utilizadas para documentos de gran importancia en línea, suelen aplicarse por ley. Las firmas electrónicas creadas mediante el uso de la criptografía se pueden validar para evitar fraudes.

- Autenticación: cuando es necesario autenticar la identidad de un usuario, se hace uso de la criptografía para confirmar la identidad y autenticar los privilegios de acceso de dicho usuario.

- Comunicaciones seguras: a la hora de compartir información clasificada o mantener una conversación privada, el cifrado de extremo a extremo protege las comunicaciones bidireccionales. Este proporciona un nivel alto de seguridad y privacidad a los usuarios.
  Véase [25].

3.1.1. Tipos de criptografía

Dentro de la criptografía podemos considerar diferentes tipos según cómo se realiza el cifrado. Entre ellos podemos destacar:

Criptografía simétrica

La criptografía simétrica utiliza la misma clave tanto para cifrar como para descifrar el mensaje que se enviar de manera segura. Por ello, tanto el receptor como el emisor deben conocer la clave. La figura 3.1 muestra cómo funciona el cifrado simétrico:

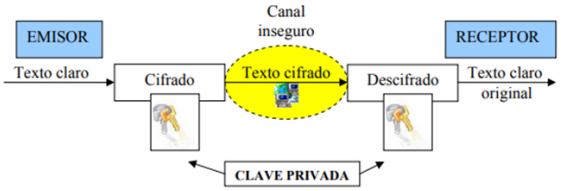

Figura 3.1: Funcionamiento del cifrado simétrico, véase [1].

Criptografía asimétrica

La criptografía asimétrica hace uso de dos claves diferentes en el proceso de cifrado/descifrado.

- Clave pública: esta clave se puede difundir sin problema a todas las personas que quieran mandar información cifrada.
- Clave privada: esta clave no se debe difundir ya que con ella es con la que se realiza el descifrado. La clave pública está vinculada a la privada para poder hacer uso de ellas en el cifrado/descifrado de una manera correcta.

La figura 3.2 muestra cómo es el proceso:

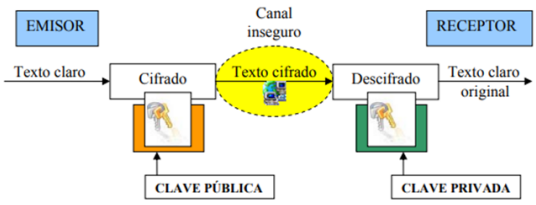

Figura 3.2: Funcionamiento del cifrado asimétrico, véase [1].

Podemos entender mejor la criptografía asimétrica con el siguiente *ejemplo*. Si 3 amigos nuestros nos quieren mandar un mensaje privado, ellos deben tener nuestra clave pública para poder cifrar el mensaje. Se la mandamos ya que no hay problema en compartir la clave pública. Ellos cifran el archivo y nos lo mandan cifrado de manera segura. Nosotros hacemos uso de la clave privada para descifrarlo y ver el mensaje.

La criptografía asimétrica tiene *ventajas con respecto a la simétrica*, como pueden ser la seguridad (ya que permite difundir la clave pública sin vulnerar la seguridad del sistema de cifrado/descifrado porque la clave privada no se difunde) y el número de claves necesarias (en el cifrado asimétrico solo hay que generar un par de claves por usuario y en el simétrico hay que crear una nueva clave por cada usuario que quiera mandarnos información).

Aún así, la criptografía asimétrica tiene una *desventaja frente a la simétrica*. La criptografía simétrica es más rápida y ágil que la asimétrica. Esto nos permite ahorrar tiempo si necesitamos intercambiar una gran cantidad de información de manera segura. Véase [1].

Criptografía híbrida

Debido a que los algoritmos de cifrado asimétricos son menos eficientes que los simétricos, no se suelen usar directamente sobre la información. Para proteger la información se utiliza el cifrado simétrico, y el asimétrico se suele usar para compartir las claves. Es decir, se utiliza la clave pública para encapsular una clave de sesión de un criptosistema simétrico y se utiliza la clave privada para desencapsular. Esto se conoce como *cifrado híbrido*.

El cifrado híbrido utiliza las propiedades del cifrado asimétrico para generar las claves y transmitirlas de manera segura por un canal no seguro para hacer uso de la eficiencia del cifrado simétrico de una manera más segura. Este tipo de criptografía es muy usado en los protocolos de transferencia de datos en la web. Véase [24].

3.1.2. Criptografía post-cuántica

> La **criptografía post-cuántica** (también conocida como criptografía a prueba de cuántica, segura para la cuántica o resistente a la cuántica) se refiere a los algoritmos criptográficos (a menudo algoritmos de clave pública) que se espera que sean seguros contra un asalto criptoanalítico por parte de un ordenador cuántico. Véase [26].

La *computación cuántica* es un gran avance en el mundo de la informática. A diferencia de la computación tradicional (que utiliza bits para representar la información), esta utiliza qubits para representar la información. Los qubits tienen la capacidad de representar simultáneamente tanto el 0 como el 1, mediante un estado de superposición. Esto permite a los ordenadores cuánticos realizar múltiples operaciones simultáneamente, otorgándoles una capacidad de procesamiento exponencialmente más rápido que los ordenadores clásicos.

Es por esta mayor capacidad de cálculo de los ordenadores cuánticos que se teme que muchos de los algoritmos criptográficos de clave pública más utilizados (como el Rivest, Shamir y Adleman (RSA) o el Diffie-Hellman (DH)) podrían ser vulnerables frente a ataques de este tipo de ordenadores. Un ejemplo de esto es que el algoritmo de Shor podría ser capaz de romper la criptografía de clave pública, como RSA y DH. Por ejemplo, este algoritmo ejecutado en un ordenador cuántico tardaría menos de 1 día en descifrar una implementado del RSA de 1024 bits. No se espera que los ordenadores cuánticos lleguen definitivamente hasta dentro de un plazo de entre 10 a 30 años, pero aún no se ha encontrado una solución concreta para los ataques cuánticos, por lo que encontrar una solución a este problema es algo de vital importancia. Véase [27] [28].

Por ello, la *criptografía post-cuántica* (PQC, de sus siglas en inglés post-quantum cryptography), busca reemplazar o complementar a los sistemas criptográficos de clave pública actuales para evitar ser vulnerables frente a ataques cuánticos. Véase [27].

3.2. Cuerpos finitos

> Un **cuerpo finito** es un conjunto finito de elementos para los cuales la suma y multiplicación están bien definidas y se cumple con las condiciones de los axiomas de cuerpo. Los cuerpos finitos son llamados también cuerpos de Galois, y a menudo se usan en criptografía y comprobación de errores. Véase [29].

Los cuerpos finitos cumplen con las siguientes *características*:

- Las operaciones fundamentales de los cuerpos finitos son la adición y la multiplicación, teniendo como elementos neutros el 0 en la suma y el 1 en la multiplicación.

- Los cuerpos finitos cumplen con las propiedades de cierre (la suma y la multiplicación realizada entre dos elementos de un cuerpo resulta en un elemento de ese cuerpo), asociatividad (la suma y la multiplicación en cuerpos finitos son asociativas), y la distributividad (la suma y multiplicación en un cuerpo finito son distributivas). Véase [30].

Existe un único cuerpo finito de orden p^n para cada primo p y cada entero positivo n, pero este único cuerpo puede ser representado de varias formas (son cuerpos isomorfos)(véase [31]). Los cuerpos finitos también son llamados cuerpos de Galois debido a Évariste Galois, uno de los primeros matemáticos en investigarlos. Véase [32].

La *característica* de un cuerpo finito es el menor número entero positivo n tal que $n = 0$. Si n no existe, entonces se dice que es la característica del cuerpo es 0. También debemos tener en cuenta que todo cuerpo finito debe tener característica p, donde p debe ser un número primo. La extensión de un cuerpo es un (K, F) donde F es un cuerpo y K es un subcuerpo de F. Las extensiones se denotarán como F/K. Véase [31].

Si un elemento del cuerpo F_q, siendo $q = p^n$, genera el grupo multiplicativo F_q^* de las unidades de F_q, este elemento se define como el *elemento primitivo del cuerpo*. Este elemento primitivo es capaz de generar todos los elementos del cuerpo a través de potencias. Todos los cuerpos finitos tienen al menos un elemento primitivo. Véase [31].

Un polinomio $f(x) \in F[x]$ es irreducible sobre un cuerpo $Fsif(x$ no puede ser expresado como producto de otros dos polinomios de $F[x]$ cuyos grados son menores que el grado de $f(x$. Estos polinomios irreducibles son importantes, pues se usan para definir las extensiones de los cuerpos finitos. Véase [32].

Los cuerpos finitos son importantes en el ámbito de la criptografía, pues son la base sobre la que se construyen numerosos algoritmos criptográficos. Estos algoritmos usan las características de los cuerpos finitos para asegurar una comunicación segura. Otro ámbito donde cobran gran importancia es el de los códigos de corrección de errores. Estos se utilizan para detectar y corregir posibles errores generados durante la transmisión. Estos códigos se basan en las propiedades de los cuerpos finitos para agregar redundancia a la información transmitida, lo que permite que en la decodificación se identifiquen y corrijan los errores.

3.3. Códigos skew Goppa

3.3.1. Códigos detectores y correctores de errores

Los códigos detectores y correctores de errores contrarrestar los posibles fallos que se pueden producir en la información durante su transmisión o almacenamiento. Una forma de hacer esto es añadiendo información necesaria a los datos que nos permita detectar y corregir estos fallos.

Los *códigos detectores* de errores tienen la capacidad de detectar cuando la información no es correcta, mientras que los *códigos correctores* de errores pueden reconstruir la información válida a partir de información con errores. Esto se logra gracias a que al codificar, se añaden unos *bits de redundancia o paridad* que indican si la información se ha dañado o no.

La *distancia de Hamming* es una métrica que hace referencia al números de bits en los que difieren dos palabras. Por ejemplo: $x = (0000), y = (0010)$, la distancia de Hamming entre x e y es 1. La distancia de Hamming de un código es la distancia mínima entre dos palabras que son válidas en dicho código. Si un código tiene una distancia de Hamming de dos, una alteración de 1 bit en una palabra produciría un error detectable. Véase [33].

Los *códigos lineales* son un tipo de códigos correctores de errores. Si $C \subseteq \mathbb{F}_q^n, C$ es un código lineal si C es un subespacio vectorial de $\mathbb{F}_q^n$, siendo $\mathbb{F}_q^n$ un $\mathbb{F}_q$-espacio vectorial de tamaño n. Si la dimensión del código linean C es k, podemos definir el código lineal como (n, k)-código. La distancia mínima del código se representa como d, y el peso mínimo (que indica el número mínimo de elementos no nulos), como w.

La *matriz generadora* de un código C es una matriz $G \in \mathcal{M}_{k\times n}(\mathbb{F}_q) = \mathbb{F}_q^{k\times n}$ cuyas filas forman una base del código C. A partir de la matriz generadora se pueden obtener todas las palabras de C. Véase [34].

La *matriz de control* o de paridad H es la matriz generadora del código dual de C: $C^{\perp}$. Como H es la matriz de control de C, podemos definir C a partir de ella de la siguiente manera: $C = z \in \mathbb{F}_q^n | zH^t = 0$. También $GH^t = 0$.

Ahora que ya sabemos qué son los códigos lineales y cómo se definen, vamos a ver cómo es el proceso de codificación y decodificación usando un código lineal:

- Codificación: para codificar debemos considerar un diccionario de palabras $(y_1...y_k) \in \mathbb{F}_q^n$, basta con multiplicar la palabra por la matriz generadora: $(y_1...y_k)G$, y esto se envía a través del canal.

- Decodificación: para decodificar una palabra $z \in \mathbb{F}_q^n$ existen 2 métodos que son completamente equivalentes, pues se use el que se use se hallará la palabra decodificada equivalente.

  El primer método es el método de decodificación basado en líderes. Dado un código lineal $C \subseteq \mathbb{F}_q^n$, definimos la siguiente relación de equivalencia:

  $$x_1 \cdots x_n \sim y_1 \cdots y_n \iff x_1 \cdots x_n - y_1 \cdots y_n \in C.$$

  Así, el conjunto de equivalencia de un vector $x_1 \cdots x_n$ está dado por:

  $$[x_1 \cdots x_n] = \{y_1 \cdots y_n \in \mathbb{F}_q^n \mid x_1 \cdots x_n - y_1 \cdots y_n \in C\}.$$

  Si recibimos la palabra $z_1 \cdots z_n \in \mathbb{F}_q^n$, calculamos $[z_1 \cdots z_n]$ y, asumiendo que el número de errores es mínimo según el principio de máxima verosimilitud, decodificamos $z_1 \cdots z_n$ como $z_1 \cdots z_n - y_1 \cdots y_n$, donde $y_1 \cdots y_n \in [z_1 \cdots z_n]$ es de peso mínimo. Si existe un único $y_1 \cdots y_n \in [z_1 \cdots z_n]$ con peso mínimo, lo llamaremos el líder de $[z_1 \cdots z_n]$, y este líder corresponde al error cometido en la transmisión. Si hay múltiples $y_1 \cdots y_n$ con el mismo peso mínimo, diremos que $z_1 \cdots z_n$ no admite decodificación única.

  El segundo método es el *método de decodificación mediante síndromes*. Sea $C \subseteq \mathbb{F}_q^n$ un código lineal con matriz de control H. Dado un vector $z_1 \cdots z_n \in \mathbb{F}_q^n$, el síndrome se define como:

  $$S(z_1 \cdots z_n) = (z_1 \cdots z_n)H^T.$$

  Las palabras del código C tienen síndrome cero. Definimos la relación de equivalencia:

  $$x_1 \cdots x_n \sim y_1 \cdots y_n \iff S(x_1 \cdots x_n) = S(y_1 \cdots y_n).$$

  Así, si $x_1 \cdots x_n \sim y_1 \cdots y_n$, entonces $S(x_1 \cdots x_n) = S(y_1 \cdots y_n)$, lo que implica que $x_1 \cdots x_n - y_1 \cdots y_n \in C$. Para decodificar una palabra $z_1 \cdots z_n \in \mathbb{F}_q^n$, determinamos $S(z_1 \cdots z_n)$ y decodificamos $z_1 \cdots z_n$ como $z_1 \cdots z_n - y_1 \cdots y_n$, donde $y_1 \cdots y_n \in \mathbb{F}_q^n$ es de peso mínimo tal que $S(z_1 \cdots z_n) = S(y_1 \cdots y_n)$. Véase [34].

3.3.2. Códigos skew Goppa

Los códigos skew Goppa son una generalización de los códigos Goppa, es decir, son un tipo de código lineal. Si se dan ciertos parámetros, el código skew Goppa es equivalente a uno normal. En [3], se define un código skew Goppa de la siguiente manera:

Sea $F \subseteq L$ una extensión de cuerpos. Sea $g \in R = L[x; \sigma, \partial]$ un polinomio invariante no nulo. Sean $\alpha_0, \ldots, \alpha_{n-1} \in L$ elementos distintos tales que $(x - \alpha_i, g)_r = 1$ para todo $0 \leq i \leq n-1$. Existen polinomios $h_i \in R$ tales que $\deg(h_i) < \deg(g)$ y

$$(x - \alpha_i)h_i - 1 \in Rg,$$

y sean $\eta_0, \ldots, \eta_{n-1} \in L^*$. Un código (generalizado) skew diferencial Goppa $C \subseteq \mathbb{F}^n$ es el conjunto de vectores $(c_0, \ldots, c_{n-1}) \in \mathbb{F}^n$ tales que

$$\sum_{i=0}^{n-1} h_i \eta_i c_i = 0.$$

Por un argumento de grado, esto es equivalente a

$$\sum_{i=0}^{n-1} h_i \eta_i c_i \in Rg.$$

El conjunto $\{\alpha_0, \ldots, \alpha_{n-1}\}$ está compuesto por los puntos de posición, g es el polinomio (skew diferencial) Goppa y $h_0, \ldots, h_{n-1}$ son los polinomios de verificación de paridad. Si $\partial = 0$, simplemente lo llamamos un código skew Goppa (generalizado). Véase [3].

3.4. Criptosistema de McEliece/Niederreiter

Ahora que ya sabemos qué son los códigos skew Goppa, en esta sección vamos a explicar qué son y cómo funcionan el criptosistema de McEliece y su variante de Niederreiter.

3.4.1. Criptosistema de McEliece

Como se afirma en [2], el *criptosistema de McEliece* fue propuesto por R.J. McEliece en 1978. Es un criptosistema de clave pública que se basa en la teoría de códigos algebraicos que está construido a partir de un código corrector de errores. Este criptosistema se basa en la dificultad de descodificar el código lineal aleatorio, así como en la complejidad de distinguir la estructura del mismo.

McEliece explicó en esta publicación [35] de 1978 que su criptosistema está basado en los códigos Goppa. Hoy en día solo es necesaria una familia de códigos que comparta longitud, dimensión y capacidad de corrección y dispongan de un algoritmo eficiente de decodificación. Si se tiene una familia de códigos con estas características comunes podemos definir un criptosistema del tipo McEliece. Para cada polinomio irreducible de grado t en $GF(2^m)$ existe un código binario irreducible de Goppa de tamaño $n = 2^m$ y dimensión $k \geq n - tm$ capaz de corregir un patrón de t o menos errores.

Suponemos que se escogen valores de t y n deseados, y se selecciona aleatoriamente un polinomio irreducible de grado t de $GF(2^m)$. Después se genera una matriz generadora G del código C de tamaño $k \times n$, que puede ser de en la forma escalonada reducida por filas. Véase [2, 35].

Después, se generan dos matrices más para el cálculo de la *clave pública* G':

- S: matriz no singular (su determinante es distinto de 0 y por tanto tiene inversa) aleatoria de tamaño $k \times k$.
- P: matriz de permutación (en cada fila y columna hay un único 1, siendo los demás elementos 0) aleatoria de tamaño $n \times n$.

Con esto se calcula $G' = SGP$, que genera un código linear con los mismos parámetros y distancia mínima que G.

Algorithm 6: KeysMcEliece

Input : El código $\mathcal{C}$ sobre $\mathbb{F}_q$ de tipo $[n, k]$.
Output: El par de claves pública y privada (κ, κ') respectivamente.

1 Se elige al azar una matriz generadora de $\mathcal{C}$, G
2 Se eligen al azar una matriz $k \times k$ regular, S, y una matriz de permutación $n \times n$, P
3 Se calcula $G' = SGP$
4 $\kappa = G'$
5 $\kappa' = (S, G, P)$
6 **return** (κ, κ')

Figura 3.3: Generación de claves de McEliece, véase [2].

Una vez calculadas las claves, se procede al cifrado y descifrado.

- Algoritmo de cifrado: se divide la información que se desea cifrar en bloques de k bits. Si m es uno de estos bloques, se transmite el vector $x = mG' + e$, donde e es un vector aleatorio de tamaño n y t elementos no nulos. La figura 3.4 contiene la descripción del algoritmo de cifrado:

Algorithm 7: CifMcEliece

Input : El mensaje en texto plano $\mathbf{m} \in \mathbb{F}_q^k$ y la clave pública $\kappa = G'$.
Output: El mensaje cifrado $\mathbf{c} \in \mathbb{F}_q^n$.

1 Se elige al azar $\mathbf{e} \in \mathbb{F}_q^n$ con peso menor o igual que t
2 Se calcula $\mathbf{c} = \mathbf{m}G' + \mathbf{e}$
3 **return** $\mathbf{c}$

Figura 3.4: Algoritmo de cifrado de McEliece, véase [2].

- Algoritmo de descifrado: se calcula $x' = xP^{-1}$, donde P^{-1} es es la matriz inversa de P y x' está a distancia t de una palabra del código generado anteriormente. Usando el algoritmo de decodificación se consigue $c = mSG$. Como G es de rango máximo, es posible, resolviendo el correspondiente sistema lineal, calcular $c' = mS$), de donde obtenemos $m = c'S^{-1}$. La figura 3.5 contiene la descripción del algoritmo de descifrado:

Algorithm 8: DescifMcEliece

Input : El mensaje cifrado $\mathbf{c} \in \mathbb{F}_q^n$ y la clave privada $\kappa' = (S, G, P)$.
Output: El mensaje original $\mathbf{m} \in \mathbb{F}_q^k$.

1 Se calcula $\mathbf{c}_1 = \mathbf{c}P^{-1}$
2 Se calcula $\mathbf{c}_2 = D_C(\mathbf{c}_1) = \mathbf{m}SG$
3 Se calcula $\mathbf{c}_3 = \mathbf{m}S$, resolviendo el sistema de ecuaciones $\mathbf{c}_2 = \mathbf{c}_3 G$
4 Se calcula $\mathbf{c}_4 = \mathbf{c}_3 S^{-1}$
5 **return** $\mathbf{c}_4$

Figura 3.5: Algoritmo de descifrado de McEliece, véase [2].

McEliece, en [35], afirma que este sistema es seguro, ya que sería capaz de resistir ataques que incluyan el intento de recuperación de G a partir de G' (ya que si n y t son lo suficientemente grandes, hay demasiados posibles G) o el intentar averiguar u a partir de x sin usar G (ya que si los parámetros son lo suficientemente grandes, las posibilidades de encontrar los errores introducidos son prácticamente nulas. También asegura que la implementación de estos algoritmos es sencilla usando lógica digital y que tasas de comunicación de 10^6 bits por segundo podrían llevarse a cabo sin problema.

Como hemos visto en esta breve explicación, McEliece implementó el criptosistema basado en códigos Goppa, generando una clase de códigos C, que junto a unos algoritmo eficientes de codificación y decodificación permite camuflar el código C para que parezca aleatorio. Véase, [2].

3.4.2. Criptosistema de Niederreiter

En 1986, Harald Niederreiter desarrolló una variación del criptosistema de McEliece. Esta variación está enfocada a ser utilizada como criptosistema híbrido, y utiliza la matriz de control de paridad en lugar de la matriz generadora. La seguridad del criptosistema aún no se ha visto comprometida, de

igual forma que el criptosistema de McEliece, ya que ambos son equivalentes en términos de seguridad. Véase, [36].

El proceso para *construir el criptosistema de Niederreiter* es bastante similar al de McEliece:

- Disponemos de una familia de códigos de longitud n, dimensión k y capacidad de corrección t construidos sobre un cuerpo finito $\mathbb{F}_q$ que disponen de un algoritmo eficiente de decodificación.
- Primero, se calcula la matriz $H \in \mathbb{F}_q^{(n-k)\times n}$ de control de paridad de un código C de la familia mencionada en el punto anterior.
- Al igual que en el criptosistema de McEliece, se generan aleatoriamente matrices cuadradas $S \in \mathbb{F}_q^{(n-k)\times(n-k)}$ y $P \in \mathbb{F}_q^{n\times n}$, siendo esta última una matriz de permutación.
- Por último, se calcula la matriz $H' = SHP$. La clave pública es H' y la clave privada es (H, S, P). Véase [36].

Esta versión está enfocada al encapsulamiento de clave. El cifrado (encapsulado) y descifrado (desencapsulado) en este criptosistema ocurren de la siguiente manera:

- Cifrado: se genera de forma aleatoria $e \in \mathbb{F}_q^n$, un vector de peso t. El cifrado (encapsulado) es $c = H' \cdot e$.
- Descifrado: se utiliza el algoritmo de decodificación del código C para calcular la única solución e' del sistema lineal $S^{-1}c = Hy$ cuyo peso es igual a t. Una vez calculado e', tenemos que $e = P^{-1} \cdot e'$. Véase [36].

Finalizado el proceso, emisor y receptor conocen, y solo ellos, el vector e, a partir del cual puede derivarse una clave de sesión para un criptosistema simétrico.

En resumen, el criptosistema de Niederreiter, al igual que el de McEliece, se basa en la teoría de códigos y la dificultad de descifrar códigos aleatorios. Ambos sistemas se mantienen seguros debido a la dificultad del problema de decodificación de códigos lineales.

3.5. Inclusión de códigos skew Goppa en el criptosistema McEliece/Niederreiter

Ahora que ya conocemos los códigos skew Goppa y el criptosistema de McEliece/Niederreiter, en esta sección se va a explicar cómo transformar el criptosistema para que trabaje con polinomios skew Goppa. Esta sección se ha obtenido de [3].

3.5.1. Parámetros del criptosistema

El primer paso para construir nuestro criptosistema es seleccionar algunos *parámetros* para el diseño de un código skew Goppa C con el que trabajará nuestro criptosistema:

- Alfabeto: en este caso, el alfabeto es un cuerpo finito $F = \mathbb{F}_q$, donde $q = p^d$ para un primo p.
- n, t: el valor de n es la longitud del código y t la capacidad de corrección $(t < n/2)$. En la práctica, t es bastante menor que n. Para que la codificación sea correcta, se debe garantizar:

$$t = \left\lfloor \frac{\deg g}{2} \right\rfloor \tag{3.1}$$

  donde g es el polinomio skew Goppa.
- L: L es la extensión de grado m del cuerpo finito F, por tanto $L = \mathbb{F}_{q^m}$, donde $q = p^d$ y por tanto q^m es p^{dm}.
- R: tenemos que crear el anillo de polinomios skew $R = L[x; \sigma]$, donde L es el cuerpo de extensión de F de grado m $(L = \mathbb{F}_{q^m})$.
- Dado n, t, $q = p^d$, queremos encontrar m y δ tales que:

$$\text{máx}\left\{ \frac{n}{10t}, \frac{n\delta}{d(p^\delta - 1)} \right\} \leq m \leq \frac{n}{4t} \quad \text{y} \quad \delta \mid dm \tag{3.2}$$

  Estos parámetros podemos encontrarlos generando todas las parejas de m y δ posibles y eligiendo una aleatoriamente.
- γ: γ es un elemento primitivo.
- μ: $\mu = \frac{dm}{\delta}$.
- k: $k = n - 2t \left\lfloor \frac{n}{4t} \right\rfloor$.

Esta proposición del criptosistema de McEliece sigue la versión dual de Niederreiter gracias al mecanismo de encapsulación de claves propuesto en.

3.5.2. Generación de la clave pública

Para generar la clave pública necesitamos una serie de puntos de posición o evaluación que están generados a partir de un *elemento primitivo γ de L y un elemento normal α del cuerpo L*. Un elemento normal de un cuerpo es aquel que constituye una base normal del cuerpo.

Un elemento $\alpha \in L = \mathbb{F}_p^{dm}$, $\left\{\alpha, \alpha^{p^\delta}, ..., \alpha^{p^{(\mu-1)\delta}}\right\}$ constituye una base normal de L si y solo si:

$$\text{mcd}\left(z^\mu - 1, \alpha z^{\mu-1} + \alpha^{p^\delta} z^{\mu-2} + \cdots + \alpha^{p^{(\mu-1)\delta}}\right) = 1 \tag{3.3}$$

Una vez que se ha calculado un elemento primitivo γ y un elemento normal α, un conjunto máximo de elementos P-independientes por la izquierda es

$$P = \left\{ \gamma^i \frac{\sigma^{j+1}(\alpha)}{\sigma^j(\alpha)} \;\middle|\; 0 \le i \le p^\delta - 2,\ 0 \le j \le \mu - 1 \right\} \tag{3.4}$$

En el caso clásico en el que $\delta = dm$, los elementos del conjunto P-independiente significan solamente puntos posicionales diferentes. Por ello, la *lista de puntos posicionales* $E = \{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$ se puede obtener seleccionando n elementos aleatorios de P. Necesitamos generar una última lista de elementos $\eta = \eta_0, \eta_1, ..., \eta_{n-1}$, donde η_i es un elemento no nulo del cuerpo L.

Para construir el *polinomio skew Goppa* $g(x)$, cogemos un polinomio mónico $h(y) \in K$ sin raíces en K, tal que $\deg_y(h) = \lfloor 2t/mu \rfloor$ y definimos $g = h(x^\mu)x^{(2t)} \mod \mu$, el cual tiene grado 2t, siendo K el subcuerpo de L de orden p^δ.

Obtenido el polinomio skew Goppa $g(x)$, podemos usar el algoritmo extendido de Euclides por la derecha para computar los *polinomios no conmutativos de paridad* $h_0, ..., h_{n-1} \in R$ tales que para cada $0 \le i \le n-1$, $grado(h_i) < 2t$ y

$$(x - \alpha_i)h_i - 1 \in Rg \tag{3.5}$$

Ahora, ya podemos construir nuestra clave pública. Para ello primero necesitamos generar la matriz de paridad H. La matriz de paridad para el código skew Goppa es:

$$H = \left(v(\sigma^{-j}(h_{i,j})\eta_i)\right)_{0 \le j \le 2t-1,\ 0 \le i \le n-1} \in F^{(2tm)\times n}$$

donde

$$h_i = \sum_{j=0}^{2t-1} h_{i,j} x^j.$$

Una vez que se ha calculado H, la clave pública H_{pub} de nuestro criptosistema se puede calcular de la siguiente manera:

- Se define $k = n - 2t\left\lfloor \frac{n}{4t} \right\rfloor$, $r_H = \text{rango}(H)$ y $A \in F^{(n-k-r_H)\times n}$, una matriz aleatoria de rango completo.

- La *matriz* H_{pub} se forma con las filas no nulas de la forma escalonada reducida por filas de la matriz bloque $(\frac{H}{A})$. Si H_{pub} tiene menos de $n-k$ filas, se debe elegir una nueva matriz A. Esta H_{pub} define un subcódigo lineal de C de dimensión k.

Después de este generar las claves, los diferentes valores quedan como sigue:

- **Parámetros**: t, n, $q = p^d$ y $k = n - 2t\left\lfloor \frac{n}{4t} \right\rfloor$.
- **Clave pública**: $H_{\text{pub}} \in F^{(n-k)\times n}$.
- **Clave privada**: L, σ, $E = \{\alpha_0, \ldots, \alpha_{n-1}\}$, $\eta_0, \ldots, \eta_{n-1}$, g y los polinomios de paridad $h_0, \ldots, h_{n-1}$.

3.5.3. Encapsulado y desencapsulado

El proceso de *encapsulado* es el siguiente:

- Se elige un vector de errores aleatorio $e \in F^n$ que contenga t elementos no nulos, con el polinomio de errores correspondiente $e(x) = \sum_{j=1}^{t} e_j x^{k_j}$, donde $0 \leq k_1 < k_2 < \cdots < k_t \leq n-1$.
- El emisor puede derivar fácilmente una clave de sesión compartida a partir de e mediante una función hash fija y de conocimiento público $\mathcal{H}$. El criptograma es

$$c = eH_{\text{pub}}^T \in F^{n-k} \tag{3.6}$$

Para *desencapsular* el criptograma el emisor puede derivar fácilmente una clave secreta compartida a partir de e mediante una función hash fija y de conocimiento público H. El criptograma es

$$c = eH_{\text{pub}}^T \in F^{n-k} \tag{3.7}$$

Para descifrar, el receptor puede calcular fácilmente $y \in F^n$ tal que

$$c = yH_{\text{pub}}^T, \tag{3.8}$$

ya que H_{pub} está en forma escalonada reducida por filas. A y se le pueden aplicar los siguientes algoritmos para obtener e:

- **Algoritmo de decodificación para códigos skew Goppa diferenciales con fallos de decodificación improbables**:

Algorithm 2 Decoding algorithm for skew differential Goppa codes with unlikely decoding failure

Require: A skew differential Goppa code $\mathcal{C}$ of length n, correction capability t, position points $\{\alpha_0, \ldots, \alpha_{n-1}\} \subseteq L$, $\eta_0, \ldots, \eta_{n-1} \in L^*$, skew differential Goppa invariant polynomial $g \in L[x;\sigma,\partial]$ with $\deg(g) = 2t$, and parity check polynomials $h_0, \ldots, h_{n-1} \in L[x;\sigma,\partial]$ of degree $2t-1$.
Require: A received word $y = (y_0, \ldots, y_{n-1}) \in F^n$.
Ensure: A vector $e \in F^n$ such that $\mathrm{w}(e) \leq t$ and $y - e \in \mathcal{C}$, or *decoding failure*.
$s \leftarrow \sum_{i=0}^{n-1} h_i \eta_i y_i$
if $s = 0$ **then**
  return the zero vector
end if
$r_{prev} \leftarrow g, r_{curr} \leftarrow s, v_{prev} \leftarrow 0, v_{curr} \leftarrow 1$ ▷ LEEA
while $\deg(r_{curr}) \geq t$ **do**
  $f, r \leftarrow$ l-quo-rem(r_{prev}, r_{curr})
  $v \leftarrow v_{prev} - f v_{curr}, v_{prev} \leftarrow v_{curr}, r_{prev} \leftarrow r_{curr}, v_{curr} \leftarrow v, r_{curr} \leftarrow r$
end while
$pos \leftarrow \{\}, other = \{0, \ldots, n-1\}$ ▷ Finding error positions
for $0 \leq i \leq n-1$ **do**
  if α_i is a right root of v_{curr} **then**
    $pos \leftarrow pos \cup \{i\}, other = other \setminus \{i\}$
  end if
end for
if $\deg(v_{curr}) >| \ pos \ |$ **then**
  'Decoding failure'
  stop
end if
for $j \in pos$ **do** ▷ Finding error values
  $\rho_j \leftarrow$ l-quo$(v_{curr}, x - \alpha_j)$
end for
Solve the linear system $r_{curr} = \sum_{j \in pos} \rho_j \eta_j e_j$
$e(x) \leftarrow \sum_{j \in pos} e_j x^j$
return the vector associated to the polynomial $e(x)$

Figura 3.6: Algoritmo de decodificación para códigos skew Goppa diferenciales con fallos de decodificación improbables, véase [3].

- **Algoritmo de resolución de fallos de decodificación**:

Algorithm 3 Solving decoding failures

Require: A skew differential Goppa code $\mathcal{C}$ of length n, correction capability t, position points $\{\alpha_0, \ldots, \alpha_{n-1}\} \subseteq L$, $\eta_0, \ldots, \eta_{n-1} \in L^*$, skew differential Goppa invariant polynomial $g \in R$ with $\deg(g) = 2t$, and parity check polynomials $h_0, \ldots, h_{n-1} \in R$ of degree $2t-1$.
Require: The polynomials v_{curr}, r_{curr} and the sets $pos, other$ in Algorithm 2
Ensure: The locator polynomial λ.
while $\deg(v_{curr}) >| \ pos \ |$ **do**
  $f \leftarrow v_{curr}, e \leftarrow \deg(f)$
  $i \leftarrow$ one element in $other, other = other \setminus \{i\}$
  $f \leftarrow \left[f, x - \alpha_i\right]_\ell$
while $\deg(f) > e$ **do**
  $e \leftarrow e + 1$
  $i \leftarrow$ one element in $other, other = other \setminus \{i\}$
  $f \leftarrow \left[f, x - \alpha_i\right]_\ell$
end while
  $pos = pos \cup \{i\}, other = \{0, \ldots, n-1\} \setminus pos$
  $v \leftarrow v_{curr}, v_{curr} \leftarrow \left[v, x - \alpha_i\right]_\ell, h \leftarrow$ l-quo$(v_{curr}, v), r_{curr} \leftarrow h r_{curr}$
for $i \in other$ **do**
  if α_i is a right root of v_{curr} **then**
    $pos \leftarrow pos \cup \{i\}, other = other \setminus \{i\}$
  end if
end for
end while
return v_{curr}, r_{curr}, pos

Figura 3.7: Algoritmo de resolución de fallos de decodificación, véase [3].

Luego, la clave secreta compartida puede ser recuperada por el receptor como $\mathcal{H}(e)$.

Si desea hacer una consulta más profunda o ver algún ejemplo, no dude en mirar [3].

Capítulo 4

Diseño e Implementación

En este capítulo vamos a ver cómo se ha diseñado el sistema que implementa el criptosistema de McEliece/Niederreiter con códigos skew Goppa y su posterior implementación.

4.1. Diseño de la solución

El proceso de diseño del sistema que vamos a implementar es fundamental para el desarrollo del proyecto, pues establece las bases para la posterior implementación. Durante este proceso de diseño se ha generado un listado de requisitos que debe cumplir el sistema, y a partir de él se ha decidido la arquitectura del sistema, así como las clases y operaciones necesarias y cómo implementarlas.

Listado de requisitos del sistema

El siguiente *listado de requisitos del sistema* se ha obtenido a partir de reuniones con mi tutor, así como del documento de patente de la propuesta de implementación generado por mi tutor junto con su grupo de trabajo, véase [4],

1. **Requisitos funcionales**:
    - El sistema debe generar una clave pública y una clave privada basándose en el criptosistema de McEliece/Niederreiter
    - Para la construcción de la clave el sistema debe usar un código skew Goppa, en lugar del código Goppa que utiliza el criptosistema de McEliece/Niederreiter convencional.
    - La clave pública debe contener todos los elementos necesarios para el encapsulado de información.

- La clave privada debe contener todos los elementos necesarios para el desencapsulado de criptogramas.
- El programa debe simular el envío y recepción de claves y criptogramas.
- El sistema debe poder encapsular un secreto compartido usando la clave pública
- El encapsulado debe de funcionar de acuerdo a lo establecido en la propuesta de implementación, véase [4].
- El sistema debe poder desencapsular un criptograma usando la clave privada.
- El desencapsulado debe de funcionar de acuerdo a lo establecido en la propuesta de implementación, véase [4].
- Se debe poder comprobar el correcto funcionamiento de los algoritmos de generación de claves, encapsulado y desencapsulado del sistema.
- El usuario debe poder interactuar con el sistema para generar una clave nueva, encapsular, desencapsular o comprobar la validez de los pasos anteriores.
- El sistema debe poder producir mensajes de error claros entendibles para un usuario no experto.

2. **Requisitos no funcionales**:

- El sistema debe ser eficiente en términos de tiempo de ejecución, para poder ser usado en aplicaciones prácticas.
- El sistema debe poder manejar diferentes tamaños de clave y de mensaje.
- El sistema debe ser fácil de usar para un usuario no experto.
- El sistema debe tener una documentación, tanto el código, como tener un manual de usuario que permita su uso a usuarios no expertos.

Arquitectura del sistema

Una vez obtenida esta lista de requisitos, podemos definir qué arquitectura queremos que tenga nuestra sistema. En este caso, la *arquitectura de nuestro sistema* está compuesta por 4 programas de python que interactúan entre ellos mediante archivos que guardan la información, simulando así el envío de la clave o del criptograma, ya que un programa guarda la información en un archivo, y el siguiente la recoge de él como si le hubiera llegado por medio de un envío:

- generar_clave.py: este programa se encarga de generar y guardar la clave pública y la clave privada, así como de los parámetros necesarios de cada una. Guardará la clave pública en un archivo llamado **Clave_pub.npz** y la privada en un archivo llamado **Clave.npz**.

- encapsular.py: este programa se encarga de recoger la clave pública del archivo Clave_pub.npz y generar un secreto compartido aleatorio que se quiere enviar. Utiliza la clave pública para encapsular el secreto compartido y generar el criptograma. Además, genera un hash del criptograma que nos sirve para después verificar que el sistema funciona correctamente. Por último, guarda el criptograma que se va a enviar al receptor en el archivo **criptograma.npy** y el hash en el archivo **hash_encapsulado.npy**.

- desencapsular.npy: este programa es el encargado de recoger el criptograma del archivo criptograma.npy y la clave privada del archivo Clave.npz. Con esto, desencapsulará el criptograma usando la clave privada, generando el vector de salida del criptosistema. Una vez obtenido este vector de salida, genera un hash igual que en encapsular.py, y lo guarda en el archivo llamado **hash_desencapsulado.npy**

- comparar_hashes.py: este último programa es el más sencillo de todos, pues obtiene los dos hashes generados anteriormente de sus correspondientes archivos y los compara. Si el hash generado durante el encapsulado y el hash generado durante el desencapsulado son iguales, quiere decir que el secreto compartido y el vector de salida son iguales, por lo que el criptosistema funciona correctamente.

Para ilustrar de mejor manera la arquitectura de nuestro sistema, este diagrama realizado con la herramienta online Visual Paradigm [37] muestra los diferentes componentes del sistema y cómo interactúan entre ellos:

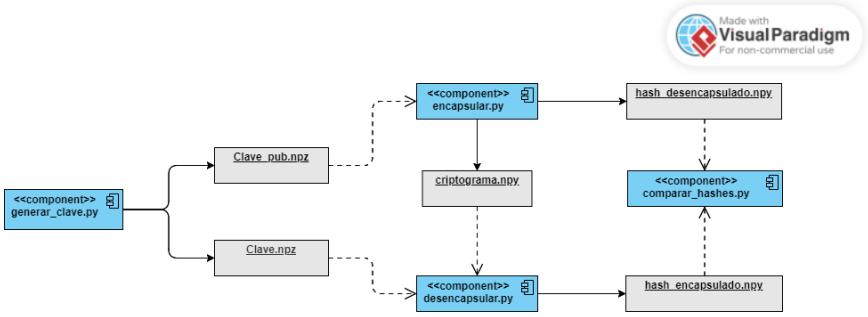

Figura 4.1: Diagrama de componentes de nuestro sistema.

*Nótese que en el diagrama 4.1, las flechas continuas indican que un programa genera ese archivo y las flechas discontinuas indican que el programa recoge datos de un archivo.

Ya que la arquitectura de nuestro sistema está bien definida y se ha explicado cómo funcionará el criptosistema, podemos representar con más profundidad los elementos que formarán parte del sistema. Para ello se ha realizado el siguiente diagrama de clases con la herramienta online Visual Paradigm [37], el cual muestra los elementos y las funciones que están presentes en cada componente del sistema:

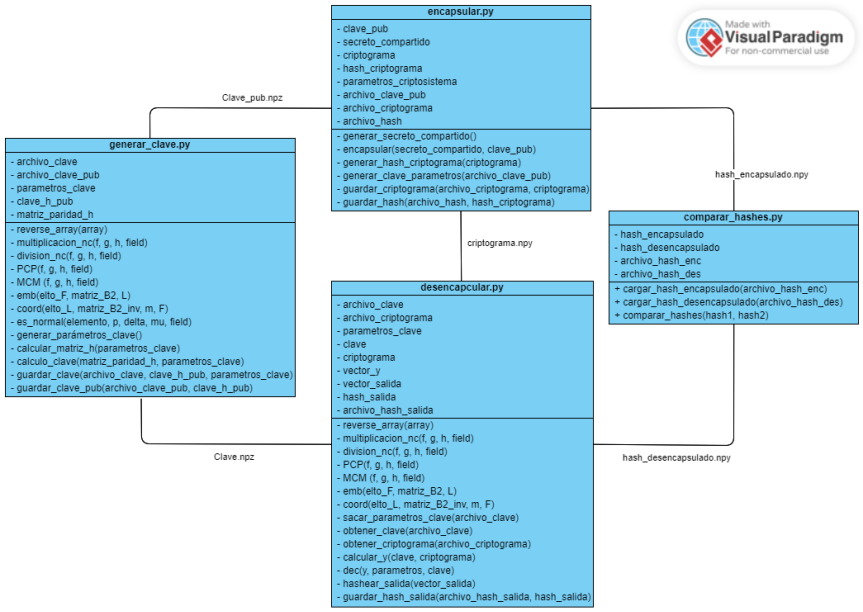

Figura 4.2: Diagrama de clases del sistema.

Para finalizar el diseño de nuestro sistema, es crucial definir un *criterio de validación* que nos permita verificar su correcto funcionamiento. En este caso, el sistema se considera que el sistema funciona correctamente si, al finalizar el proceso completo, el hash generado por el secreto compartido durante el proceso de encapsulación coincide con el hash generado a partir del vector de salida al desencapsular.

Este criterio asegura que, al utilizar la misma clave tanto en el proceso de encapsulación como en el de desencapsulación, se obtenga exactamente la misma información que se pretendía enviar de forma segura. En otras palabras, si los hashes son iguales, se puede concluir que el sistema ha funcionado correctamente, garantizando que la información encapsulada y desencapsulada coincide, lo que significa que el mensaje original se ha transmitido y recuperado sin alteraciones.

4.2. Implementación de la solución

Cuando ya se ha diseñado nuestro sistema, podemos proceder a la implementación. La implementación de nuestro sistema se ha ceñido a su diseño, además de seguir las directrices que se encuentran en la propuesta de implementación, véase [4]

A continuación, procederemos a explicar el *código* generado durante la implementación:

4.2.1. Funciones comunes del sistema

En nuestro sistema podemos encontrar algunas funciones, las cuales se utilizan tanto para la generación de las claves como para el desencapsulado del criptograma. Entre ellas podemos encontrar:

- reverse_array(array): esta función recibe una lista como parámetro y devuelve una lista con el orden inverso a la original. Nos sirve para interactuar de manera correcta con la librería galois y obtener los coeficientes de los polinomios en el orden correcto.

```
1  def reverse_array(arr):
2      """
3      Invierte el orden de los elementos de una lista.
4
5      Parámetros:
6      arr (list): La lista de elementos a invertir.
7
8      Retorna:
9      list: Una nueva lista con los elementos en orden
           inverso.
10     """
11     return arr[::-1]
```

- multiplicacion_nc(f, g, h, field): está función implementa la multiplicación no conmutativa de dos polinomios f y g y devuelve el producto, según se describe en el siguiente diagrama de flujo extraído de la propuesta de implementación:

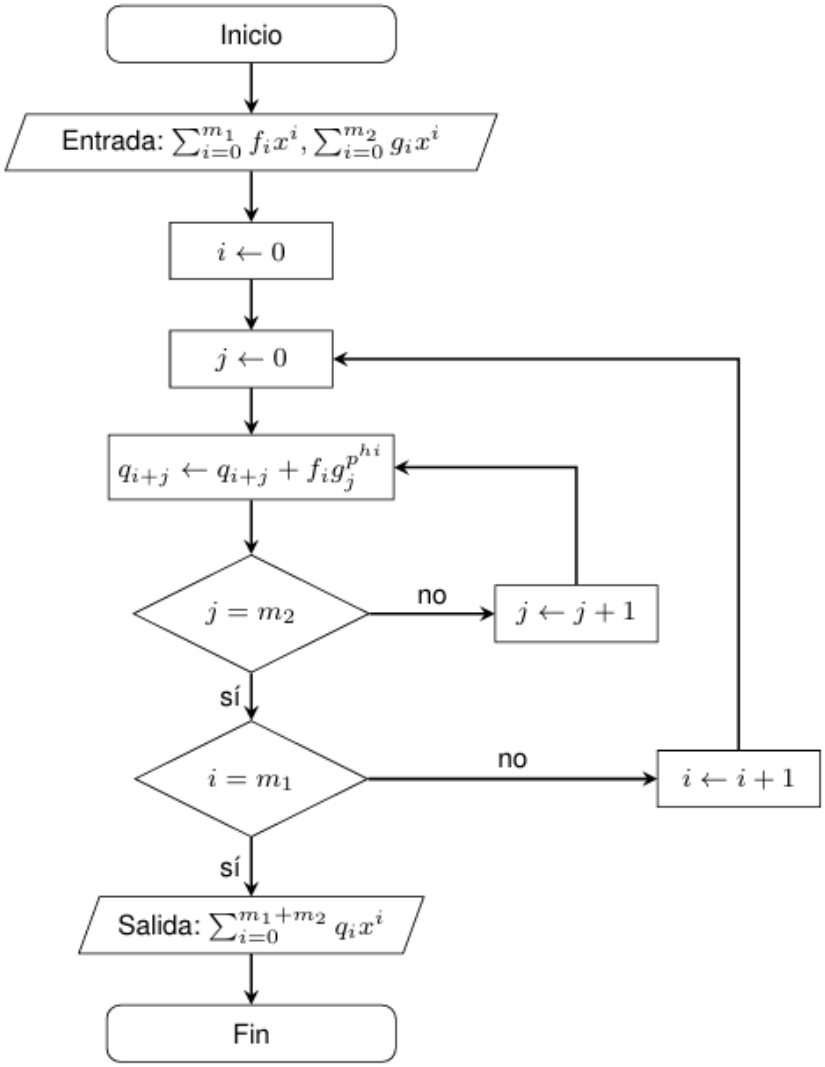

Figura 4.3: Algoritmo de multiplicación no conmutativa, véase [4].

```
def multiplicacion_nc(f, g, h, field):
    """
    Realiza la multiplicación no conmutativa de dos
       polinomios f y g en un cuerpo dado.

    Parámetros:
```

```
    f (galois.Poly): El primer polinomio.
    g (galois.Poly): El segundo polinomio.
    h (int): Parámetro del criptosistema.
    field (galois.Field): El cuerpo sobre el cual se realiza la operación.

    Retorna:
    galois.Poly: El polinomio resultado de la multiplicación no conmutativa.
    """
    p = field.characteristic  # Característica del cuerpo

    coef_f2 = reverse_array(f.coefficients())  # Coeficientes de f en orden inverso
    f2 = [coeficiente for coeficiente in coef_f2]

    coef_g2 = reverse_array(g.coefficients())  # Coeficientes de g en orden inverso
    g2 = [coeficiente for coeficiente in coef_g2]

    tam = len(f2) + len(g2) - 1  # Tamaño del polinomio resultado
    resultado = [field(0)] * tam  # Inicialización del resultado con ceros

    for i in range(len(f2)):
        exponente = p ** (h * i)  # Calcula el exponente para la operación no conmutativa
        for j in range(len(g2)):
            resultado[i + j] = resultado[i + j] + (f2[i] * (g2[j] ** exponente)) # Calcula los coeficientes del resultado (de menos a mayor)

    resultado = galois.Poly(reverse_array(resultado), field=field)  # Invierte el orden de los coeficientes (se guardan de mayor a menor en un polinomio)

    return resultado
```

- division_nc(f, g, h, field): está función implementa la división no conmutativa de dos polinomios f y g y devuelve el cociente y el resto de la operación, según se describe en el siguiente diagrama de flujo extraído de la propuesta de implementación:

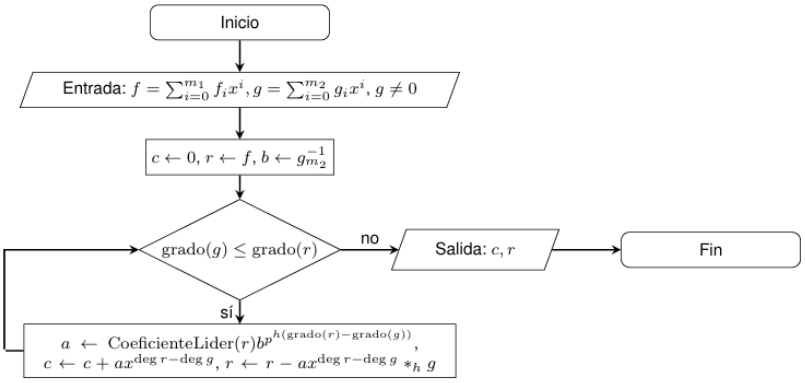

Figura 4.4: Algoritmo de división no conmutativa, véase en [4]

```python
def division_nc(f, g, h, field):
    """
    Realiza la división no conmutativa de dos polinomios f
        y g en un cuerpo dado.

    Parámetros:
    f (galois.Poly): El dividendo.
    g (galois.Poly): El divisor.
    h (int): Parámetro del criptosistema.
    field (galois.Field): El cuerpo sobre el cual se
        realiza la operación.

    Retorna:
    tuple: El cociente y el residuo de la división no
        conmutativa.
    """
    p = field.characteristic  # Característica del cuerpo

    if g == galois.Poly([0], field=field): #Si el divisor
        es 0 salimos de la operación
        return None

    # Parámetros iniciales
    c = galois.Poly([0], field=field)  # Cociente
        inicializado en cero
    r = f  # Residuo inicializado como el dividendo
    coefs_g = [coeficiente for coeficiente in g.
        coefficients()]  # Coeficientes de g (de mayor a
        menor)
    b = (coefs_g[0] ** (-1))  # Inverso del primer
```

```
        coeficiente de g
24
25      while g.degree <= r.degree and r != galois.Poly([0],
            field=field):
26          coefs_r = [coeficiente for coeficiente in r.
                coefficients()]  # Coeficientes de r (de mayor
                a menor)
27          expB = p ** (h * (r.degree - g.degree))  #
                Exponente para la operación
28          a = coefs_r[0] * (pow(b, expB))  # Coeficiente
                principal para el monomio
29
30          # Construcción del monomio correspondiente
31          vector_monomio = [a]
32          for i in range(r.degree - g.degree):
33              vector_monomio.append(0)
34
35          monomio = galois.Poly(vector_monomio, field=field)
                # Creación del monomio
36          c = c + monomio  # Actualización del cociente
37          r = r - multiplicacion_nc(monomio, g, h, field)  #
                Actualización del residuo
38
39      return c, r
```

- PCP(f, g, h, field): está función ejecuta el algoritmo extendido de Euclides no conmutativo a izquierda de dos polinomios f y g, y devuelve el coeficiente lineal correspondiente a f si el máximo común divisor de f y g a derecha es 1, y cero en otro caso, según se describe en el siguiente diagrama de flujo extraído de la propuesta de implementación:

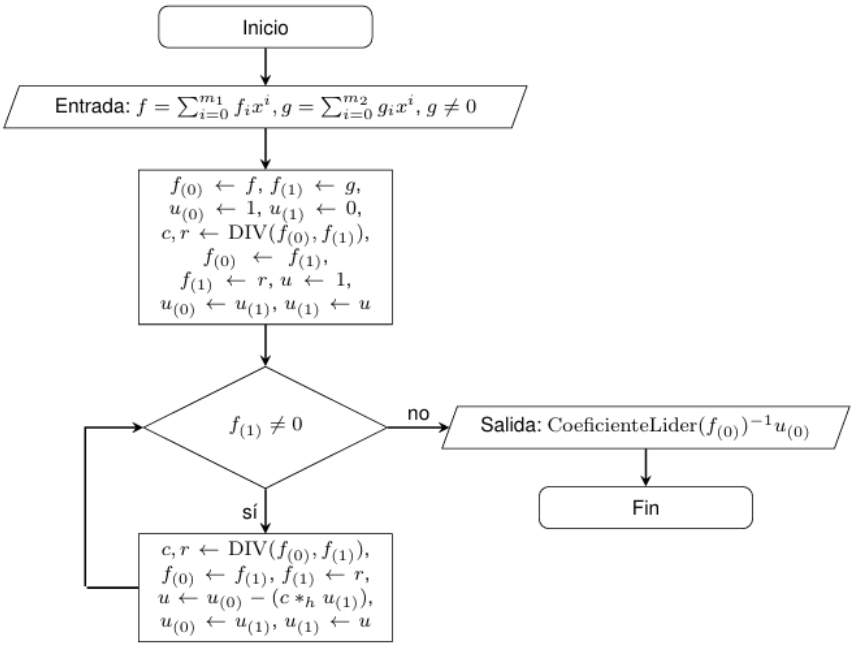

Figura 4.5: Algoritmo PCP, véase en [4]

```
1  def PCP(f, g, h, field):
2      """
3      Calcula el máximo común divisor a la derecha de dos
           polinomios f y g utilizando el algoritmo de
           Euclides extendido
4      no conmutativo para quedarnos con el coeficiente de
           Bezout correspondiente al primer polinomio.
5
6      Parámetros:
7      f (galois.Poly): El primer polinomio.
8      g (galois.Poly): El segundo polinomio.
9      h (int): Parámetro del criptosistema.
10     field (galois.Field): El cuerpo sobre el cual se
           realiza la operación.
11
12     Retorna:
13     galois.Poly: El máximo común divisor de f y g, o cero
           si no existe.
```

```
    """
    if g == field(0): #Si el segundo polinomio es 0 salimos de la función
        return None

    # Inicialización de variables
    f0 = f
    f1 = g

    u0 = galois.Poly.One(field)  # Polinomio unidad
    u1 = galois.Poly.Zero(field)  # Polinomio cero

    # Primer paso del algoritmo
    cociente, resto = division_nc(f0, f1, h, field)
    f0 = f1
    f1 = resto
    u = galois.Poly.One(field)
    u0 = u1
    u1 = u

    # Iteración hasta que el residuo sea cero
    while f1 != galois.Poly.Zero(field):
        cociente, resto = division_nc(f0, f1, h, field)
        f0 = f1
        f1 = resto
        u = u0 - multiplicacion_nc(cociente, u1, h, field)
        u0 = u1
        u1 = u

    coefs_f0 = [coeficiente for coeficiente in f0.coefficients()]  # Coeficientes de f0

    # Si el grado es uno, retorna el inverso del primer coeficiente multiplicado por u0, si no, devuelve 0
    if len(coefs_f0) == 1:
        return (coefs_f0[0] ** (-1) * u0)
    else:
        return galois.Poly.Zero(field)
```

- MCM(f, g, h, field): está función calcula el mínimo común múltiplo (mcm) no conmutativo a izquierda de dos polinomios f y g, según se describe en el siguiente diagrama de flujo extraído de la propuesta de implementación:

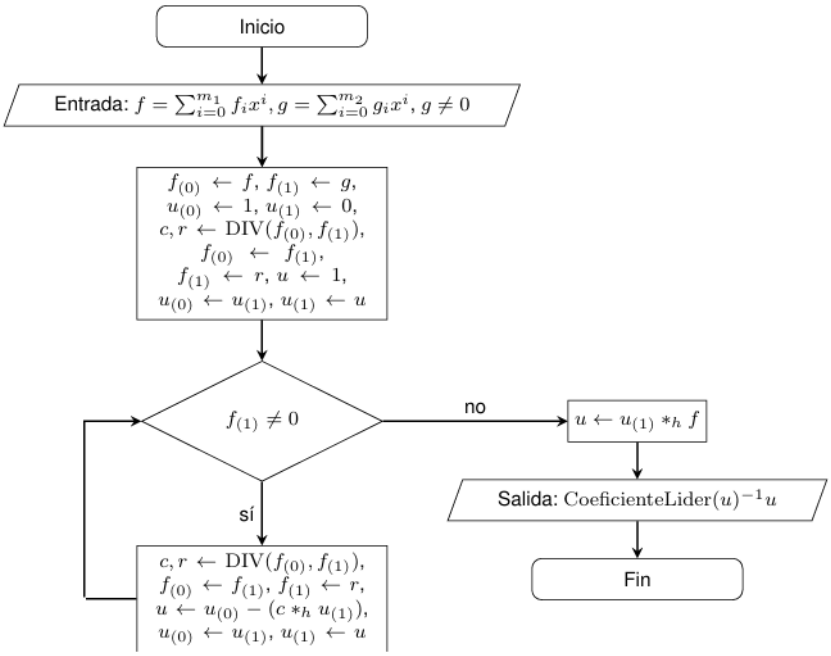

Figura 4.6: Algoritmo MCM no conmutativo, véase en [4].

```
1  def MCM(f, g, h, field):
2      """
3      Calcula el mínimo común múltiplo de dos polinomios f y
           g utilizando un algoritmo no conmutativo.
4
5      Parámetros:
6      f (galois.Poly): El primer polinomio.
7      g (galois.Poly): El segundo polinomio.
8      h (int): Parámetro del criptosistema.
9      field (galois.Field): El cuerpo sobre el cual se
          realiza la operación.
10
11     Retorna:
12     galois.Poly: El mínimo común múltiplo de f y g.
13     """
14     if g == galois.Poly([0], field=field): #Si el segundo
          polinomio es 0 devolvemos el valor 0
15         return galois.Poly([0], field=field)
16
17     # Inicialización de variables
18     f0 = f
```

```
19      f1 = g
20
21      u0 = galois.Poly([1], field=field)  # Polinomio unidad
22      u1 = galois.Poly([0], field=field)  # Polinomio cero
23
24      # Primer paso del algoritmo
25      cociente, resto = division_nc(f0, f1, h, field)
26      f0 = f1
27      f1 = resto
28      u = galois.Poly([1], field=field)
29      u0 = u1
30      u1 = u
31
32      # Iteración hasta que el residuo sea cero
33      while f1 != galois.Poly([0], field=field):
34          cociente, resto = division_nc(f0, f1, h, field)
35          f0 = f1
36          f1 = resto
37          u = u0 - multiplicacion_nc(cociente, u1, h, field)
38          u0 = u1
39          u1 = u
40
41      # Multiplicación del resultado por el polinomio
            original
42      u = multiplicacion_nc(u1, f, h, field)
43      coefs_u = [coeficiente for coeficiente in u.
            coefficients()]  # Coeficientes de u
44
45      return (coefs_u[0] ** (-1) * u)
```

- emb(elto_F, matriz_B2, L): está función realiza el embebimiento de un elemento del cuerpo F al cuerpo L utilizando la matriz B2 como base. Devuelve el elemento de F embebido en el cuerpo L:

```
def emb(elto_F, matriz_B2, L):
    """
    Embebe un elemento de un cuerpo F en un cuerpo L
       utilizando una matriz de base.

    Parámetros:
    elto_F (galois.Poly): El elemento del cuerpo F.
    matriz_B2 (np.array): La matriz de base utilizada para
        la transformación.
    L (galois.Field): El cuerpo en el que se embebe el
       elemento.

    Retorna:
    galois.Poly: El elemento embebido en el cuerpo L.
    """
    # Pasamos elto_F a coordenadas ascendentes
    vector = reverse_array(list(F.vector(elto_F)))

    # Extender el vector con ceros para el tamaño adecuado
    vector += [galois.GF(int(2))(0)] * (d * (m - 1))

    # Convertir la lista a un array del cuerpo Galois
    vector_GF2 = galois.GF(int(2))(vector)

    # Multiplicar el vector extendido por la matriz B2
    vector_L = np.dot(vector_GF2, matriz_B2)

    # Convertir el vector al cuerpo L
    elto_L = L.Vector(reverse_array(vector_L))

    return elto_L
```

- coord(elto_L, matriz_B2_inv, m, F): está función realiza el paso inverso a la función anterior, pues calcula las coordenadas de un elemento del cuerpo L en el cuerpo F usando la matriz inversa a la matriz de base B2 y devuelve un listado de los elementos de F que corresponden con esas coordenadas:

```
def coord(elto_L, matriz_B2_inversa, m, F):
    """
    Calcula las coordenadas de un elemento de un cuerpo L
       en un cuerpo F utilizando una matriz inversa.

    Parámetros:
    elto_L (galois.Poly): El elemento en el cuerpo L.
    matriz_B2_inversa (np.array): La matriz inversa
       utilizada para la transformación.
```

```
    m (int): Un parámetro relacionado con la extensión del
        cuerpo.
    F (galois.Field): El cuerpo objetivo para las
        coordenadas.

    Retorna:
    list: Una lista de elementos de F que corresponden a
        las coordenadas.
    """
    # Convertir el elemento L a un vector y luego invertir
        el orden
    vector_elto_L = reverse_array(L.vector(elto_L))

    # Multiplicar el vector por la matriz inversa
    vector_eltos_F = np.dot(vector_elto_L,
        matriz_B2_inversa)

    # Dividir el vector en subvectores según el tamaño m
    subvectores_F = np.split(vector_eltos_F, m)

    # Convertir los subvectores en elementos de F
    eltos_F = [F.Vector(reverse_array(list(subvector)))
        for subvector in subvectores_F]

    return eltos_F
```

- es_normal(elto_L, p, delta, mu, cuerpo): está función verifica si un elemento es normal, es decir, si ese elemento genera una base normal de la extensión de ese cuerpo. Devuelve true en caso de que lo sea, false en caso contrario:

```
def es_normal(elemento, p, delta, mu, cuerpo):
    """
    Verifica si un elemento es normal, es decir, si genera
        una base normal de la extensión K sobre L.

    Parámetros:
    elemento (galois.Poly): El elemento a verificar.
    p (int): La característica del cuerpo.
    delta (int): Un parámetro relacionado con la extensión
        .
    mu (int): El grado de la extensión.
    cuerpo (galois.Field): El cuerpo sobre el cual se
        realiza la verificación.

    Retorna:
    bool: True si el elemento genera una base normal,
        False en caso contrario.
    """
    # Crear el primer polinomio 1 - x^mu
    terminos_pol_1 = [1]
    for i in range(mu - 1):
```

```
18            terminos_pol_1.append(0)
19        terminos_pol_1.append(-1)
20
21        polinomio_1 = galois.Poly(terminos_pol_1,field=cuerpo)
22
23        # Crear el segundo polinomio con los términos elemento
              ^(p^(i*delta))
24        terms_2 = [elemento ** (p ** (i * delta)) for i in
              range(mu)]
25        polinomio2 = galois.Poly(terms_2, field=cuerpo)
26
27        # Calcular el máximo común divisor (MCD) de los dos
              polinomios
28        gcd = galois.gcd(polinomio_1, polinomio2)
29
30        # Verificar si el MCD es 1
31        return gcd == galois.Poly.One(cuerpo)
```

4.2.2. generar_clave.py

Como ya se ha explicado antes, este programa genera los parámetros necesarios para la construcción del criptosistema y genera las claves pública y privada, de acuerdo con [4]. Vamos a ver cómo se realiza esto:

Primero, se inicia el contador de tiempo para medir cuánto se tarda en generar la clave y se inicializan las variables n, t, p y d con valores deseados:

```
1  print()
2  print("Generando la clave de nuestro criptosistema.")
3  print()
4
5  inicio = time.time() #Iniciamos el tiempo de ejecución del
       programa
6
7  print("Calculando parámetros...")
8  print()
9
10 # CÁLCULO DE PARÁMETROS
11
12 #base_prime, base_exp = 2, 1
13 #base_prime, base_exp = 2, 2
14 #base_prime, base_exp = 2, 3
15 #base_prime, base_exp = 2, 4
16 base_prime, base_exp = 2, 8 #Byte
17 p, d = base_prime, base_exp
18
19 #length, correction_capability = 512, 6
20 length, correction_capability = 512, 5
21 #length, correction_capability = 1024, 10
22
23 n, t = length, correction_capability
```

Seguimos con el cálculo de m, δ y μ, restringiendo este último a qué debe ser mayor a $2t$ para dificultar más los posibles ataques dándole más variabilidad al polinomio g:

```
    # Calcula los valores de mmin y mmax
mmin, mmax = ceil(n / (10 * t)), floor(n / (4 * t))

k=n-(2*t*mmax)

# Encuentra las opciones válidas de m y delta
opciones = []
for m in range(mmin, mmax+1):
    for delta in range(1, d * m):
        if (d * m) % delta == 0 and n * delta / (d * (p **
            delta - 1)) <= m:
             opciones.append((m, delta))

#Elección de una pareja de m y delta válida
m, delta = random.choice(opciones)

#Cáluclo de mu
mu=(d*m)//delta

while(2*t<mu):
    m, delta = random.choice(opciones)
    mu=int((d*m)/delta)
```

Se calculan los últimos parámetros y se muestran por pantalla su valor:

```
    #Cálculo de h
h=random.randint(1,d*m)
while(gcd(h,d*m)!=delta):
    h=random.randint(1,d*m)

#Cálculo de r
r=(sum(p**(i*d) for i in range (m)))

print()
print("PARÁMETROS: ")
print("n: ",n, "t: ",t," p: ",p," d: ",d," k: ",k," m: ",m,"
    delta: ",delta," h: ",h, ", mu: ", mu, " r: ", r)
print()
```

A continuación, se calcula el cuerpo finito L, seleccionando un polinomio irreducible para definirlo. Además, se muestran por pantalla las propiedades del cuerpo creado y un elemento aleatorio del mismo:

```
#Construcción del cuerpo L

#Obtención del polinomio irreducible de L
polinomio_def_L = galois.primitive_poly(p, d*m)

#Construimos L con el polinomio irredicible calculado.
```

```
L = galois.GF(pow(p,d*m), irreducible_poly=galois.
    primitive_poly(p, d*m), repr='poly')
primitivo_L=L.primitive_element
num_aleatorio=random.randint(0, p**(d*m))
L_elemento = primitivo_L**num_aleatorio
print()
print("El cuerpo finito de Galois L es GF(p^d): ",L.properties)
print("Elemento random de L: ", L_elemento)
```

Hacemos lo mismo con F, pero en esta ocasión el polinomio irreducible usado para definir F se calcula partiendo del usado para definir L, permitiendo realizar las operaciones emb y coord de manera correcta:

```
    #Construcción del cuerpo F

#Obtención del polinomio irreducible de F a partir del de L
primitivo_FenL = primitivo_L**r

lista_irreducibles = list(galois.irreducible_polys(p,d))
for polinomio in lista_irreducibles:
    aux = list(polinomio.coefficients(order='asc'))
    suma = L(0)
    for i in range(len(aux)):
        suma += int(aux[i])*primitivo_FenL**i
    if suma == galois.Poly.Zero(field=L):
        polinomio_def_F = polinomio

#Construimos F con el polinomio irredicible calculado
F = galois.GF(pow(p,d), irreducible_poly=polinomio_def_F, repr=
    'poly')
F_elemento = random.choice(F.elements)
print()
print("El cuerpo finito de Galois F es GF(p^d): ",F.properties)
print("Elemento random de F: ", F_elemento)
```

Una vez ya tenemos los cuerpos finitos necesarios calculados, necesitamos calcular la matriz de base B2 y su inversa para poder realizar las operaciones emb y coord:

```
    # Obtención de las matrices de cambio de base entre los
        cuerpos F y L

B2=[primitivo_L**(i+j*r) for i in range(m) for j in range(d) ]

B2_aux = [reverse_array(list(L.vector(i))) for i in B2]

matriz_B2_np = np.array(B2_aux).reshape(d*m, d*m)

matriz_B2=galois.GF(int(2))(matriz_B2_np)
matriz_B2_inversa=np.linalg.inv(matriz_B2)

print()
print("Matriz para embebimiento de F en L: ", matriz_B2)
```

El siguiente paso consiste en calcular un elemento α que genere una base normal de L:

```
    alpha=L_elemento

while(not es_normal(alpha, p, delta, mu, L)): #Calculamos
    elementos hasta que sea normal
    num_aleatorio=random.randint(0, p**(d*m))
    alpha = primitivo_L**num_aleatorio

print("El elememento ",alpha," es normal.")

#Cálculo del vector normalizador siguiendo las restricciones
    correspondiente
print()
print("Calculando el vector normalizador...")
print()
```

Llegados a este punto, nos ponemos a calcular el vector normalizador (compuesto por n elementos no nulos de L) y el vector de puntos de posición/evaluación (compuesto por n elementos del conjunto $\{\gamma^j \alpha^{p^{hi}(p^{h-1})}$ tales que $0 \leq j \leq p^\delta - 2,\ 0 \leq i \leq \mu - 1\}$., donde *gamma* es un elemento primitivo del cuerpo L previamente definido:

```
    elementos_aleatorios_eta=[] #Definimos una lista donde
        guardaremos los elementos no nulos
barra_eta = Bar('Vector normalizador:', max=n) #Este elemento
    de la libreria progress genera una barra en la salida del
    programa que va mostrando el progreso del bucle

#Se deben seleccionar n elementos no nulos de L.
while len(elementos_aleatorios_eta)<n:
    num_aleatorio=random.randint(0, p**(d*m))
    elemento_eta = primitivo_L**num_aleatorio #Calculamos un
        elemento aleatorio de L

    if elemento_eta != L(0):
        elementos_aleatorios_eta.append(elemento_eta) #Si es no
            nulo, lo añadimos
        barra_eta.next() #Actualizamos la barra de progreso

print("Vector normalizador calculado")
print("Tamaño del vector normalizador: ", len(
    elementos_aleatorios_eta))

#Cálculo de los puntos de evaluación
print()
print("Cálculo de los puntos de evaluación...")
puntos_evaluacion=[] #Definimos una lista donde guardaremos los
     puntos de evaluación
gamma=L.primitive_element #Obtenemos un elemento primitivo de L
print("Gamma = ", gamma)
```

```
print()
barra_puntos_evaluacion = Bar('Puntos de evaluación:', max=n) # Este elemento de la libreria progress genera una barra en la salida del programa que va mostrando el progreso del bucle
#Selección de n elementos random diferentes del cuerpo L
while(len(puntos_evaluacion)<n):
    i=random.randint(0,mu-1)
    j=random.randint(0, (p**delta)-2)
    punto_evaluacion=(gamma**(j))*((alpha**(p**(h*(i+1))))/(alpha**((p**(h*i))))) #Calculamos el punto de evaluación
    if(punto_evaluacion not in puntos_evaluacion):
        puntos_evaluacion.append(punto_evaluacion) #Si el elemento no se encuentra ya en la lista, lo añadimos
        barra_puntos_evaluacion.next() #Actualizamos la barra de progreso

print("Puntos de evaluación calculados")
print("Tamaño del vector de puntos de evaluación: ", len(puntos_evaluacion))
```

El siguiente paso es calcular el polinomio no conmutativo modular de skew Goppa $g(x)$ para que cumpla las condiciones indicadas en [4] y usarlo para el cálculo de la lista de n polinomios no conmutativos de paridad, que se calculan siguiendo la siguiente expresión: $h_i = \text{PCP}(x - \alpha_i, g)$ para cada $i = 0, ..., n-1$. :

```
    #Construcción del polinomio g (normal o bilatero)
#G es un polinomio bilatero. estos se construyen como el producto de un polinomio central por una potencia de x.
#Los polinomios centrales tienen coeficientes en K y valores no nulos solo en potencias múltiplos de mu.
#Para ello:

print()
print("Construcción del polinomio de Goppa y los polinomios de paridad")

#Calculamos un primitivo de K (K es el subcuerpo de L de orden p^(delta))
exponente_K = sum(p**(i*delta) for i in range(mu))

primitivo_K = primitivo_L**exponente_K

#Fabricamos polinomio de K en L -> el grado es s=2t/mu (si no es entero, pillamos la parte entera)
#Queremos polinomio de ese grado -> pillamos una lista de 2t/mu +1 elementos (un 1 y 2t/mu elementos de K)
#Cómo cogemos eltos de K de forma aleatoria y homogénea? Repetimos S veces
#Cogemos número aleatorio entre 0 y p**(delta) -1 randint(0,(p
```

```
    **delta)-1 ). Si este número es p**(delta)-1 añadimos el 0
    a la lista.
#Si no, elevamos primK a ese numero aleatorio y añadimos.
    Tenemos coeficientes de polonimio de grado s

#Este comentario de arriba no se si quitarlo

cociente_S, resto_S=divmod(2*t,mu) #Calculamos el cociente y el
     resto de dividir 2t entre mu

repetir=True

while repetir: #Calculamos (en bucle, si es necesario) los
    coeficientes de g
    repetir=False
    coeficientes_g=[1] #El primer coeficiente de g es 1 para
        que g tenga una potencia de x
    for i in range(0,cociente_S): #De 0 a cociente_S-1 veces
        num_aleatorio=random.randint(0,(p**delta)-1) #
            Calculamos un número aleatorio entre 0 y p^(delta)
            -1
        if num_aleatorio == ((p**delta)-1):
            coeficientes_g.append(L(0)) #Si el número es
                exactamente p^(delta)-1, añadimos un 0 a los
                coeficientes
        else:
            coeficientes_g.append(primitivo_K**num_aleatorio) #
                Si no, añadimos el elemento correspondiente
                visto en K.

    print("Tamaño Coeficientes polinomio Goppa: ", len(
        coeficientes_g))

    #Tenemos que extender el polinomio de coeficientes g para
        que los coeficientes sean solo en posiciones múltiplos
        de mu y que g tenga el grado deseado
    coeficientes_extendidos_g = []
    i=mu-1
    j=resto_S
    for elemento in coeficientes_g:
        if(len(coeficientes_extendidos_g)>0):
            coeficientes_extendidos_g.extend([0] * i) #
                Calculamos y rellenamos las posiciones mú
                ltiplos de mu
        coeficientes_extendidos_g.append(elemento)

    # Añadir j ceros al final de la lista
    coeficientes_extendidos_g.extend([0] * j) #Extendemos el
        vector de coeficientes al final para generar un
        polinomio del grado deseado

    g=galois.Poly(coeficientes_extendidos_g, field=L) #
        Construimos el polinomio g
    print("Polinomio g construido. ")
```

```
    print()

    #Polinomios de paridad son los inversos de polinomios tipo x-punto_evaluacion mod g.
    print("Cálculando los polinomio no conmutativos de paridad ...")
    print()

    polinomios_nc_paridad=[] #Definimos una lista donde guardaremos los polinomios de paridad
    barra_polinomios_paridad = Bar('Polinomios de paridad:', max=len(puntos_evaluacion)) #Este elemento de la libreria progress genera una barra en la salida del programa que va mostrando el progreso del bucle
    for punto_evaluacion in puntos_evaluacion: #Para cada punto de evaluación
        polinomio=galois.Poly([1, punto_evaluacion], field=L) # calculamos el polinomio de la forma x-punto de evaluación
        polinomio_nc_paridad=PCP(polinomio, g, h, L) # Calculamos el polinomio nc de paridad correspondiente
        if(polinomio_nc_paridad==galois.Poly.Zero(L)):
            repetir=True
            print("Polinomio de Goppa no válido, reiniciamos el cálculo.") #Si algun modulo es 0, el polinomio g no es válido y volvemos a construirlo
            break
        polinomios_nc_paridad.append(polinomio_nc_paridad) #Añadimos el polinomio de paridad a la lista de polinomios de paridad
        barra_polinomios_paridad.next() #Actualizamos la barra de progreso

print()

print("Tamaño de la lista de polinomios no conmutativos de paridad H: ", len(polinomios_nc_paridad))
```

Ya tenemos todos los elementos del criptosistema necesarios para construir la matriz de paridad H, siguiendo las indicaciones de la propuesta de implementación, véase en [4]:

```
    #Cálculo de la matriz de paridad H
#Definición de las distintas matrices necesarias para constuir la clave pública
print("Cálculo de la matriz de paridad...")
H=[[L(0) for _ in range (n)] for _ in range (2*t)]
H_negada=[[L(0) for _ in range (n)] for _ in range (2*t)]
H_negada_en_F=[[[] for _ in range (n)] for _ in range (2*t)]
matriz_H=[[F(0) for _ in range (n)] for _ in range (2*t*m)] # matriz de paridad H

```

```
for j in range(n):
    coefs_polinomio=reverse_array(polinomios_nc_paridad[j].
        coefficients()) #Obtenemos los parámetros de menor a
        mayor de cada polinomio de paridad

    for i in range(2*t):
        H[i][j]=coefs_polinomio[i] #Guardamos los coeficientes
            de los polinomios de paridad
        H_negada[i][j]=(H[i][j]**(p**(((-i)%mu)*h)))*
            elementos_aleatorios_eta[j] #Calculamos los
            elementos de la matriz H_negada
        H_negada_en_F[i][j]=coord(H_negada[i][j],
            matriz_B2_inversa, m, F) #Calculamos las
            coordenadas en F de los elementos de H_negada

print("Paso 1 completado")
#Cálculo de los elementos de la matriz de paridad a partir de
for i in range(2*t*m):
    for j in range (n):
        a,b=divmod(i,m) #Calculamos las posiciones a y b
        matriz_H[i][j]=H_negada_en_F[a][j][b] #seleccionamos
            los elementos necesarios y los guardamos en la
            matriz de paridad H

print()
print("La matriz de paridad H se ha calculado. Tiene un tamaño
    de ",len(H),"x",len(H[0]))
```

Ahora, usamos la matriz de paridad H para calcular la clave pública H_{pub} de nuestro criptosistema, siguiendo lo indicado en [4]:

```
    #CÁLCULO DE LA CLAVE PÚBLICA
print()
print("Calculando la clave pública...")

matriz_H=F(matriz_H)

if np.linalg.matrix_rank(matriz_H)==(n-k): #Si el rango de la
    matriz de paridad es n-k,
    H_pub = matriz_H.row_reduce()          #la clave pública es
        la forma escalonada reducida por filas de la matriz de
        paridad,
else:
    sigue=True
    while(sigue): #Si no, repetimos en bucle:
        matriz_r=([[F(0) for _ in range (n)] for _ in range (n-
            k-np.linalg.matrix_rank(matriz_H))]) #Construimos
            una matriz auxiliar r de tamaño n-k-rango(matriz_H)
             x n
        matriz_r=F(matriz_r) #Transformamos la matriz de una
            lista de listas a una matriz del cuerpo F
        for i in range (n-k-np.linalg.matrix_rank(matriz_H)):
            for j in range(n):
```

```
                matriz_r[i][j]=random.choice(F.elements) #
                    Rellenamos la matriz r con elementos random
                     del cuerpo F

        matriz_q=matriz_H
        matriz_q=matriz_q.tolist() #Transformamos la matriz de
            una matriz de F a una lista de listas
        for fila in matriz_r:
            matriz_q.append(fila) #Añadimos las filas de la
                matriz r a la matriz aleatoria Q

        matriz_q = F(matriz_q) #Volvemos a convertir la matriz
            a una matriz de F
        matriz_q_rref = matriz_q.row_reduce() #Calculamos la
            forma escalonada reducida por filas de la matriz q
        H_pub = matriz_q_rref[:n - k] #La clave pública son las
             n-k primeras filas de la matriz escalonada
            reducida por filas de q
        if(np.linalg.matrix_rank(H_pub)==(n-k)): #Si la clave p
            ública tiene rango n-k es válida, si no se repite
            este proceso
            sigue=False

print("Tenemos la clave pública, una matriz de dimensiones ",
    len(H_pub),"x",{len(H_pub[0])})
```

Para terminar con generar_clave.py, guardamos las claves públicas y privadas en los archivos correspondientes de la carpeta archivos y paramos el tiempo de ejecución. Mostramos por pantalla el nombre de los archivos generados y el tiempo transcurrido:

```
    Guardamos la clave en archivos simulando su envio a un
        transmisor y receptor

#El receptor necesita todos los parámetros y elementos de la
    clave para desencapsular
clave_privada='Clave.npz'
np.savez("../archivos/"+clave_privada, H_pub=H_pub, n=n, t=t, p
    =p, d=d, k=k, m=m, delta=delta, h=h, mu=mu, r=r,
    polinomio_def_L=polinomio_def_L.coefficients(),
    polinomio_def_F=polinomio_def_F.coefficients(),
    polinomios_nc_paridad=polinomios_nc_paridad,
    elementos_aleatorios_eta=elementos_aleatorios_eta,
    puntos_evaluacion=puntos_evaluacion, g=g.coefficients(),
    matriz_B2=matriz_B2)
print()
print("La clave privada y los parámetros del criptosistema se
    han guardado en el archivo ",clave_privada, "de la carpeta
    archivos")

#El transmisor necesita algunos parámetros y elementos de la
    clave para encapsular
clave_publica='Clave_pub.npz'
```

```
11 np.savez("../archivos/"+clave_publica, H_pub=H_pub, n=n, t=t, p
       =p, d=d, k=k, polinomio_def_F=polinomio_def_F.coefficients
       ())
12 print()
13 print("La clave pública y los parámetros del criptosistema se
       han guardado en el archivo ",clave_publica, "de la carpeta
       archivos")
14
15 #Obtenemos el tiempo que ha tardado la generación de la clave y
        lo mostramos por pantalla
16 final = time.time() #Finalizamos el tiempo de ejecución del
       programa
17 tiempo_transcurrido = final - inicio #Calculamos el tiempo
       transcurrido durante la ejecución
18 print("Clave generada en: ",tiempo_transcurrido," segundos.")
```

4.2.3. encapsular.py

El programa **encapsular.py** coge la clave pública del archivo Clave_pub.npz generado anteriormente, extrae los elementos necesarios de ella y crea un criptograma a partir de un secreto aleatorio que se desea enviar al receptor. Por último, crea un hash del secreto compartido y guarda en archivos tanto el criptograma como el hash del secreto. El código de encapsular.py se basa en [4]

Primero, se inicia el cálculo del tiempo de ejecución y se extrae la clave y sus elementos del archivo cuyo nombre (sin la ruta a la carpeta /archivos, que se añade automáticamente en el programa), se pasa como parámetro al programa. En caso de no pasar bien los parámetros, el programa informa del error:

```
1     # Verificar que se pasen dos argumentos
2 if len(sys.argv) != 2: #Si hay error en los argumentos, se
      indica cómo debe ejecutarse
3     print()
4     print("ERROR: Debe ejecutar el archivo con el siguiente
          comando: python.exe encapsular.py <
          nombre_archivo_clave_publica>")
5     sys.exit(1)
6
7 inicio = time.time() #Iniciamos el tiempo de ejecución del
      programa
8
9 # Obtener el de los archivos desde los argumentos
10 archivo_clave = "../archivos/"+sys.argv[1]
11
12 print()
13 print("Encapsulamiento de un secreto compartido usando la clave
       previamente generada")
14
15
```

```
# Cargar los datos desde el archivo de clave .npz
data = np.load(archivo_clave, allow_pickle=True)

# Obtenemos los parámetros del archivo y los convertimos al
    tipo correspondiente)
n = int(data['n'])
t = int(data['t'])
p = int(data['p'])
d = int(data['d'])
k = int(data['k'])

#Construcción del cuerpo F

#Obtenemos los coeficientes del polinomio irreducible de F y
    construimos el polinomio.
coefs_def_F = data['polinomio_def_F']
polinomio_def_F=galois.Poly(coefs_def_F,field=galois.GF(2))

#Construimos el cuerpo F
F = galois.GF(pow(p,d), irreducible_poly=polinomio_def_F, repr=
    'poly')
F_elemento = random.choice(F.elements)
print()
print("El cuerpo finito de Galois F es GF(p^d): ",F.properties)
print("Elemento random de F: ", F_elemento)

#Obtenemos la clave pública del archivo de la clave
H_pub = data['H_pub']
H_pub=F(H_pub) #La transformamos a una matriz de F
H_Pub_T=H_pub.T #Obtenemos la matriz traspuesta de la clave pú
    blica

print()
print("La clave pública H_pub ha sido cargada correctamente")

print()
print("PARÁMETROS:")
print("n: ",n," t: ",t," p: ",p," d: ",d," k: ",k)
```

A continuación, se calcula el secreto aleatorio que se quiere compartir, creando un vector de n elementos de F con t elementos no nulos:

```
    print("Cálculo del secreto compartido...")
print()
#Calculamos el secreto compartido:

secreto_compartido = F([F(0) for _ in range(n)]) #Creamos una
    lista de n elementos nulos

contador = 0
while contador < t: #Ejecutamos t veces
    pos = random.randint(0, n - 1) #Calculamos una posición
        random de la lista
    valor_no_nulo = random.choice(F.elements) #Calculamos un
```

```
        elemento random de F
    if (secreto_compartido[pos] == F(0)) and (valor_no_nulo !=
        F(0)): #Si la posición es nula y el elemento no es nulo
        secreto_compartido[pos] = valor_no_nulo #Cambiamos la
            posición elegida de la lista al valor aleatorio
        contador += 1 #Aumentamos el contado
```

Una vez generado el secreto compartido, procedemos a encapsularlo multiplicándolo por la traspuesta de la clave pública H_{pub}, generando así el criptograma. Este se guarda en el archivo criptograma.npy, cuyo nombre se muestra por pantalla:

```
    print("Cálculo del criptograma...")
#Cálculo del criptograma
criptograma=F(secreto_compartido@H_Pub_T) #El criptograma es la
    matriz en F resultante de multiplicar e secreto compartido
    por la traspuesta de la clave pública

# Mostrar el criptograma generado por pantalla
print()
print("Criptograma generado para el envio: ",criptograma)

# Guardar el criptograma en un archivo llamado "criptograma.npy
    "
criptograma_file_name = "criptograma.npy"
np.save("../archivos/"+criptograma_file_name, criptograma)

print()
print("El criptograma se ha guardado en el archivo: ",
    criptograma_file_name, "de la carpeta archivos")
```

Por último, se crea el hash correspondiente al secreto compartido, se guarda en el archivo hash_encapsulado.py y se para el tiempo de ejecución. Se muestra por pantalla el archivo del hash y el tiempo de ejecución:

```
    #Creación de un hash del secreto compartido para testear
        posteriormente el funcionamiento del sistema

# Crear un objeto hash usando SHA-3 (en este caso, sha3_256)
hash_obj = hashlib.sha3_256()

secreto_str = str(secreto_compartido)

# Alimentar el objeto hash con el texto en bytes
hash_obj.update(secreto_str.encode('utf-8'))

# Obtener el hash resultante en formato hexadecimal
hash_result = hash_obj.hexdigest()

print()
print("El secreto compartido ha generado el siguiente hash: ",
    hash_result)

```

```
17
18 #GUARDAR EL HASH DEL SECRETO COMPARTIDO EN UN ARCHIVO PARA
       COMPROBAR QUE FUNCIONA ENCAPSULADO Y DES()
19
20 # Convertir el hash a un objeto numpy (array de cadena de texto
       )
21 hash_array = np.array([hash_result])
22 # Guardar el hash en un archivo usando np.save
23 file_name = "hash_encapsulado.npy"
24 np.save("../archivos/"+file_name, hash_array)
25
26 print()
27 print("El hash se ha guardado en el archivo: ",file_name, "de
       la carpeta archivos")
28
29 final = time.time() #Finalizamos el tiempo de ejecución del
       programa
30 tiempo_transcurrido = final - inicio #Calculamos el tiempo
       transcurrido durante la ejecución
31 print("Secreto compartido encapsulado en: ",tiempo_transcurrido
       ," segundos.")
```

4.2.4. desencapsular.py

Este tercer programa de nuestro sistema coge la clave privada del primer archivo que se le pase como parámetro, y el criptograma del segundo archivo que se le pase como parámetro. De igual manera que en encapsular.py, si hay algún fallo se le indica al usuario. El programa aplica el algoritmo de desencapsulado al criptograma usando la clave, y obtiene el vector de salida. También se genera un hash del vector de salida. Se guarda el hash en el archivo hash_desencapsulado.py.

La propuesta de implementación [4] nos explica cómo generar el vector y cómo aplicarle el algoritmo de desencapsulado. Además, nos proporciona un diagrama donde se explica el algoritmo de desencapsulado:

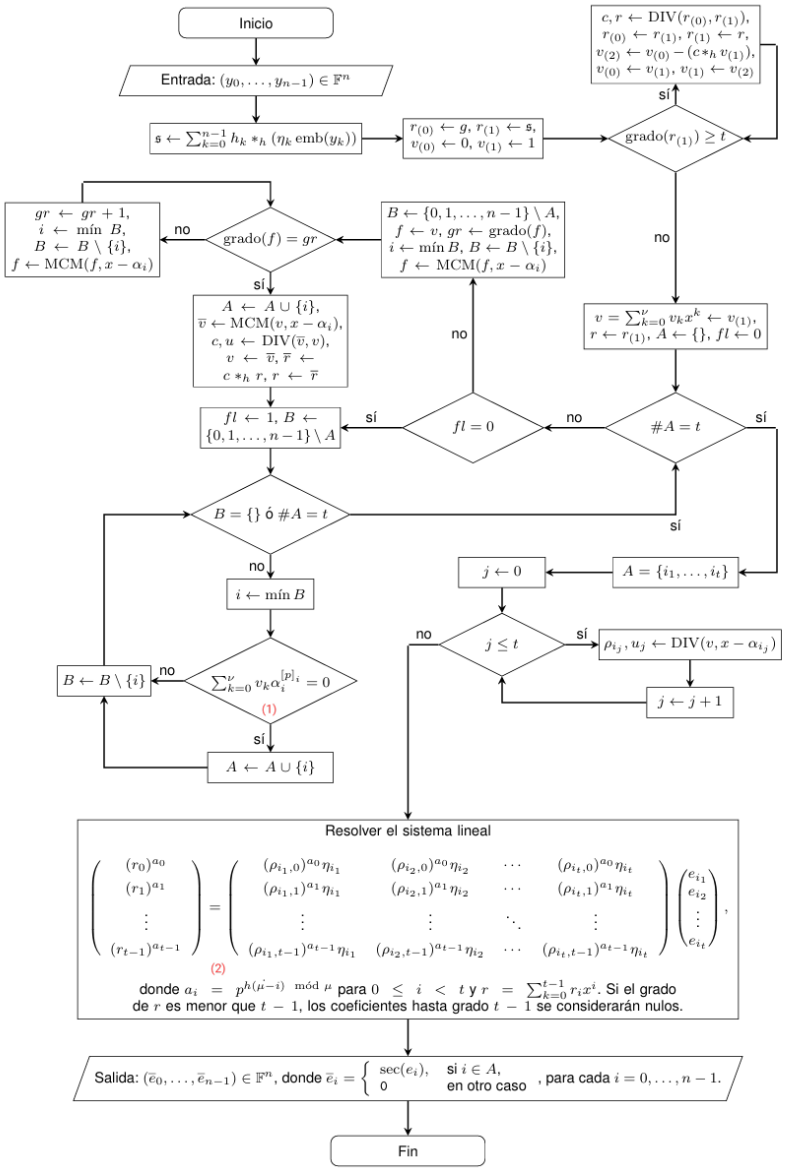

Figura 4.7: Algoritmo de desencapsulado, véase en [4]

*Nótese en la figura 4.7 las siguientes erratas:

- La ecuación (1) en realidad es: $\sum_{k=0}^{\nu} v_k \alpha_i^{[p]_k}$
- en (2), el cálculo de a_i es: $a_i = p^{h((-i) \bmod \mu)}$

Lo primero que hace el programa desencapsular.py es obtener los nombres de los archivos que se le pasan como parámetros e iniciar el tiempo de ejecución:

```
    # Verificar que se pasen dos argumentos (los nombres de los
        archivos)
if len(sys.argv) != 3:
    print()
    print("ERROR: Debe ejecutar el archivo con el siguiente
        comando: python.exe encapsular.py <nombre_archivo_clave
        > <nombre_archivo_criptograma>")
    sys.exit(1)

# Obtener el de los archivos desde los argumentos
archivo_clave = "../archivos/"+sys.argv[1]
#Obtener el criptograma del archivo
archivo_criptograma = "../archivos/"+sys.argv[2]

inicio = time.time() #Iniciamos el tiempo de ejecución del
    programa
```

Una vez obtenidos los archivos de donde obtener la clave y el criptograma, extraemos de los archivos los parámetros de la clave, la clave y el criptograma:

```
    # Cargar los datos desde el archivo .npz
data = np.load(archivo_clave, allow_pickle=True)

print()
print("Desencapsulamiento de un criptograma recibido usando la
    clave previamente generada")

print("Obtención de los parámetros del archivo...")
# Obtenemos los parámetros del archivo y los convertimos al
    tipo correspondiente
n = int(data['n'])
t = int(data['t'])
p = int(data['p'])
d = int(data['d'])
k = int(data['k'])
m = int(data['m'])
delta = int(data['delta'])
h = int(data['h'])
mu = int(data['mu'])


#Construcción del cuerpo L

#Obtenemos el polinomio irreducible de L del archivo
coefs_def_L = data['polinomio_def_L']
polinomio_def_L = galois.Poly(coefs_def_L,field=galois.GF(2))

#Construimos el cuerpo L y obtenemos algunos elementos del
```

```
    mismo
L = galois.GF(pow(p,d*m), irreducible_poly=polinomio_def_L,
    repr='poly')
primitivo_L=L.primitive_element
num_aleatorio=random.randint(0, p**(d*m))
L_elemento = primitivo_L**num_aleatorio
print()
print(f"El cuerpo finito de Galois L es GF(p^(d*m)): {L.
    properties}")
print(f"Elemento random de L: {L_elemento} ")

#Construcción del cuerpo F

#Obtenemos el polinomio irreducible de F del archivo
coefs_def_F = data['polinomio_def_F']
polinomio_def_F = galois.Poly(coefs_def_F,field=galois.GF(2))

#Construimos el cuerpo F y obtenemos algunos elementos del
    mismo
F = galois.GF(pow(p,d), irreducible_poly=polinomio_def_F, repr=
    'poly')
F_elemento = random.choice(F.elements)
print()
print(f"El cuerpo finito de Galois F es GF(p^d): {F.properties}
    ")
print(f"Elemento random de F: {F_elemento} ")

#Obtención del resto de elementos de la clave del archivo
r= int(data['r'])

matriz_B2 = data['matriz_B2']
matriz_B2=galois.GF(2)(matriz_B2)
matriz_B2_inv = np.linalg.inv(matriz_B2) #Cálculo de la matriz
    inversa de B2

polinomios_nc_paridad = data['polinomios_nc_paridad']
elementos_aleatorios_eta = data['elementos_aleatorios_eta']
puntos_evaluacion = data['puntos_evaluacion']
g_coefficients = data['g']
g = galois.Poly(g_coefficients, field=L)

#Obtención de la clave pública del archivo
H_pub = data['H_pub']
H_pub=F(H_pub)
print("Clave obtenida del archivo")


#Obtención del criptograma del archivo
criptograma = F(np.load(archivo_criptograma, allow_pickle=True)
    )
print()
print("Criptograma recibido desde el archivo.")

print()
```

```
print("PARÁMETROS:")
print("n: ",n, "t: ",t," p: ",p," d: ",d," k: ",k," m: ",m,"
    delta: ",delta," h: ",h, ", mu: ", mu, " r: ", r)
print()
```

Llegados a este punto de la ejecución, ya se han generado los parámetros y la clave, y se ha obtenido el criptograma, pero el algoritmo de desencapsulado obtiene como entrada el vector y. Necesitamos construirlo, para ello definimos un vector de n elementos nulos. Calculamos la posición donde se encuentra el pivote j de la fila i de H_{pub} y calculamos $y[j] = criptograma[i]$, para cada fila i de la matriz H_{pub}:

```
    #Cálculo del vector y
y=[F(0)] * n #Creamos un vector y con n posiciones nulas

# OBTENER POSICIONES DE LOS PIVOTES de H_pub
posiciones = []

# Iterar sobre cada fila de la matriz
for i in range(len(H_pub)):
    # Inicializamos la posición como -1 (por si no hay
        elementos no nulos)
    posicion = -1
    # Recorremos cada elemento de la fila
    for j in range(len(H_pub[i])):
        if H_pub[i][j] != 0: #Cuando se diferente de 0 (será el
            pivote)
            posicion = j
            break  # Salimos del bucle cuando encontramos el
                primer no nulo

    y[posicion]=criptograma[i] #Actualizamos la posición donde
        se encuentra el pivote en el vector y con el
        criptograma correspondiente de la fila de H_pub

print("Polinomio y calculado.")
```

Como ya hemos calculado el vector y, que es la entrada del algoritmo de desencapsulado, ya podemos aplicar el algoritmo para obtener el vector de salida:

```
    #ALGORITMO DEC
print()
print("Comienzo del algoritmo DEC(): ")

print()
print("Calculando el polinomio síndrome...")

#Calculamos el síndrome (s)
barra_syndrome = Bar('Polinomio síndrome:', max=n) #Este elemento de la libreria progress genera una barra en la salida del programa que va mostrando el progreso del bucle
s=galois.Poly(L(0), field=L) #Definimos el polinomio sindrome como s

for i in range(n): #Recorremos todo el vector y
    elto_L = emb(y[i], matriz_B2, L) # Calcular el término
    resultado = multiplicacion_nc(polinomios_nc_paridad[i], galois.Poly(elementos_aleatorios_eta[i] * elto_L, field=L), h, L) # Realizar la multiplicación
    s += resultado # Acumulación en `s`
    barra_syndrome.next() #Actualizamos la barra de progeso

print()
print("Grado del polinomio síndrome: ", s.degree)

print()
print("Cálculo de polinomios localizador y evaluador de errores")

r0 = g  # r0 se inicializa con el polinomio g de la clave
r1 = s  # r1 se inicializa con el polinomio s
v0 = galois.Poly(L(0), field=L)  # Polinomio inicial v0, que es el polinomio cero en el cuerpo L
v1 = galois.Poly(L(1), field=L)  # Polinomio inicial v1, que es el polinomio uno en el cuerpo L

# Iteración usando el algoritmo de Euclides extendido hasta que el grado de r1 sea menor que t
while r1.degree >= t:

    c, r = division_nc(r0, r1, h, L) # División no conmutativa de r0 por r1 en el cuerpo L con un parámetro h
    r0 = r1  # r0 toma el valor de r1 (anterior)
    r1 = r   # r1 toma el valor del residuo r (nuevo)

    v2 = (v0 - (multiplicacion_nc(c, v1, h, L)))  # Se calcula el nuevo v2 como v0 - c * v1# Actualización del polinomio v usando la operación de multiplicación y sustracción
```

```
37
38      # Desplazamiento de v0 y v1 para la próxima iteración
39      v0 = v1  # v0 toma el valor de v1 (anterior)
40      v1 = v2  # v1 toma el valor de v2 (nuevo)
41
42  v = v1 # Al final del bucle, v1 contiene el polinomio
        localizador y evaluador de errores
43
44  print()
45  print("Grado del polinomio localizador y evaluador de errores:
        ", v.degree)
46
47
48  if(v.degree != t): #Si el grado del polinomio localizador no es
        t, ha habido un error
49      print("Fallo de desencapsulado!")
50      #Terminar ejecución
51      sys.exit(1)
52
53  r = r1 # Inicializa el polinomio r con el valor de r1
54  A = set()  # Conjunto que almacenará las posiciones de error
        encontradas, inicialmente vacio
55  fl = 0  # Indicador para determinar la primera ejecución del
        bucle principal
56
57  # Bucle que continúa hasta que se encuentren exactamente t
        posiciones de error
58  while len(A) != t:
59
60      if fl != 0:  # Si no es la primera iteración
61
62          B = set(range(n)) - A # B es el conjunto complementario
                de A en el rango de evaluación
63
64
65          f = v # Se inicializa f como el polinomio v
66          gr = f.degree  # Se guarda el grado del polinomio f
67          i = min(B)  # Se selecciona el menor índice de B
68          B.remove(i)  # Se elimina el índice i de B
69
70          # Se calcula el MCM de f y (x - punto_de_evaluacion[i])
                y se actualiza f
71          f = MCM(f, galois.Poly([1, puntos_evaluacion[i]], field
                =L), h, L)
72
73          # Se incrementa el grado de f hasta que sea igual al
                grado inicial gr, recorriendo todo el conjunto B
74          while f.degree != gr:
75              gr += 1
76              i = min(B)
77              B.remove(i)
78              f = MCM(f, galois.Poly([1, puntos_evaluacion[i]],
                    field=L), h, L)
79
```

```
80          # Añade la posición i al conjunto de posiciones de
                error A
81          A.add(i)
82
83          # Se calcula el MCM de v y (x - punto_de_evaluacion[i])
                para obtener v_negada
84          v_negada = MCM(v, galois.Poly([1, puntos_evaluacion[i
                ]], field=L), h, L)
85
86          # Se divide v_negada por v para obtener el cociente c
87          c, u = division_nc(v_negada, v, h, L)
88
89          # Se actualiza v con v_negada
90          v = v_negada
91
92          r_negada = multiplicacion_nc(c, r, h, L) # Se
                multiplica c por r
93          r = r_negada #Se actualiza r con el resultado
94
95      # Marca que se ha completado la primera iteración
96      fl = 1
97
98      # B es el conjunto de índices no presentes en A
99      B = set(range(n)) - A
100
101     print()
102     print("Calculando las posiciones de error...")
103     print()
104
105     # Bucle para encontrar las posiciones de error restantes (
            en la primera iteración entra aquí directamente)
106     while len(B) > 0 and len(A) != t:
107         i = min(B)  # Selecciona el menor índice de B
108         vector_v = reverse_array(v.coefficients())  # Se
                obtienen los coeficientes de v de menor a mayor
109         polinomio = L(0)  # Inicializa el polinomiko 0 en el
                cuerpo L
110
111         # Calcula el polinomio en el punto de evaluación
112         for k in range(len(vector_v)):
113             Pk = ((p ** (k * h)) - 1) // ((p ** h) - 1)
114             polinomio += (vector_v[k] * (puntos_evaluacion[i]
                    ** Pk))
115
116         # Si el polinomio evaluado es cero, se ha encontrado
                una posición de error y se añade la posición a la
                lista A
117         if polinomio == galois.Poly([0], field=L):
118             A.add(i)
119
120         # Elimina el índice i de B
121         B.remove(i)
122
123     # Si no se encuentran suficientes posiciones de error, se
```

```
        asume que el criptograma o clave es incorrecta
124     if len(A) < t:
125         print("Criptograma o clave incorrecta.")
126         # Termina la ejecución del programa
127         sys.exit(1)
128 
129 print("Posiciones de error calculadas con éxito")
130 
131 #Calculamos elementos necesarios para constuir el sistema de
        ecuaciones
132 rho = [L(0)] * t #Calculamos una lista de t elementos nulos
133 u = [L(0)] * t #Calculamos una lista de t elementos nulos
134 A=list(A) #Convertimos el conjunto A a una lista
135 A.sort() #Ordenamos la lista A
136 
137 print("Construyendo el sistema de ecuaciones para obtener el
        secreto compartido...")
138 for j in range(t): #Cálculo del vector rho, realizando la
        división correspondiente
139     rho[j],u[j] = division_nc(v, galois.Poly([1,
            puntos_evaluacion[A[j]]], field=L), h, L)
140 
141 
142 #RESOLVER SISTEMA LINEAL FINAL
143 vector_a=[0] * t  #Creamos el vector a con t posiciones nulas
144 
145 for i in range (t):
146     vector_a[i]=p ** (h * ((-i) % mu)) #Rellenamos el vector a
            con el cálculo correspondiente
147 
148 coefs_r=reverse_array(r.coefficients()) #Obtenemos los
        coeficientes de r de menor a mayor
149 num_ceros = (t) - len(coefs_r) #Si el grado es menor que t,
        rellenamos con 0
150 if num_ceros > 0:
151     coefs_r = coefs_r +( [0] * num_ceros)
152 
153 matriz_r= [L(0)] * t #Creamos la matriz de una fila r con t
        elementos nulos
154 for i in range(t):
155     matriz_r[i]=coefs_r[i]**vector_a[i] #Rellenamos r con el cá
            lculo correspondiente
156 matriz_r=L(matriz_r).T #Trasponemos r para que sea una matriz
        de una columna
157 
158 matriz_sistema= [[L(0) for _ in range(t)] for _ in range(t)] #
        Creamos la matriz sistema de t*t elementos nulos
159 
160 for j in range (t): #Rellenamos la matriz sistema
161 
162     coefs_rho=reverse_array(rho[j].coefficients()) #Obtenemos
            los coeficientes de rho de menor a mayor
163     for i in range (t):
164         #Actualizamos cada elemento de matriz sistema con el cá
```

```
            lculo correspondiente
165         matriz_sistema[i][j]=(coefs_rho[i]**vector_a[i])*(
                elementos_aleatorios_eta[A[j]])
166
167 matriz_sistema=L(matriz_sistema) #Convertimos la matriz sistema
        a una matriz de L
168
169
170 print("Resolviendo el sistema de ecuaciones...")
171
172 #Tenemos el siguiente sistema de ecuaciones: matriz_r=
        matriz_sistema*matriz_e
173 solucion_e =np.linalg.solve(matriz_sistema, matriz_r) #Hallamos
        la solución del sistema de ecuaciones con la libreria
        numpy
174
175 print("Sistema resuelto")
176
177 print()
178 print("Obteniendo el valor del secreto compartido...")
179
180 #Obtención de la salida del algoritmo de desencapsulado:
181 vector_salida=F([F(0) for _ in range(n)]) #Creamos el vector de
        salida con n posiciones nulas
182
183 for i in range(len(solucion_e)): #recorremos la solución del
        sistema de ecuaciones
184     lista_eltos_F = coord(solucion_e[i], matriz_B2_inv, m, F) #
            Obtenemos las coordenadas en F del elemento en L de la
            solución
185     primer_elto_F = lista_eltos_F[0] #Nos quedamos con la
            primera coordenada
186     vector_salida[A[i]]=primer_elto_F #Actualizamos la posición
             del error correspondiente con el elemento
            correspondiente
187
188 #Hemos desencapsulado el criptograma y obtenido el secreto
        compartido
189 print("Secreto compartido obtenido.")
```

Para terminar con la ejecución del programa desencapsular.py, vamos a generar un hash a partir del vector de salida, para posteriormente poder comparar este hash con el originado a partir del secreto compartido y comprobar que el encapsulado y desencapsulado han funcionado bien. Guardamos este hash en el archivo hash_desencapsulado.npy. También se termina con la medición del tiempo de ejecución y se muestra este tiempo por pantalla:

```
    #Creación de un hash del secreto compartido para testear posteriormente el funcionamiento del sistema

print("Generando un hash para el secreto compartido recibido...")
# Crear un objeto hash usando SHA-3 (en este caso, sha3_256)
hash_obj = hashlib.sha3_256()

salida_str = str(vector_salida)

# Alimentar el objeto hash con el texto en bytes
hash_obj.update(salida_str.encode('utf-8'))

# Obtener el hash resultante en formato hexadecimal
hash_result = hash_obj.hexdigest()

print("El secreto compartido ha generado el siguiente hash: ", hash_result)

#GUARDAR EL HASH DEL SECRETO COMPARTIDO EN UN ARCHIVO PARA COMPROBAR QUE FUNCIONA ENCAPSULADO Y DES()

# Convertir el hash a un objeto numpy (array de cadena de texto)
hash_array = np.array([hash_result])
# Guardar el hash en un archivo usando np.save
file_name = "hash_desencapsulado.npy"
np.save("../archivos/"+file_name, hash_array)

print()
print("El hash del secreto compartido recibido se ha guardado en el siguiente archivo: ", file_name, "de la carpeta archivos")

final = time.time() #Finalizamos el tiempo de ejecución del programa
tiempo_transcurrido = final - inicio #Calculamos el tiempo transcurrido durante la ejecución
print()
print("Criptograma desencapsulado en: ",tiempo_transcurrido," segundos.")
```

4.2.5. comparar_hashes.py

Este último programa es el más sencillo del sistema. Este programa recibe el nombre de dos archivos como parámetros y obtiene los hashes generados en la encapsulación y la desencapsulación de estos archivos. Después, los compara para comprobar si el sistema funciona de manera correcta.

Primero, el programa extrae los hashes de los archivos:

```
    # Verificar que se pasen dos argumentos (los nombres de los archivos)
if len(sys.argv) != 3:
    print("Uso: python comparar_hashes.py <archivo_hash1.npy> <archivo_hash2.npy>")
    sys.exit(1)

inicio = time.time() #Iniciamos el tiempo de ejecución del programa

print()
print("Comparación de hashes para comprobar el funcionamiento del ciptosistema.")
print()

# Obtener los nombres de los archivos desde los argumentos
file_encapsulado = sys.argv[1]
file_dencapsulado = sys.argv[2]

# Cargar el hash generado en el encapsulado desde el archivo
hash_encapsulado_array = np.load("../archivos/"+file_encapsulado)
hash_encapsulado = hash_encapsulado_array[0]

# Cargar el hash generado en el desencapsulado desde el archiv
hash_encapsulado_array = np.load("../archivos/"+file_dencapsulado)
hash_desencapsulado = hash_encapsulado_array[0]
```

Una vez ha obtenido los hashes, los compara. Si son iguales, significa que el sistema funciona correctamente y muestra ese mensaje por pantalla, junto con el tiempo de ejecución de este programa.

```
    # Mostrar los hashes cargados
print(f"Hash del encapsulado {file_encapsulado}: {hash_encapsulado}")
print(f"Hash del desencapsulado {file_dencapsulado}: {hash_desencapsulado}")
print()
```

```
# Comparar los hashes y mostrar el resultado
if hash_encapsulado == hash_desencapsulado:
    print("Los hashes son iguales. El secreto compartido se ha
        transmitido con éxito")
    print()
    print("EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE")
    print()
    print("SE ACABÓ :)")
else:
    print("ERROR: Los hashes son diferentes.")

final = time.time() #Finalizamos el tiempo de ejecución del
    programa
tiempo_transcurrido = final - inicio #Calculamos el tiempo
    transcurrido durante la ejecución
print()
print("Hashes comparados en: ",tiempo_transcurrido," segundos."
    )
```

Ya hemos finalizado con la explicación de la implementación realizada, la cuál está basada en la propuesta de implementación realizada por mi tutor del TFG junto con su equipo de trabajo., véase en [4].

Capítulo 5

Resultados

En este quinto capítulo se van a mostrar los resultados de diversas ejecuciones de nuestro sistema. Se generarán 3 claves diferentes, encapsulando y desencapsulando 3 mensajes diferentes con cada una, para comprobar tanto si el resultado del proceso de encapsulación y desencapsulación es el correcto, como para analizar posteriormente los tiempos de ejecución de cada parte del sistema.

Para estas ejecuciones se ha decidido usar como *parámetros iniciales* **p**=2, **d**=8, **n**=512 y **t**=5. Nótese que si se desean realizar pruebas con otros parámetros iniciales (lo cuál afecta al tiempo de ejecución ya que se estarían modificando los tamaños, los errores a corregir y las definiciones del cuerpos finitos que se usan), se pueden probar las siguientes combinaciones de n, t, p y d:

- n=512, t=5, p=2, d=4
- n=1024, t=10, p=2, d=8
- n=1024, t=10, p=2, d=4

5.1. Resultados de ejecución

En esta sección se van a mostrar los resultados de la ejecución del sistema completo 3 veces, generando 3 claves, y de una parte del sistema 9 veces ya que generamos 3 criptogramas con cada clave.

5.1.1. Primera clave

Primero, *generamos la primera clave*. En este caso, los parámetros con los que se construirá la clave serán:

```
PS C:\Users\gonza\Desktop\Prácticas\4º Ing Informática\TFG\src> python.exe generar_clave.py

Generando la clave de nuestro criptosistema.

Calculando parámetros...


PARÁMETROS:
n:  512 t:  5  p:  2  d:  8  k:  262  m:  15  delta:  20  h:  20 , mu:  6  r:  52126588069996700898188512167856 65
```

Figura 5.1: Parámetros de la clave número 1

Y con estos parámetros obtenemos el siguiente resultado:

```
Tenemos la clave pública, una matriz de dimensiones  250 x {512}

La clave privada y los parámetros del criptosistema se han guardado en el archivo  Clave.npz de la carpeta archivos

La clave pública y los parámetros del criptosistema se han guardado en el archivo  Clave_pub.npz de la carpeta archivos
Clave generada en:  138.2684977054596  segundos.
```

Figura 5.2: Resultado de generar la clave 1

Una vez obtenida la primera clave, podemos ejecutar la *primera encapsulación y desencapsulación de la primera clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  296aa31f8cae298b91c8966eb75eff70260ae2afe766463a61395ed651ad76a3

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  2.822549819946289  segundos.
```

Figura 5.3: Resultado de la primera encapsulación con la primera clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  296aa31f8cae298b91c8966eb75eff70260ae2afe766463a61395ed651ad76a3

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  15.585115194320679  segundos.
```

Figura 5.4: Resultado de la primera desencapsulación con la primera clave

```
Hash del encapsulado hash_encapsulado.npy: 296aa31f8cae298b91c8966eb75eff70260ae2afe766463a61395ed651ad76a3
Hash del desencapsulado hash_desencapsulado.npy: 296aa31f8cae298b91c8966eb75eff70260ae2afe766463a61395ed651ad76a3

Los hashes son iguales. El secreto compartido se ha transmitido con éxito

¡¡EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE!!

SE ACABÓ :)

Hashes comparados en:  0.008370399475097656  segundos.
```

Figura 5.5: Resultado de la comparación del primer encapsulado-desencapsulado con la primera clave

Seguimos con la ejecución de la *segunda encapsulación y desencapsulación de la primera clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  98df2cc91d178ab75450038f45a073f5e24f3e040b302bddf71151894c7d45b0

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  2.764902114868164  segundos.
```

Figura 5.6: Resultado de la segunda encapsulación con la primera clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  98df2cc91d178ab75450038f45a073f5e24f3e040b302bddf71151894c7d45b0

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  15.431063890457153  segundos.
```

Figura 5.7: Resultado de la segunda desencapsulación con la primera clave

```
Comparación de hashes para comprobar el funcionamiento del ciptosistema.

Hash del encapsulado hash_encapsulado.npy: 98df2cc91d178ab75450038f45a073f5e24f3e040b302bddf71151894c7d45b0
Hash del desencapsulado hash_desencapsulado.npy: 98df2cc91d178ab75450038f45a073f5e24f3e040b302bddf71151894c7d45b0

Los hashes son iguales. El secreto compartido se ha transmitido con éxito

¡¡EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE!!

SE ACABÓ :)

Hashes comparados en:  0.015131711959838867  segundos.
```

Figura 5.8: Resultado de la comparación del segundo encapsulado-desencapsulado con la primera clave

Por último, realizamos la *tercera encapsulación y desencapsulación de la primera clave*, la cuál nos deja estos resultados:

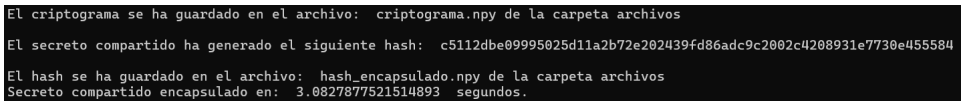

Figura 5.9: Resultado de la tercera encapsulación con la primera clave

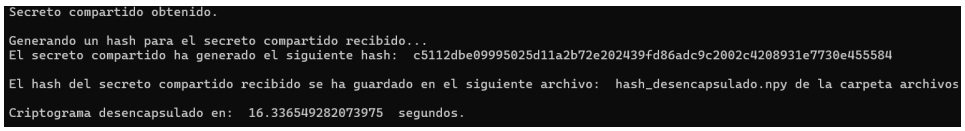

Figura 5.10: Resultado de la tercera desencapsulación con la primera clave

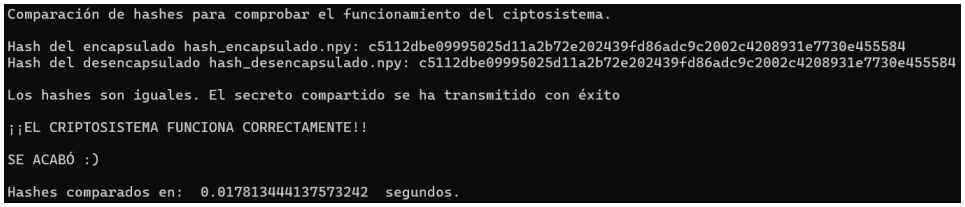

Figura 5.11: Resultado de la comparación del tercer encapsulado-desencapsulado con la primera clave

Como podemos observar en las imágenes de los resultados 1.1 1.2 1.3, las comparaciones de todos los encapsulamientos y desencapsulamientos con primera clave indican que el funcionamiento es correcto.

5.1.2. Segunda clave

Primero, *generamos la segunda clave*. En este caso, los parámetros con los que se construirá la clave serán:

```
Generando la clave de nuestro criptosistema.

Calculando parámetros...


PARÁMETROS:
n:  512 t:  5  p:  2  d:  8  k:  262  m:  13  delta:  13  h:  65 , mu:  8  r:  795388611907908644076362795533
```

Figura 5.12: Parámetros de la clave número 2

Y con estos parámetros obtenemos el siguiente resultado:

```
Tenemos la clave pública, una matriz de dimensiones  250 x {512}

La clave privada y los parámetros del criptosistema se han guardado en el archivo  Clave.npz de la carpeta archivos

La clave pública y los parámetros del criptosistema se han guardado en el archivo  Clave_pub.npz de la carpeta archivos
Clave generada en:  804.5778102874756  segundos.
```

Figura 5.13: Resultado de generar la clave 2

Una vez obtenida la segunda clave, podemos ejecutar la *primera encapsulación y desencapsulación de la segunda clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  560fcd7c13e0e84238b6976d7d2c91cc3aa43f8fba8513a46d726dc4ae02be37

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  3.695436477661133  segundos.
```

Figura 5.14: Resultado de la primera encapsulación con la segunda clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  560fcd7c13e0e84238b6976d7d2c91cc3aa43f8fba8513a46d726dc4ae02be37

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  99.12124347686768  segundos.
```

Figura 5.15: Resultado de la primera desencapsulación con la segunda clave

```
Comparación de hashes para comprobar el funcionamiento del ciptosistema.

Hash del encapsulado hash_encapsulado.npy: 560fcd7c13e0e84238b6976d7d2c91cc3aa43f8fba8513a46d726dc4ae02be37
Hash del desencapsulado hash_desencapsulado.npy: 560fcd7c13e0e84238b6976d7d2c91cc3aa43f8fba8513a46d726dc4ae02be37

Los hashes son iguales. El secreto compartido se ha transmitido con éxito

¡¡EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE!!

SE ACABÓ :)

Hashes comparados en:  0.0149304866790771148  segundos.
```

Figura 5.16: Resultado de la comparación del primer encapsulado-desencapsulado con la segunda clave

Seguimos con la ejecución de la *segunda encapsulación y desencapsulación de la segunda clave*, la cuál nos deja estos resultados:

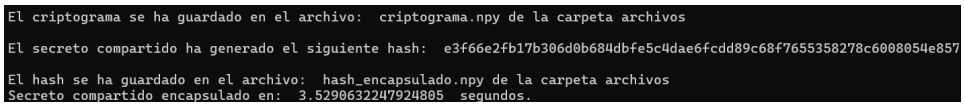

Figura 5.17: Resultado de la segunda encapsulación con la segunda clave

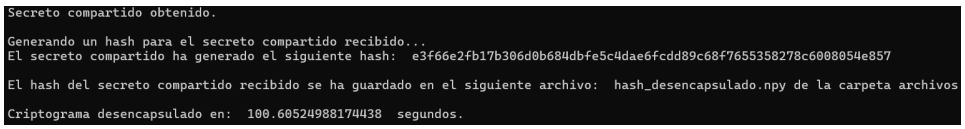

Figura 5.18: Resultado de la segunda desencapsulación con la segunda clave

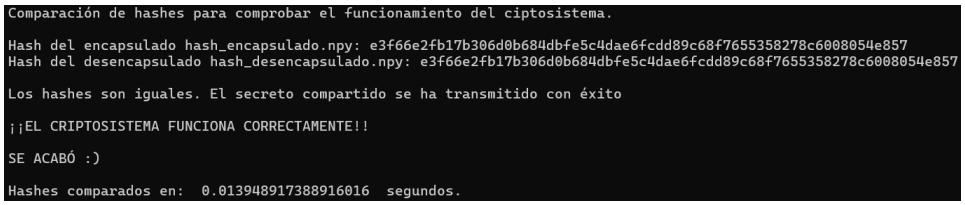

Figura 5.19: Resultado de la comparación del segundo encapsulado-desencapsulado con la segunda clave

Por último, realizamos la *tercera encapsulación y desencapsulación de la segunda clave*, la cuál nos deja estos resultados:

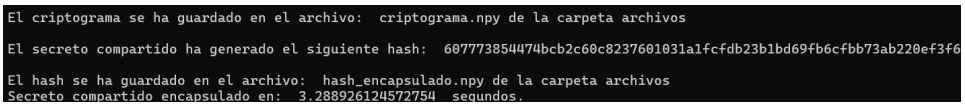

Figura 5.20: Resultado de la tercera encapsulación con la segunda clave

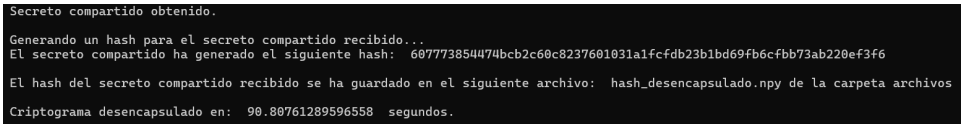

Figura 5.21: Resultado de la tercera desencapsulación con la segunda clave

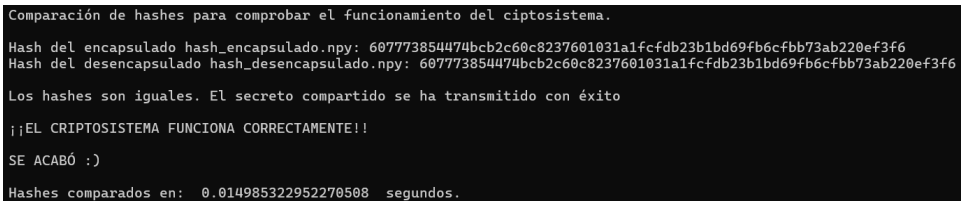

Figura 5.22: Resultado de la comparación del tercer encapsulado-desencapsulado con la segunda clave

Como podemos observar en las imágenes de los resultados 2.1 2.2 2.3, las comparaciones de todos los encapsulamientos y desencapsulamientos con la segunda clave indican que el funcionamiento es correcto.

5.1.3. Tercera clave

Primero, *generamos la tercera clave*. En este caso, los parámetros con los que se construirá la clave serán:

```
Generando la clave de nuestro criptosistema.

Calculando parámetros...


PARÁMETROS:
n:  512 t:  5  p:  2  d:  8  k:  262  m:  12  delta:  16  h:  16 , mu:  6  r:  3106986765265268140923292l7
```

Figura 5.23: Parámetros de la clave número 3

Y con estos parámetros obtenemos el siguiente resultado:

```
Tenemos la clave pública, una matriz de dimensiones  250 x {512}

La clave privada y los parámetros del criptosistema se han guardado en el archivo  Clave.npz de la carpeta archivos

La clave pública y los parámetros del criptosistema se han guardado en el archivo  Clave_pub.npz de la carpeta archivos
Clave generada en:  286.9669244289398  segundos.
```

Figura 5.24: Resultado de generar la clave 3

Una vez obtenida la tercera clave, podemos ejecutar la *primera encapsulación y desencapsulación de la tercera clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  d664a6e3f50090d389b7635d86d8c50927e7090ac93183402d410bc8a0e0973a

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  3.5916519165039062  segundos.
```

Figura 5.25: Resultado de la primera encapsulación con la tercera clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  d664a6e3f50090d389b7635d86d8c50927e7090ac93183402d410bc8a0e0973a

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  30.55164909362793  segundos.
```

Figura 5.26: Resultado de la primera desencapsulación con la tercera clave

```
Comparación de hashes para comprobar el funcionamiento del ciptosistema.

Hash del encapsulado hash_encapsulado.npy: d664a6e3f50090d389b7635d86d8c50927e7090ac93183402d410bc8a0e0973a
Hash del desencapsulado hash_desencapsulado.npy: d664a6e3f50090d389b7635d86d8c50927e7090ac93183402d410bc8a0e0973a

Los hashes son iguales. El secreto compartido se ha transmitido con éxito

¡¡EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE!!

SE ACABÓ :)

Hashes comparados en:  0.021719455718994141  segundos.
```

Figura 5.27: Resultado de la comparación del primer encapsulado-desencapsulado con la tercera clave

Seguimos con la ejecución de la *segunda encapsulación y desencapsulación de la tercera clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  d7020687f41a68227dfd2a60de81a770c3ee849d0c7eec7d322e087e34eda4a3

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  3.4677133560180664  segundos.
```

Figura 5.28: Resultado de la segunda encapsulación con la tercera clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  d7020687f41a68227dfd2a60de81a770c3ee849d0c7eec7d322e087e34eda4a3

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  30.794833183288574  segundos.
```

Figura 5.29: Resultado de la segunda desencapsulación con la tercera clave

```
Comparación de hashes para comprobar el funcionamiento del ciptosistema.

Hash del encapsulado hash_encapsulado.npy: d7020687f41a68227dfd2a60de81a770c3ee849d0c7eec7d322e087e34eda4a3
Hash del desencapsulado hash_desencapsulado.npy: d7020687f41a68227dfd2a60de81a770c3ee849d0c7eec7d322e087e34eda4a3

Los hashes son iguales. El secreto compartido se ha transmitido con éxito

¡¡EL CRIPTOSISTEMA FUNCIONA CORRECTAMENTE!!

SE ACABÓ :)

Hashes comparados en:  0.018471240997314453  segundos.
```

Figura 5.30: Resultado de la comparación del segundo encapsulado-desencapsulado con la tercera clave

Por último, realizamos la *tercera encapsulación y desencapsulación de la tercera clave*, la cuál nos deja estos resultados:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  7e5d170618586e8baa81025c9a0d7957b2ba84bd098dda8d645f3d4a7ba4afe4

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
Secreto compartido encapsulado en:  3.8090803623199463  segundos.
```

Figura 5.31: Resultado de la tercera encapsulación con la tercera clave

```
Secreto compartido obtenido.

Generando un hash para el secreto compartido recibido...
El secreto compartido ha generado el siguiente hash:  7e5d170618586e8baa81025c9a0d7957b2ba84bd098dda8d645f3d4a7ba4afe4

El hash del secreto compartido recibido se ha guardado en el siguiente archivo:  hash_desencapsulado.npy de la carpeta archivos

Criptograma desencapsulado en:  28.48783254623413  segundos.
```

Figura 5.32: Resultado de la tercera desencapsulación con la tercera clave

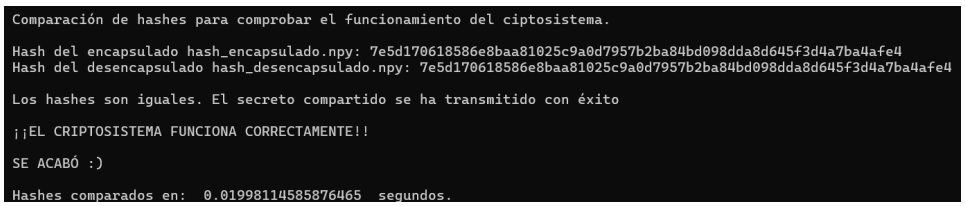

Figura 5.33: Resultado de la comparación del tercer encapsulado-desencapsulado con la tercera clave

Como podemos observar en las imágenes de los resultados 3.1 3.2 3.3, las comparaciones de todos los encapsulamientos y desencapsulamientos con la tercera clave indican que el funcionamiento es correcto.

5.2. Análisis de los resultados

Habiendo realizado varias ejecuciones, tanto de la generación de la clave, como de el encapsulado y desencapsulado de la ejecución de la clave, podemos realizar un análisis de estos resultados para poder sacar algunas conclusiones.

5.2.1. Funcionamiento del sistema

Como hemos podido comprobar en cada ejecución, el resultado es siempre el mismo: el sistema funciona de manera correcta, ya que siempre se obtiene el mensaje original (secreto compartido) desencapsulando el criptograma previamente encapsulando, usando siempre la misma clave para en encapsulado y desencapsulado.

Ya que este comportamiento se repite para cada clave que generemos, podemos afirmar que la *implementación del criptosistema de McEliece/Niederreiter usando códigos skew Goppa es correcta* y por lo tanto el resultado de la realización del proyecto es satisfactorio.

5.2.2. Tiempos de ejecución

En esta segunda parte del análisis de resultados vamos a centrarnos en los tiempos de ejecución que los ejemplos nos han brindado. Las siguientes tablas muestra los *tiempos de ejecución de cada clave*:

- **Primera clave**:

Operación	Ejecución 1	Ejecución 2	Ejecución 3
Generación	138.268s	-	-
Encapsulado	2.822s	2.764s	3.082s
Desencapsulado	15.585s	15.431s	16.336s
Comparación de Hashes	0.008s	0.015s	0.017s

Tabla 5.1: Tiempos de ejecución para Clave 1

- **Segunda clave**:

Operación	Ejecución 1	Ejecución 2	Ejecución 3
Generación	804.577s	-	-
Encapsulado	3.695s	3.529s	3.288s
Desencapsulado	99.121s	100.605s	90.807s
Comparación de Hashes	0.149s	0.013s	0.014s

Tabla 5.2: Tiempos de ejecución para Clave 2

- **Tercera clave**:

Operación	Ejecución 1	Ejecución 2	Ejecución 3
Generación	286.966s	-	-
Encapsulado	3.591s	3.467s	3.809s
Desencapsulado	30.551s	30.794s	28.487s
Comparación de Hashes	0.021s	0.018s	0.019s

Tabla 5.3: Tiempos de ejecución para Clave 3

Se pueden obtener algunas conclusiones mirando solo estos tiempos de ejecución, pero vamos a calcular los *tiempos medios de ejecución* de cada operación de cada clave para poder sacar más conclusiones de ellos:

- **Primera clave**:

  Tiempo medio de encapsulado para la clave 1 =

$$\frac{2.822\,\mathrm{s} + 2.764\,\mathrm{s} + 3.082\,\mathrm{s}}{3} = \frac{8.668\,\mathrm{s}}{3} = 2.889\,\mathrm{s}$$

  Tiempo medio de desencapsulado para la clave 1 =

$$\frac{15.585\,\mathrm{s} + 15.431\,\mathrm{s} + 16.336\,\mathrm{s}}{3} = \frac{47.352\,\mathrm{s}}{3} = 15.784\,\mathrm{s}$$

  Tiempo medio de la comparación de hashes para la clave 1 =

$$\frac{0.008\,\mathrm{s} + 0.015\,\mathrm{s} + 0.017\,\mathrm{s}}{3} = \frac{0.040\,\mathrm{s}}{3} = 0.013\,\mathrm{s}$$

- **Segunda clave**:

  Tiempo medio de encapsulado para la clave 2 =

$$\frac{3.695\,\text{s} + 3.529\,\text{s} + 3.288\,\text{s}}{3} = \frac{10.512\,\text{s}}{3} = 3.511\,\text{s}$$

  Tiempo medio de desencapsulado para la clave 2 =

$$\frac{99.121\,\text{s} + 100.605\,\text{s} + 90.807\,\text{s}}{3} = \frac{290.533\,\text{s}}{3} = 96.511\,\text{s}$$

  Tiempo medio de la comparación de hashes para la clave 2 =

$$\frac{0.149\,\text{s} + 0.013\,\text{s} + 0.014\,\text{s}}{3} = \frac{0.176\,\text{s}}{3} = 0.059\,\text{s}$$

- **Tercera clave**:

  Tiempo medio de encapsulado para Clave 3 =

$$\frac{3.591\,\text{s} + 3.467\,\text{s} + 3.809\,\text{s}}{3} = \frac{10.867\,\text{s}}{3} = 3.622\,\text{s}$$

  Tiempo medio de desencapsulado para la clave 3 =

$$\frac{30.551\,\text{s} + 30.794\,\text{s} + 28.487\,\text{s}}{3} = \frac{89.832\,\text{s}}{3} = 29.611\,\text{s}$$

  Tiempo medio de Comparación de Hashes para Clave 3 =

$$\frac{0.021\,\text{s} + 0.018\,\text{s} + 0.019\,\text{s}}{3} = \frac{0.058\,\text{s}}{3} = 0.019\,\text{s}$$

Recogemos estos tiempos que acabamos de calcular en una nueva tabla para poder ser analizados con más facilidad.

Clave	Generación	Encapsulado (medio)	Desencapsulado (medio)	Comparación de Hashes (medio)
Clave 1	138.268s	2.889s	15.784s	0.013s
Clave 2	804.577s	3.511s	96.511s	0.059s
Clave 3	286.966s	3.622s	29.611s	0.019s

Tabla 5.4: Tiempos medios de ejecución por clave

Teniendo ya todos los tiempos de ejecución necesarios, podemos proceder con el análisis.

Como ya hemos demostrado antes, el sistema funciona correctamente, pero, además, lo hace en un tiempo que nos *permite en un futuro aplicar este criptosistema en aplicaciones reales completamente funcionales* que requieran un cifrado post-cuántico.

Esto se debe a que los tiempos de ejecución son bajos, lo que permitiría el envío y recepción de información encapsulada casi instantánea. Puede parecer que la generación de la clave pueda ser un problema para esto, ya que en uno de los casos tarda más de 10 minutos, pese a que en los otros tarda menos de 5 minutos. Esto en realidad no es un problema, pues se puede generar una sola clave y realizar múltiples envíos seguros de información usando esta clave, sin tener que generar otra en un largo tiempo.

Además, si atendemos a los tiempos medios de ejecución, podemos ver que el tiempo de desencapsulado es directamente proporcional al tiempo que se tarda en generar la clave. Esto puede deberse a que varios de los parámetros y vectores con los que se construye la clave son elegidos aleatoriamente, y en el caso en que estos elementos sean más grandes o tarden más en procesarse en las distintas operaciones que se le aplican puede ocasionar este aumento en el tiempo de ejecución. Otra posible causa de este aumento de tiempo puede ser simplemente que el ordenador donde se ejecuta el programa tenga más carga y tarde más en procesar.

Capítulo 6

Conclusiones

En este último capítulo de la memoria se va a hablar de las conclusiones a las que se han llegado tras la realización del proyecto. Además, se exponen algunas posibles líneas de investigación que se podrían explorar en el futuro a partir de este proyecto.

6.1. Conclusiones del proyecto

El objetivo principal de este proyecto era lograr una implementación del criptosistema de McEliece/Niederreiter usando códigos skew Goppa, para conseguir un criptosistema que sea resistente a ataques de ordenadores cuánticos, los cuales en un futuro serán un problema para la criptografía pues son capaces de vulnerar otros criptosistemas de clave pública como el RSA.

Para la realización de este proyecto primero se hizo un estudio de las herramientas disponibles para realizar la implementación, y se seleccionaron las que se consideraron más adecuadas. Aún así, hemos podido observar que alguna de estas herramientas está un poco limitada para el trabajo específico que queríamos realizar. Un ejemplo de ello es la librería Galois, que nos permite trabajar con cuerpos finitos en python. Esta librería era la más adecuada para la realización de este proyecto, pero está poco optimizada para la realización de varias herramientas necesarias en este proyecto.

Después de seleccionar las herramientas, se estudiaron los diferentes conceptos que era necesario comprender para realizar correctamente el proyecto. Cuando se tenían claros los conceptos, se comenzó con el diseño del sistema, momento en el que se determinaron los requisitos del sistema, así como su arquitectura y la manera en que se iba a validar su funcionamiento.

En el momento en el que se tenía el sistema diseñado, se comenzó con la implementación. Al inicio de la misma se tardó un poco en comprender bien cómo combinar las librerías usadas para lograr un resultado correcto y eficiente. Finalizando la implementación se tuvieron algunos pequeños errores que tardaron en ser encontrados para ser corregidos, y esto retraso un poco el finalizar la implementación del código.

Por último, se comprobó que el sistema funcionaba correctamente, y como se ha demostrado, funciona a la perfección. Se ha conseguido implementar un sistema en python que simula el criptosistema de McEliece/Niederreiter usando códigos skew Goppa en un tiempo de ejecución razonable.

6.2. Futuras líneas de investigación

A partir de este sistema que funciona perfectamente, se puede trabajar en diferentes líneas de investigación para desarrollar nuevas aplicaciones y mejoras en un futuro:

- Implementación de un sistema de mensajería distribuido que permita el envío seguro de información online usando el criptosistema de McEliece/Niederreiter con códigos skew Goppa.
- Optimización de los algoritmos usados en este proyecto y la implementación del sistema realizado en este proyecto con estos algoritmos optimizados
- Implementación de una aplicación completamente funcional que incluya una interfaz gráfica intuitiva para el envío seguro de información
- Búsqueda de nueva funcionalidad para el sistema, logrando por ejemplo poder elegir otro sistema de cifrado.

Bibliografía

[1] INCIBE. ¿sabías que existen distintos tipos de cifrado para proteger la privacidad? https://www.incibe.es/ciudadania/blog/sabias-que-existen-distintos-tipos-de-cifrado-para-proteger-la-privacidad, 2023.

[2] Lucía Fraile de Antonio et al. Criptosistema de mceliece. 2023.

[3] José Gómez-Torrecillas, Francisco Javier Lobillo, and Gabriel Navarro. Skew differential goppa codes and their application to mceliece cryptosystem. *Designs, Codes and Cryptography*, 91(12):3995–4017, 2023.

[4] OTRI Universidad de Granada. Procedimiento y dispositivo de cifrado/descifrado post-cuántico usando códigos lineales. Solicitud de Patente P2024900054 en OEPM Madrid, febrero 2022.

[5] CaixaBank Tech. Ordenadores cuánticos y la seguridad de los algoritmos criptográficos actuales. https://www.caixabanktech.com/es/blogs/ordenadores-cuanticos-y-la-seguridad-de-los-algoritmos-criptograficos-actuales/, 2024.

[6] Daniel Felipe Rambaut Lemus. *Introducción a la Criptografía post-cuántica basada en teoría de códigos*. PhD thesis, Universidad del Rosario Bogota-Colombia, 2021.

[7] GanttProject Developers. Ganttproject - free project management software. https://www.ganttproject.biz/, 2024.

[8] DigiCert. What is public key cryptography?, 2024.

[9] MineryReport. Intercambio de claves diffie-hellman: Seguridad en comunicaciones, 2024.

[10] KeepCoding. ¿qué es rsa en criptografía?, 2024.

[11] Keeper Security. What is elliptic curve cryptography?, 2023. Accessed: 2024-09-05.

[12] Keyfactor. Nist announces round 3 finalists for post-quantum cryptography competition, 2023.

[13] ComputerWorld. Nist publica nuevos borradores de estándares de criptografía postcuántica y llama a su revisión, 2024.

[14] Abhinn Pandey. Python vs c++ vs java: Choosing the right language for your project, 2023.

[15] *SageMath Documentation*, 2024.

[16] Matt Hostetter. Galois: A performant NumPy extension for Galois fields. `https://galois.readthedocs.io/en/v0.4.0/#`, 2020.

[17] Pyfinite: A python library for finite fields. `https://pypi.org/project/pyfinite/`.

[18] NumPy Developers. Numpy - the fundamental package for scientific computing with python. `https://numpy.org/`, 2024.

[19] Python Software Foundation. Documentación de la biblioteca sys en python 3. `https://docs.python.org/es/3/library/sys.html`, 2024.

[20] Python Software Foundation. Documentación de la biblioteca time en python 3. `https://docs.python.org/es/3/library/time.html`, 2024.

[21] Python Software Foundation. Documentación de la biblioteca math en python 3.10. `https://docs.python.org/es/3.10/library/math.html`, 2024.

[22] Python Software Foundation. Documentación de la biblioteca random en python 3. `https://docs.python.org/es/3/library/random.html`, 2024.

[23] Python Software Foundation. Documentación de la biblioteca hashlib en python 3. `https://docs.python.org/es/3/library/hashlib.html`, 2024.

[24] Inc. Amazon Web Services. ¿qué es la criptografía? `https://aws.amazon.com/es/what-is/cryptography`, 2024.

[25] IBM Cloud Education. ¿qué es la criptografía? `https://www.ibm.com/es-es/topics/cryptography`, 2023.

[26] Utimaco. ¿qué es la criptografía postcuántica (pqc)? `https://utimaco.com/es/servicio/base-de-conocimientos/criptografia-postcuantica/que-es-la-criptografia-postcuantica-pqc`, 2023.

[27] Fundación Bankinter. Criptografía post-cuántica: ¿el futuro de la seguridad? https://www.fundacionbankinter.org/noticias/criptografia-post-cuantica/?_adin=02021864894, 2023.

[28] MIT Technology Review. ¿qué es la criptografía poscuántica y por qué se volverá imprescindible? https://www.technologyreview.es/s/11310/que-es-la-criptografia-poscuantica-y-por-que-se-volvera-impresdincible, 2023.

[29] Wolfram Alpha. Campos finitos. https://es.wolframalpha.com/examples/mathematics/algebra/finite-fields, 2024.

[30] VPN Unlimited. Finite field: Qué es y cómo se utiliza en criptografía, 2024.

[31] José Antonio Gómez Saura. Cuerpos finitos. https://www.um.es/documents/118351/1884002/TFG_GOMEZ+SAURA.pdf/32c1e98c-156f-4d7f-ae6f-019100e4b2f7, 2021.

[32] Universidad Politécnica Salesiana. Cuerpos finitos. http://abstract.ups.edu/aata-es/finite.html, 2024.

[33] Javier Gracia Saiz. Introducción a los códigos de corrección de errores, 2023.

[34] Universidad del País Vasco (EHU). Tema 3: Códigos correctores de errores, 2017.

[35] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[36] David Moreno Centeno et al. Criptografía post-cuántica: Análisis de mceliece y una nueva versión con mpc. 2019.

[37] Visual Paradigm. Visual paradigm - herramienta uml online. https://online.visual-paradigm.com/es/diagrams/features/uml-tool/, 2024.

[38] Gonzalo Sanz Guerrero. Criptosistema de mceliece/niederreiter con códigos skew goppa. https://github.com/gsanzguerrero/Criptosistema-de-McEliece-Niederreiter-con-c-digos-skew-Goppa, 2024.

[39] Microsoft. Documentación de visual studio code. https://code.visualstudio.com/docs, 2024.

[40] Python Software Foundation. Python - sitio oficial. https://www.python.org/, 2024.

[41] Python Guide. Guía de instalación de python en linux. `https://python-guide-es.readthedocs.io/es/latest/starting/install3/linux.html`, 2024.

[42] Kinsta. Cómo instalar python. `https://kinsta.com/es/base-de-conocimiento/instalar-python/`, 2024.

[43] Kenneth Reitz and Contributors. Progress - simple and flexible progress bars for python. `https://pypi.org/project/progress/`, 2024.

Apéndice A

Repositorio del Proyecto en GitHub

A.1. Descripción del Repositorio

Este apéndice proporciona información sobre el repositorio de GitHub donde se encuentra alojado el código fuente del proyecto desarrollado para este Trabajo de Fin de Grado. El repositorio contiene todo el código y documentación necesarios para replicar los resultados presentados en esta memoria.

A.2. Enlace al repositorio

El repositorio puede ser accedido en el siguiente enlace:

https://github.com/gsanzguerrero/Criptosistema-de-McEliece-Niederreiter-con-c-digos-skew-Goppa, véase en [38].

A.3. Instrucciones de Acceso y Uso

Para clonar el repositorio, utiliza el siguiente comando de *git* en la carpeta donde quieras clonar el repositorio desde tu terminal:

```
git clone https://github.com/gsanzguerrero/Criptosistema-de-McEliece-
    Niederreiter-con-c-digos-skew-Goppa.git
```

Alternativamente, puedes descargar el código como un archivo `.zip` desde la página del repositorio.

La guía de uso se encuentra en el Apéndice B: Manual de usuario.

A.4. Estructura del Repositorio

El repositorio está organizado de la siguiente manera:

- **README.md**: Archivo con información general del proyecto, instrucciones de instalación y uso.
- **src/**: Carpeta que contiene el código fuente del proyecto. Aquí podrás encontrar los 4 programas de los que se compone este proyecto:
  - generar_clave.py: este programa es el encargado de generar la clave pública y los parámetros del criptosistema.
  - encapsular.py: este programa recibe la clave pública generada y construye un secreto compartido aleatorio y lo encapsula en un criptograma para ser enviado. Además, genera un hash del secreto compartido que posteriormente será utilizado para comprobar el funcionamiento del criptosistema.
  - desencapsular.py: este programa recibe el criptograma y la clave y desencapsula el criptograma para obtener el secreto compartido. También crea un hash del criptograma desencapsulado para testear el funcionamiento del criptosistema.
  - comprobar_hashes.py: este programa recibe los dos hashes generados al encapsular y desencapsular y los compara para comprobar si el criptosistema funciona correctamente o no.
- **.gitignore**: Este archivo indica qué elementos ignorar a la hora de actualizar contenido en repositorio al hacer un commit.
- **Memoria.pdf**: este documento es la memoria del proyecto, que incluye una introducción, el estado del arte, el fundamento teórico, cómo fue el proceso de diseño y la implementación del código, resultados y conclusiones, junto con bibliografía y otros apéndices, los cuales incluyen un enlace al repositorio, un manual de usuario e instrucciones de instalación tanto del entorno de desarrollo como de las librerías necesarias.

Apéndice B

Manual de usuario

En este apéndice se explica cómo ejecutar este criptosistema de McEliece/Niederreiter con códigos skew Goppa implementado en Python.

Primero, se debe descargar el código del proyecto que está alojado en el siguiente repositorio de GitHub, véase [38]. Para ello, debemos descargarlo tal y como se indica en el Apéndice A: Repositorio del proyecto en GitHub.

Una vez descargado el código, debemos crear una carpeta llamada archivos en la raíz del proyecto, para que la estructura de carpetas del proyecto quede de la siguiente manera:

```
proyecto/
|-- archivos/
|-- src/
|   |-- generar_clave.py
|   |-- encapsular.py
|   |-- desencapsular.py
|   `-- comparar_hashes.py
|-- .gitignore
|-- README.md
`-- memoria.pdf
```

Esta carpeta es necesaria, ya que los archivos que se generan con la ejecución de los diferentes programas que componen el sistema se guardarán aquí.

Ahora que ya tenemos el código y las carpetas del proyecto con la estructura necesaria, podemos comenzar la ejecución del sistema. Para ejecutar este sistema debemos seguir estos pasos:

1. **Generación de las claves**: para generar las claves debemos acceder a la carpeta **/src** donde se encuentran los archivos de python. Una vez en la carpeta **/src**, ejecutamos el comando python con el archivo generar_clave.py:

```
cd src

python.exe generar_clave.py
```

   Vemos en la salida el nombre de los archivos donde se ha guardado las claves. El programa directamente guarda la clave en un archivo en la carpeta archivos:

```
Tenemos la clave pública, una matriz de dimensiones  250 x {512}

La clave privada y los parámetros del criptosistema se han guardado en el archivo  Clave.npz de la carpeta archivos

La clave pública y los parámetros del criptosistema se han guardado en el archivo  Clave_pub.npz de la carpeta archivos
Clave generada en:  457.7966203689575  segundos.
```

Figura B.1: Salida del programa generar_clave.py

2. **Encapsulado**: para generar el secreto compartido y encapsularlo debemos ejecutar el comando python junto con el nombre del archivo a ejecutar, **encapsular.py** junto con el nombre del archivo donde está la clave pública, que se pasa como argumento. No hace falta poner la carpeta **/archivos**, ya que el programa la incorpora directamente a la hora de buscar el archivo:

```
python.exe encapsular.py Clave_pub.npz
```

   En la salida de este programa podemos ver en qué archivos se han guardado tanto el criptograma como el hash generado del secreto compartido:

```
El criptograma se ha guardado en el archivo:  criptograma.npy de la carpeta archivos

El secreto compartido ha generado el siguiente hash:  9597eac619badc2640ec0f3adf1590cdebf3623bd7130c8a5afaf7ebbd1fae0f

El hash se ha guardado en el archivo:  hash_encapsulado.npy de la carpeta archivos
```

Figura B.2: Salida del programa encapsular.py

3. **Desencapsulado**: para desencapsular el secreto compartido y obtener el secreto compartido, debemos ejecutar el comando python junto con el nombre del archivo a ejecutar, **desencapsular.py** junto con el nombre del archivo donde está la clave privada y el nombre del archivo donde está guardado el criptograma, que se pasan como argumento. De igual manera que al encapsular, no hace falta poner la carpeta **/archivos**, ya que el programa la incorpora directamente a la hora de buscar el archivo:

```
python.exe desencapsular.py Clave.npz criptograma.npy
```

   En la salida de este programa podemos ver en qué archivo se ha guardado el hash generado del criptograma desencapsulado:

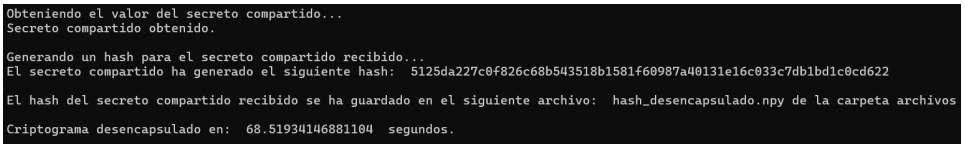

Figura B.3: Salida del programa desencapsular.py

4. **Comprobación de funcionamiento**: para comprobar el correcto funcionamiento del encapsulado y desencapsulado, debemos ejecutar el comando python junto con el nombre del archivo a ejecutar, **comparar_hashes.py** junto con el nombre del archivo donde están los hashes del secreto compartido original y del criptograma desencapsulado , que se pasan como argumento. De igual manera que en casos anteriores, no hace falta poner la carpeta **/archivos**, ya que el programa la incorpora directamente a la hora de buscar el archivo:

```
python.exe comparar_hashes.py hash_encapsulado.npy
    hash_desencapsulado.npy
```

   En la salida de este último programa podemos ver que el criptosistema funciona correctamente:

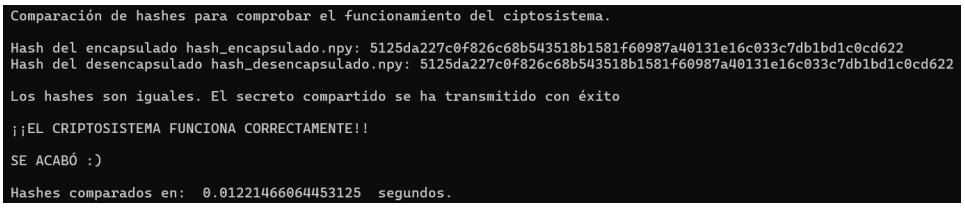

Figura B.4: Salida del programa comparar_hashes.py

Apéndice C

Instalación del Entorno de Desarrollo

Este apéndice proporciona instrucciones sobre cómo instalar y configurar el entorno de desarrollo necesario para ejecutar del proyecto. El entorno de desarrollo del proyecto incluye Visual Studio Code como editor de código y Python como lenguaje de programación. Las instrucciones cubren los tres principales sistemas operativos: Windows, Ubuntu y macOS.

C.1. Visual Studio Code

Visual Studio Code es un editor de código fuente ligero y potente, ideal para el desarrollo en Python.

C.1.1. Windows

Para instalar Visual Studio Code en Windows:

- Descarga el instalador desde la página oficial de Visual Studio Code.
- Ejecuta el archivo descargado y sigue las instrucciones del asistente de instalación.
- Una vez instalado, abre Visual Studio Code y instala la extensión de Python desde el Marketplace de Visual Studio Code.

Para más detalles, consulta el tutorial de instalación en Visual Studio Code en Windows.

C.1.2. Ubuntu

Para instalar Visual Studio Code en Ubuntu:

- Abre una terminal y ejecuta los siguientes comandos:

```
$ sudo apt-get install wget gpg
$ wget -qO- https://packages.microsoft.com/keys/microsoft
    .asc | gpg --dearmor > packages.microsoft.gpg
$ sudo install -D -o root -g root -m 644 packages.
    microsoft.gpg /etc/apt/keyrings/packages.microsoft.
    gpg
$ echo "deb [arch=amd64,arm64,armhf signed-by=/etc/apt/
    keyrings/packages.microsoft.gpg] https://packages.
    microsoft.com/repos/code stable main" |sudo tee /etc/
    apt/sources.list.d/vscode.list > /dev/null
$ rm -f packages.microsoft.gpg
```

- Después, actualiza la caché de paquetes e instala el paquete desde la terminal:

```
$ sudo apt install apt-transport-https
$ sudo apt update
$ sudo apt install code
```

- Abre Visual Studio Code e instala la extensión de Python desde el Marketplace.

Para más detalles, consulta el tutorial de instalación en Visual Studio Code en Ubuntu.

C.1.3. macOS

Para instalar Visual Studio Code en macOS:

- Descarga el archivo '.dmg' desde la página oficial de Visual Studio Code.
- Abre el archivo descargado y arrastra el icono de Visual Studio Code a la carpeta de Aplicaciones.
- Abre Visual Studio Code y instala la extensión de Python desde el Marketplace.

Para más detalles, consulta el tutorial de instalación en Visual Studio Code en macOS.
Véase en [39].

C.2. Python

Python es el lenguaje de programación utilizado en el desarrollo del proyecto.

C.2.1. Windows

Para instalar Python en Windows:

- Descarga el instalador desde la página oficial de Python.
- Ejecuta el archivo descargado y asegúrate de marcar la opción **Add Python to PATH** antes de hacer clic en **Install Now**.
- Verifica la instalación abriendo una terminal (cmd) y ejecutando:

```
python --version
```

Consulta el tutorial de instalación en Python en Windows.

C.2.2. Ubuntu

Para instalar Python en Ubuntu:

- Puedes ver si tienes alguna versión de python preinstalada:

```
$ python3 --version
```

- Abre una terminal y ejecuta el siguiente comando:

```
$ sudo apt-get update
$ sudo apt-get install python3.6
```

- Verifica la instalación ejecutando:

```
$ python3 --version
```

Consulte el tutorial de instalación en Python en Ubuntu, véase [41].

C.2.3. macOS

MacOS ya viene con la versión 2.7 de python preinstalada. Esta versión está obsoleta, por lo que es recomendable descargar e instalar la última versión estable (3.x).

Para instalar Python en macOS:

- Descarga el instalador desde la página oficial de Python.
- Ejecuta el instalador y sigue las instrucciones
- Verifica la instalación ejecutando:

```
python3 --version
```

Consulte el tutorial de instalación de Python en macOS, véase [42] [40].

Apéndice D

Instalación de las librerías necesarias

Este apéndice indica cómo instalar las diferentes librerías que se usan durante la ejecución del sistema. En este proyecto, se utilizan varias librerías de Python. A continuación, se detallan las instrucciones para instalar cada una de ellas, según el sistema operativo que utilices.

D.1. Librerías preinstaladas

Las siguientes librerías vienen preinstaladas con la distribución estándar de Python, por lo que no es necesario instalarlas manualmente:

- math: esta librería permite trabajar con diferentes funciones matemáticas. Véase [21],
- sys: este módulo da acceso a variables usadas o mantenidas por el intérprete y a funciones que interactúan fuertemente con el intérprete. Véase [19].
- time: esta librería permite trabajar con el tiempo. En este caso utilizamos cronómetros. Véase [20].
- random: este módulo permite generar números aleatorios y seleccionar elementos aleatorios de un conjunto. Véase [22].
- hashlib: usando esta librería podemos trabajar con diferentes algoritmos de hashing y cifrado. Véase [23].

Estas librerías están disponibles en cualquier sistema operativo donde se haya instalado Python.

D.2. Librerías a Instalar

Las siguientes librerías no vienen preinstaladas con Python y deben instalarse por separado:

- numpy: la biblioteca numpy permite trabajar con objetos multidimensionales, por lo que facilita la computación científica. Véase [18].
- galois: esta librería extiende la librería numpy de python para poder trabajar con cuerpos finitos. Véase [16].
- progress: este módulo permite mostrar el progreso de ejecución del programa permitiendo mostrar por pantalla barras de progreso. Véase [43].

Instrucciones de Instalación

A continuación, se presentan las instrucciones para instalar las librerías en los sistemas operativos más comunes: Windows, Ubuntu y macOS.

- **Windows:**
  1. Abre el símbolo del sistema (Cmd) o PowerShell.
  2. Asegúrate de tener `pip` instalado. Puedes verificarlo con el comando:

     ```
     python -m ensurepip --upgrade
     ```

  3. Instala las librerías utilizando el siguiente comando:

     ```
     pip install galois numpy progress
     ```

- **Ubuntu:**
  1. Abre la terminal.
  2. Asegúrate de tener `pip` instalado. Puedes instalarlo con:

     ```
     sudo apt-get install python3-pip
     ```

  3. Instala las librerías utilizando el siguiente comando:

     ```
     pip3 install galois numpy progress
     ```

- **macOS:**

  1. Abre la terminal.
  2. Asegúrate de tener `pip` instalado. Puedes instalarlo con:

```
sudo easy_install pip
```

  3. Instala las librerías utilizando el siguiente comando:

```
pip install galois numpy progress
```

Verificación de Instalación

Para verificar que las librerías se han instalado correctamente, abre una consola de Python e intenta importar cada una de ellas:

```
1 import galois
2 import numpy as np
3 from progress.bar import Bar
```

Si no aparece ningún error, las librerías se han instalado correctamente.