UNIVERSIDAD
DE GRANADA

Facultad de Ciencias y Escuela Superior de Ingeniería Informática y Matemáticas

Grado en Matemáticas e Ingeniería Informática

TRABAJO DE FIN DE GRADO

Implementación y criptoanálisis del criptosistema GLN

Presentado por:
Carmen Azorín Martí

Curso académico 2025-2026

Implementación y criptoanálisis del criptosistema GLN

Carmen Azorín Martí

Carmen Azorín Martí *Implementación y criptoanálisis del criptosistema GLN*.
Trabajo de fin de Grado. Curso académico 2025-2026.

Responsable de tutorización

Gabriel Navarro Garulo
Álgebra y Teoría de la Información

Grado en Matemáticas e Ingeniería Informática

Facultad de Ciencias y Escuela Superior de Ingeniería Informática y Matemáticas

Universidad de Granada

Declaración de originalidad

D./Dña. Carmen Azorín Martí

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2025-2026, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 23 de noviembre de 2025

Fdo: Carmen Azorín Martí

Agradecimientos

Quiero empezar agradeciendo a mi familia, que ha estado a mi lado durante todos estos años de carrera, y los he sentido cerca aún estando a distancia. Siempre han confiado en mi esfuerzo y en mis capacidades, y su apoyo incondicional ha sido fundamental.

También quiero agradecer a todos los amigos que he hecho en el camino. Sin ellos, no podría haber sacado esta carrera adelante. Nos hemos apoyado mutuamente en todo momento, hemos estudiado juntos, compartido risas, lágrimas y nos hemos alegrado por cada pequeño logro conseguido. Especialmente, a María y a Laura, quienes siempre han conseguido sacarme una sonrisa, independientemente de si el día era bueno o malo. Gracias a ellas, esta experiencia ha sido mucho más enriquecedora y llena de bonitos recuerdos.

Quisiera hacer también una mención a una persona que estuvo a mi lado en casi todo momento, apoyándome tanto en los estudios como en lo personal y que ha dejado una huella que siempre recordaré con mucho cariño.

Y por último, gracias a todas las personas que he conocido en estos años y que han formado parte de mi vida en Granada.

Resumen

Descripción del problema

En el contexto actual de la seguridad de la información, la criptografía clásica se apoya en problemas matemáticos que se consideran intratables para los ordenadores convencionales, como la factorización entera o el problema del logaritmo discreto. Sin embargo, el posible desarrollo de computadores cuántico haría vulnerables muchos de los criptosistemas estandarizados basados en dichos problemas, como RSA o Diffie–Hellman, lo que plantea la necesidad de estudiar alternativas resistentes frente a adversarios cuánticos.

Este Trabajo Fin de Grado se centra en el estudio y la implementación del criptosistema GLN, un esquema de clave pública basado en el problema de la suma de subconjuntos (Subset Sum Problem, SSP) y, más concretamente, en el WMSSP. El objetivo principal es analizar la construcción matemática de GLN, implementar el criptosistema de forma modular y eficiente, y evaluar su seguridad frente a distintos tipos de ataques, tanto clásicos (fuerza bruta o recuperación de clave privada) como basados en retículos (ataques de baja densidad mediante LLL).

Contexto matemático

Para contextualizar el trabajo, se comienzan presentando los fundamentos teóricos de la computación: las máquinas de Turing como modelo formal de algoritmo y el concepto de complejidad temporal como número de pasos en función del tamaño de la entrada. En este marco se introduce el problema de la suma de subconjuntos, SSP, que consiste en decidir si existe un subconjunto de una familia de enteros que sume un valor dado. Se explica que SSP es NP-difícil y no se conoce ningún algoritmo que lo resuelva en tiempo polinomial en el caso general, lo que lo convierte en un candidato natural para construir criptosistemas.

A continuación se introduce la criptografía postcuántica. Se repasan brevemente los primeros criptosistemas clásicos, de estructura muy simple y fácilmente atacables, y se distingue entre criptosistemas de clave simétrica y de clave pública. El criptosistema GLN pertenece a esta segunda familia: existe una clave pública con la que el emisor cifra los mensajes y una clave privada, conocida únicamente por el receptor esperado, con la que se realiza el descifrado.

Se presentan las principales familias de criptografía postcuántica relevantes para el trabajo. Por un lado, la criptografía basada en códigos, con el ejemplo del sistema de McEliece, en la que la clave pública es una versión disfrazada de un código corrector de errores, obtenida mediante transformaciones lineales. GLN se inspira en esta idea: en lugar de ocultar un código, oculta una instancia estructurada de SSP transformándola en una instancia de tipo WMSSP mediante operaciones aritméticas modulares.

Por otro lado, se revisan los esquemas de tipo mochila clásicos como Merkle–Hellman, que también se basan en variantes de SSP pero que fueron debilitados por ataques de baja densidad. Estos ataques motivan el estudio de la teoría de retículos y de algoritmos de

reducción de base, dado que muchas instancias de SSP pueden transformarse en instancias del Shortest Vector Problem (SVP) en un retículo apropiado.

En este contexto se introducen los conceptos de retículo, bases, volumen, mínimos sucesivos y el problema del vector más corto. Se desarrolla la ortogonalización de Gram–Schmidt y se describe el algoritmo de Lenstra–Lenstra–Lovász (LLL), que permite obtener bases reducidas y constituye la herramienta central en los ataques de baja densidad contra criptosistemas de tipo mochila. Este contexto matemático proporciona el lenguaje necesario para entender tanto el diseño de GLN como los ataques a los que puede verse sometido.

Finalmente, se describe con detalle la construcción del criptosistema GLN: motivación algebraica, algoritmos de generación de claves, cifrado y descifrado, y se justifican rigurosamente sus propiedades de corrección. En particular, se demuestra que, bajo las condiciones impuestas sobre los parámetros, el mensaje descifrado existe, es único y coincide con el mensaje original.

Implementación y experimentación del criptosistema

La implementación de GLN se ha diseñado siguiendo una arquitectura modular y extensible, de forma que cada componente (generación de claves, cifrado, descifrado, generación de parámetros y ataques) se encapsula en módulos independientes que pueden modificarse o reemplazarse sin afectar al resto del sistema.

Dado que el criptosistema trabaja con enteros de tamaño muy grande, se ha desarrollado una clase específica para representar y operar con estos enteros, que sirve de base para todos los algoritmos aritméticos implementados. Sobre esta representación se implementan algoritmos fundamentales como el algoritmo de Euclides extendido truncado, la exponenciación modular eficiente y un generador de números aleatorios adaptado a este tipo de objetos. Una decisión clave de diseño es la verificación de primalidad de grandes enteros, para lo cual se implementa y explica el test de Miller–Rabin como prueba de primalidad probabilística.

Además de las fases básicas de generación de claves, cifrado y descifrado, se implementa el llamado ataque por tríos, un ataque clásico a la clave privada que explora combinaciones de posibles parámetros (como factores primos y estructuras internas de la clave) a partir de la información de la clave pública. También se implementan funciones auxiliares para la generación de parámetros que deben satisfacer ciertas condiciones aritméticas, necesarias para garantizar la corrección matemática del esquema.

En cuanto a la evaluación experimental, se han definido y utilizado varias métricas de seguridad y rendimiento. Por un lado, la seguridad medida en bits frente a ataques por fuerza bruta y por tríos. Por otro lado, los tiempos de ejecución de cada fase del criptosistema (generación de claves, cifrado, descifrado), el tamaño de la clave pública y el cálculo de la densidad de las instancias asociadas al problema SSP, relevante para los ataques de baja densidad basados en retículos.

Se describe cómo ejecutar el código desarrollado y las distintas salidas que puede producir (claves generadas, mensajes cifrados y descifrados, tiempos de ejecución, estimaciones de seguridad y resultados de ataques). A partir de estos experimentos, se proponen parámetros candidatos considerados seguros frente al ataque por tríos, controlando a la vez que el tamaño de la clave pública no crezca de forma inasumible y que los tiempos de ejecución sean razonables. Se identifican configuraciones que maximizan la seguridad y minimizan el coste temporal.

Finalmente, se analizan estos parámetros frente a ataques de baja densidad basados en

LLL, implementados con la ayuda de SageMath. Los experimentos confirman que el éxito de dichos ataques depende mayormente de la densidad de la instancia del problema, por lo que se identifican rangos de densidad que deben evitarse para que el sistema no sea atacado con relativa facilidad. La discusión se completa teniendo en cuenta simultáneamente los tres tipos de ataque considerados (tríos, LLL y fuerza bruta), y se recomiendan parámetros que ofrecen un compromiso razonable entre seguridad y eficiencia para el criptosistema GLN.

Summary

Problem description

In the current context of information security, classical cryptography relies on mathematical problems that are believed to be intractable for conventional computers, such as integer factorization or the discrete logarithm problem. However, the potential development of large-scale quantum computers would render many standardized cryptosystems based on these problems vulnerable, including RSA and Diffie–Hellman, which creates an urgent need to study alternatives that remain secure against quantum adversaries.

This Bachelor's Thesis focuses on the study and implementation of the GLN cryptosystem, a public-key scheme based on the Subset Sum Problem (SSP) and, more specifically, on certain knapsack-type variants. The main objective is to analyze the mathematical construction of GLN, implement the cryptosystem in a modular and efficient way, and evaluate its security against different kinds of attacks, both classical (brute force, private key recovery) and lattice-based (low-density attacks using LLL).

Mathematical background

To rigorously frame the work, we first present the theoretical foundations of computation: Turing machines as a formal model of algorithms, and the notion of time complexity as the number of steps as a function of input size. Within this framework, we introduce the Subset Sum Problem (SSP), which asks whether there exists a subset of a given set of integers that sums to a prescribed value. It is explained that SSP is NP-hard and that no polynomial-time algorithm is known for solving it in the general case, which makes it a natural candidate for building cryptosystems.

We then introduce post-quantum cryptography. We briefly review the earliest classical cryptosystems, which had very simple structures and could be broken relatively easily, and we distinguish between symmetric-key and public-key cryptosystems. The GLN cryptosystem belongs to the latter family: there is a public key used by the sender to encrypt messages, and a private key, known only to the legitimate receiver, which is used for decryption.

The main families of post-quantum cryptography relevant to this work are presented next. On the one hand, code-based cryptography, exemplified by the McEliece system, in which the public key is a "disguised" version of an error-correcting code, obtained via linear transformations. GLN is inspired by this idea: instead of hiding a code, it hides a structured instance of SSP by transforming it into a weighted SSP instance using modular arithmetic operations.

On the other hand, we review classical knapsack schemes such as Merkle–Hellman, which are also based on SSP variants but were weakened by low-density attacks and other cryptanalytic techniques. These attacks motivate the study of lattice theory and basis reduction algorithms, since many instances of SSP can be transformed into instances of the Shortest Vector Problem (SVP) in a suitable lattice.

In this context, we introduce the concepts of lattice, basis, volume, successive minima, and the shortest vector problem. We develop Gram–Schmidt orthogonalization and describe the Lenstra–Lenstra–Lovász (LLL) algorithm, which produces reduced bases and is the central tool in low-density attacks on knapsack-type cryptosystems. This mathematical background provides the language needed to understand both the design of GLN and the attacks to which it may be exposed.

Finally, we describe in detail the construction of the GLN cryptosystem: its algebraic motivation, the algorithms for key generation, encryption, and decryption, and a rigorous justification of its correctness. In particular, we prove that, under the imposed parameter conditions, the decrypted message exists, is unique, and coincides with the original plaintext.

Implementation and experimental evaluation of the cryptosystem

The implementation of GLN has been designed following a modular and extensible architecture, so that each component (key generation, encryption, decryption, parameter generation, and attacks) is encapsulated in an independent module that can be modified or replaced without affecting the rest of the system.

Since the cryptosystem operates on very large integers, a dedicated class has been developed to represent and manipulate such integers, which serves as the foundation for all arithmetic algorithms used. On top of this representation, fundamental algorithms are implemented, such as the extended Euclidean algorithm, efficient modular exponentiation, and a random number generator adapted to these large-integer objects. A key design decision is the primality testing of large integers, for which the Miller–Rabin probabilistic primality test is implemented and explained, ensuring a reasonable balance between efficiency and reliability.

In addition to the basic key generation, encryption, and decryption phases, the so-called trio attack is implemented, a classical private-key attack that explores combinations of possible parameters (such as prime factors and internal key structure) derived from the public key. Auxiliary functions are also implemented for parameter generation, enforcing specific arithmetic conditions required to guarantee the mathematical correctness of the scheme.

For the experimental evaluation, several security and performance metrics are defined and used. On the one hand, security in bits against brute-force attacks, which serves as a reference for the trio attack. On the other hand, the execution times of each phase of the cryptosystem (key generation, encryption, decryption) and the density of the SSP instances associated with the scheme, which is relevant for low-density lattice-based attacks.

We describe how to run the developed code and the different outputs it can produce (generated keys, encrypted and decrypted messages, execution times, security estimates, and attack results). Based on these experiments, we propose candidate parameter sets considered secure against the trio attack, while simultaneously ensuring that the size of the public key does not grow excessively and that execution times remain acceptable. Using Pareto frontiers, we identify configurations that maximize security (in terms of brute-force bits and resistance to the trio attack) while minimizing computational cost.

Finally, these parameters are analyzed against low-density attacks based on LLL, implemented with the help of SageMath. The experiments confirm that the success of such attacks critically depends on the density of the problem instance, so ranges of density are identified that must be avoided to prevent the system from being broken too easily. The discussion

is completed by simultaneously taking into account the three types of attacks considered (trios, LLL, and brute force), and recommendations are formulated for parameter choices that offer a reasonable trade-off between security and efficiency for the GLN cryptosystem.

Índice general

Introducción

1. Contexto

La seguridad de la información es uno de los pilares fundamentales en la sociedad digital contemporánea. Las comunicaciones personales, las transacciones financieros, incluso los servicios gubernamentales dependen de la criptografía para garantizar la confidencialidad, integridad y autenticidad de los datos. Durante las últimas décadas, los sistemas criptográficos más utilizados (como RSA y la criptografía de Curva Elíptica ECC) han proporcionado una base sólida de seguridad, apoyada en problemas matemáticos cuya resolución se considera intratable con los medios computacionales clásicos.

Sin embargo, el avance de la computación cuántica ha puesto en riesgo esta situación. La existencia de algoritmos cuánticos, como el algoritmo de Shor, capaces de factorizar números enteros o resolver logaritmos discretos en tiempo polinómico, compromete la seguridad de los sistemas actuales. Esto ha impulsado el desarrollo de la criptografía post-cuántica, que busca diseñar sistemas resistentes tanto a ataques clásicos como cuánticos. Las propuestas más prometedoras se basan en problemas matemáticos de alta complejidad. Entre ellos, el problema de la mochila o suma de subconjuntos (SSP).

El criptosistema GLN pertenece a esta familia de problemas. Se trata de un esquema de cifrado asimétrico inspirado en la dificultad del SSP, un problema NP-completo que consiste en determinar si existe un subconjunto de números que sume un valor objetivo. Dado que no se conoce un algoritmo eficiente que lo resuelva en general, incluso para ordenadores cuánticos, el GLN es un posible alternativa viable en el contexto post-cuántico. Sin embargo, como ocurre con muchas propuestas teóricas su eficacia y resistencia práctica deben ser analizadas de manera empírica.

En este trabajo se presenta la implementación y evaluación del criptosistema GLN, con el objetivo de comprobar su viabilidad práctica, analizar su rendimiento y estudiar su vulnerabilidad frente a ataque combinatorios y de retículos. La combinación de fundamentos teóricos, experimentación computacional y análisis de resultados permite obtener una visión completa del sistema criptográfico y de su potencial como sistema segura en el contexto post-cuántico.

2. Motivación

El desarrollo de la computación cuántica representa uno de los mayores desafíos en la historia de la criptografía moderna. Las agencias de estandarización, como el National Institute of Standards and Technology (NIST), han iniciado procesos de selección de algoritmos post-cuánticos para definir los futuros estándares de seguridad digital. En este proceso se evalúan nuevas propuestas que combinan robustez teórica, eficiencia práctica y resistencia demostrable frente a ataques conocidos.

El criptosistema GLN surge como una propuesta alternativa al esquema clásico de Merkle-Hellman, mejorando sus puntos débiles mediante la incorporación de transformaciones alge-

braicas y parámetros de alta densidad. Sin embargo, su comportamiento real y la resistencia a ataques específicos son análisis necesarios para saber si puede ser considerado una opción viable dentro de la criptografía post-cuántica.

Además de la relevancia criptográfica, este trabajo tiene motivación didáctica y experimental: construir desde cero un sistema de cifrado asimétrico completo en C++, abarcando desde la generación de claves hasta la decodificación, y sometiéndolo a pruebas de rendimiento. de esta forma, se busca unir el rigor teórico de la computación con la validación empírica del comportamiento del sistema.

3. Propuesta

Este Trabajo Fin de Grado propone la implementación integral del criptosistema GLN y su análisis empírico mediante experimentación controlada. La propuesta combina tres ejes fundamentales:

- Fundamentación teórica: Se revisan los conceptos esenciales de la teoría de la computación, como las máquinas de Turing, la complejidad temporal y la NP-completitud, para contextualizar la dificultad del problema de la suma de subconjuntos. También se repasan los principales tipos de sistema criptográficos (simétricos, asimétricos y post-cuánticos), destacando sus diferencias, ventajas y limitaciones.
- Implementación práctica: Se desarrolla una implementación modular en C++ del criptosistema GLN, que incluye los algoritmos de generación de claves, cifrado, descifrado y ataques a la clave privada. La implementación se complementa con utilidades matemáticas propias, como el manejo de enteros grandes y generación determinista de primos.
- Evaluación y criptoanálisis: Se realiza un estudio experimental variando parámetros del sistema (tamaño del mensaje, densidad, rango de primos, etc.) con el objetivo de medir tiempos de ejecución, tamaño de claves y resistencia frente a ataques conocidos. En particular, se analiza los ataques combinatorios por tríos y por fuerza bruta y el ataque por reducción de retículos (LLL).

4. Estructura del trabajo

El contenido del trabajo se organiza en varias partes:

1. Capítulo 1: Presenta los fundamentos teóricos de la computación, incluyendo máquinas de Turing, los conceptos de decidibilidad y complejidad, y la clasificación de problemas según su dificultad computacional. Además, se presenta el problema matemático base del criptosistema: el problema de la suma de subconjuntos.
2. Capítulo 2: Revisa los principios básicos de la criptografía clásica moderna, incluyendo los tipos de cifrado y las propiedades que definen un sistema seguro. Además, introduce la criptografía postcuántica, sus objetivos, principales familias y los problemas matemáticos sobre los que se apoyan.
3. Capítulo 3: Describe en detalle el criptosistema GLN, su diseño, funcionamiento y los algoritmos asociados. Además, se presenta la base matemática de manera sólida, demostrando la corrección del criptosistema.

4. Capítulo 4: Se presentan los ataques conocidos a la familia de esquemas basados en mochila, como el criptosistema GLN. En concreto, se analizan el ataque a la clave privada (ataque por tríos), el ataque por fuerza bruta y el ataque por reducción a problemas de retículos usando el algoritmo LLL.

5. Capítulo 5: Explica la implementación en C++, detallando los módulos desarrollados, la estructura del código y las estrategias usadas para optimizar su rendimiento. Además, presenta cuáles son las métricas utilizadas en los experimentos y cómo se calculan.

6. Capítulo 6: Se analizan los resultados obtenidos tras la ejecución de los experimentos. Se analizan los parámetros influyentes en cada uno de los ataques: por tríos, por fuerza bruta y por reducción de retículos. También se analizan los costes computacionales de cada una de las fases del criptosistema: generación de claves, cifrado y descifrado. Y también se mide el tamaño de la clave pública con respecto a las distintas configuraciones de parámetros. Con estos análisis, se realiza una conclusión final sobre cuáles son los parámetros seguros y computacionalmente eficientes.

5. Objetivos

El objetivo general del proyecto es implementar y analizar el criptosistema GLN con el fin de evaluar su viabilidad práctica y su seguridad frente a ataques conocidos.

Los objetivos específicos son:

- Revisar los fundamentos teóricos de la computación y la criptografía necesarios para entender el sistema GLN.
- Diseñar e implementar el esquema GLN completo en C++, aplicando buenas prácticas de ingeniería de software.
- Analizar empíricamente el rendimiento del sistema en función de distintos parámetros estructurales.
- Evaluar su resistencia práctica frente a los ataques conocidos y estudiados y comparar los resultados con la complejidad teórica esperada.

1. Fundamentos teóricos de la computación

1.1. Máquinas de Turing

En 1936, Alan Turing propuso un modelo matemático que desde entonces se considera un fundamento teórico de la computación: la máquina de Turing [29]. Para entender qué es una máquina de Turing, necesitamos formalizar algunos conceptos básicos. Este capítulo recopila definiciones y teoremas mayormente seguidos en [27] y [15].

Definición 1.1. Sea Σ un alfabeto. Llamamos lenguaje sobre Σ a cualquier subconjunto de Σ^*.

Un alfabeto es un conjunto finito y no vacío de símbolos. Una cadena (o palabra) sobre Σ es una secuencia finita de símbolos de Σ.

Ejemplo 1.2. Algunos ejemplos de alfabetos son:

- $\Sigma_1 = \{0, 1\}$
- $\Sigma_2 = \{a, b, c, d, e, ..., z\}$
- $\Sigma_3 = \{*, @, \%\}$

Algunos ejemplos de lenguajes son:

- El lenguaje sobre el alfabeto Σ_1, que contiene todas las cadenas binarias de longitud 2,

$$L = \{00, 01, 10, 11\}$$

- El lenguaje sobre el alfabeto Σ_2, que contiene todas las palabras en español,

$$L = \{adios, ahora, algo, ...\}$$

- El lenguaje de todas las cadenas binarias que empiezan por 1,

$$L = \{1, 10, 11, 100, 101, 110, 111, ...\} = \{w \in \{0, 1\}^* : w \text{ empieza por } 1\}$$

Un automáta finito determinista es un modelo matemático de una máquina muy simple, con memoria limitada, que lee una cadena símbolo a símbolo y decide si debe aceptarla o rechazarla. Veamos la definición formal.

Definición 1.3. Un autómata finito determinista (AFD) es una 5-tupla $(Q, \Sigma, \delta, q_0, F)$, donde

1. Q es un conjunto finito de estados,

2. Σ es un conjunto finito de símbolos (alfabeto),
3. $\delta : Q \times \Sigma \to Q$ es la función de transición,
4. $q_0 \in Q$ es el estado inicial y,
5. $F \subseteq Q$ es el conjunto de estados aceptados.

Al procesar una cadena $w = a_1 a_2 \dots a_n \in \Sigma^*$, el autómata comienza con el estado inicial q_0 y aplica sucesivamente la función de transición a cada símbolo:

$$q_{i+1} = \delta(q_i, a_i) \quad \text{para } i = 0, \dots, n-1.$$

Decimos que el autómata acepta la cadena w si el estado en el que termina, q_n, pertenece a F. En caso contrario, la rechaza.

Ejemplo 1.4. Analicemos el autómata finito que acepta el lenguaje de las cadenas que terminan en 01.

$$L = \{w \in \{0,1\}^* : w \text{ termina en } 01\}.$$

Vamos a considerar estos tres estados:

- q_0 es el estado inicial,
- q_1 es el estado cuando se ha añadido un 0 y
- q_2 es el estado cuando se ha añadido un 1 después de un 0.

Tenemos entonces que $Q = \{q_0, q_1, q_2\}$ y el estado final es $F = \{q_2\}$. Además, el alfabeto será $\Sigma = \{0, 1\}$, aunque podríamos considerar cadenas con más símbolos y seguiría funcionando de la misma forma. Finalmente, consideremos la siguiente función de transición:

δ	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

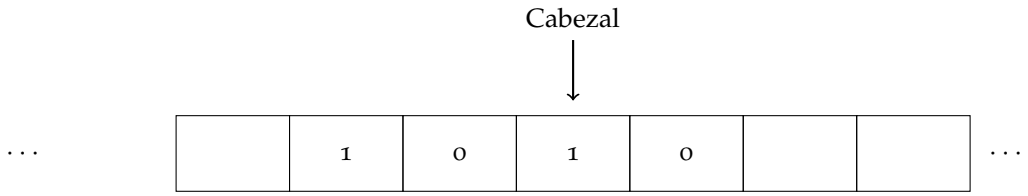
Observemos que cada vez que se añade un 0, se pasa al estado q_1 independientemente del estado en el que nos encontremos. Si estamos en el estado q_1 y añadimos un 1, pasamos al estado final q_2 en el que podemos seguir añadiendo símbolos o parar.

■

Vemos, por tanto, que un autómata finito determinista es un modelo matemático que procesa cadenas de símbolos y decide si deben ser aceptadas o rechazadas según su función de transición y sus estados de aceptación.

Una máquina de Turing puede entenderse como una generalización de este modelo. Al igual que un AFD, dispone de estados de aceptación y de rechazo, y el cómputo termina en cuanto la máquina entra en alguno de ellos. Si nunca alcanza ninguno de estos estados, la máquina continúa ejecutándose indefinidamente.

La diferencia esencial es que la máquina de Turing dispone de una memoria potencialmente infinita, implementada mediante una cinta infinita dividida en celdas, y una cabeza lectora que puede moverse a izquierda o derecha y que permite tanto leer como escribir símbolos en la cinta. Esta memoria sin restricciones hace que las máquinas de Turing sean un modelo mucho más expresivo que los autómatas finitos.



Definición 1.5. Una máquina de Turing determinista es una 7-tupla

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{aceptar}, q_{rechazar}),$$

donde,

1. Q es un conjunto finito de estados,
2. Σ es el alfabeto de entrada, que no contiene el espacio en blanco ($\square \notin \Sigma$),
3. Γ es el alfabeto de la cinta, donde $\square \in \Gamma$ y $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ es la función de transición,
5. $q_0 \in Q$ es el estado inicial,
6. $q_{aceptar} \in Q$ es el estado de aceptar y
7. $q_{rechazar} \in Q$ es el estado de rechazar ($q_{aceptar} \neq q_{rechazar}$).

Notemos que, en las máquinas de Turing, la función de transición tiene ahora la forma

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

El significado de una transición

$$\delta(q, a) = (q', b, D)$$

es el siguiente: si la máquina se encuentra en el estado q y la cabeza de lectura se sitúa sobre una celda que contiene el símbolo a, entonces la máquina pasa al estado q', escribe el símbolo b en esa misma celda y, finalmente, desplaza la cabeza una posición en la dirección indicada por D. En particular, si $D = L$ la cabeza se mueve una posición a la izquierda y, si $D = R$, una posición a la derecha.

De ahora en adelante, si no se especifica lo contrario, con máquina de Turing nos referimos a las deterministas que acabamos de definir.

Definición 1.6. Sea M una máquina de Turing. El conjunto de todas las cadenas que M acepta se denomina lenguaje reconocido por M y se denota por $L(M)$.

Un lenguaje L se dice reconocible si existe una máquina de Turing M tal que

$$L = L(M),$$

es decir, M acepta exactamente todas las cadenas de L.

Dada una cadena de entrada, una máquina de Turing puede aceptarla, rechazarla o bien no detenerse nunca (entrar en bucle).

Definición 1.7. Una máquina de Turing se denomina decisor (o algoritmo) si se detiene para toda cadena de entrada, es decir, si para cualquier cadena acepta o rechaza en un número finito de pasos.

Un lenguaje L se dice decidible si existe un decisor que lo decide, esto es, una máquina de Turing que acepta exactamente las cadenas de L y rechaza todas las demás.

Veamos un ejemplo de un lenguaje reconocible y decidible. Que sea reconocible implica que existe una máquina de Turing que aceptará todos las cadenas del lenguaje, y las cadenas que no pertenezcan las rechazará o se quedará en bucle. Que sea decidible implica que existe una máquina que aceptará o rechazará siempre en tiempo finito.

Ejemplo 1.8. Consideremos el lenguaje

$$L = \{0^i 1^{i+j} 0^j : i, j > 0\}$$

conteniendo las cadenas con i ceros al principio, seguidos de $i + j$ unos y que terminan en j ceros al final. Por ejemplo, las cadenas $0110 \in L$ y $001110 \in L$, pero $01110 \notin L$ y $00110 \notin L$.

Dada una cadena, vamos a colocar cada símbolo suyo en una casilla de la cinta. Los espacios en blanco ($\square$) marcan el principio y el final de la cadena.

La máquina de Turing que decide este lenguaje empieza marcando el primer 0 con una X y se mueve hacia la derecha hasta encontrar un 1, que marcará con una Y. Este proceso lo repetirá para todos los ceros de la izquierda. Luego, empieza a marcar los ceros del final con una Z y, por cada 0 a la derecha, busca un 1 sin marcar a la izquierda y lo sustituye por una Y.

Si todo encaja, es decir, si al marcar el último 0, se marca el último 1 y viceversa, entonces la cadena se acepta. Si se encuentra algún fallo, entonces la cadena se rechaza.

La máquina de Turing podría contener los estados siguientes:

- q_0 estado inicial,
- q_1 marca un 0 con X y busca un 1 a la derecha,
- q_2 marca un 1 con Y y busca un 0 al principio,
- q_3 detecta fin de 0s iniciales y busca 0s al final de la cadena,
- q_4 marca 0 con Z y busca un 1 a la izquierda sin marcar,
- q_5 marca 1 con Y y busca un 0 al final,
- q_6 comprueba que no quedan símbolos sin marcar y
- $q_{aceptar}$ y $q_{rechazar}$.

Observemos que el alfabeto de entrada $\Sigma = \{0, 1\}$ es diferente al alfabeto de la cinta $\Gamma = \{0, 1, X, Y, Z, \square\}$.

■

Definición 1.9. Una máquina de Turing multicinta es una máquina de Turing que dispone de varias cintas de trabajo, cada una de ellas con su propia cabeza de lectura y escritura.

En el caso de una máquina con k cintas, la función de transición tiene la forma

$$\delta\colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k.$$

Dada una configuración

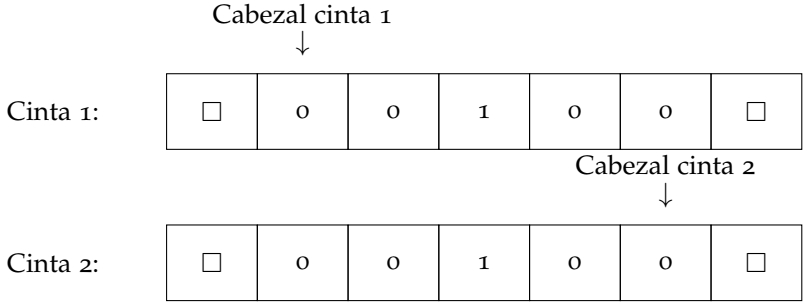
$$\delta(q, a_1, \ldots, a_k) = (q', b_1, \ldots, b_k, D_1, \ldots, D_k),$$

esto significa que, si la máquina se encuentra en el estado q y las cabezas de las k cintas leen respectivamente los símbolos $a_1, \ldots, a_k$, entonces:

- la máquina pasa al estado q',
- en cada cinta j, con $1 \leq j \leq k$, se escribe el símbolo b_j en la celda actual,
- la cabeza de la cinta j se desplaza una posición en la dirección indicada por D_j, donde $D_j \in \{L, R, S\}$, es decir, L indica movimiento a la izquierda, R a la derecha y S que la cabeza permanece en la misma celda.

Ejemplo 1.10. Veamos un ejemplo de máquina con dos cintas. Para ello, consideremos el lenguaje $L = \{w \in \{0,1\}^* : w = w^R\}$, donde w^R es el reverso de w. Es decir, el lenguaje contiene las cadenas que son palíndromas. Por ejemplo, 00100 o 10101.

Una máquina multicinta que reconoce este lenguaje podría tener dos cintas: en la primera cinta se encuentra la cadena original; en la segunda cinta, la cadena en orden inverso. Se trataría entonces de comparar que ambas cintas son idénticas.



Para copiar la cadena en orden inverso, se trataría de situar la cabeza de la primera cinta al principio de la cadena. Dicha cabeza lee el valor y la cabeza de la segunda cinta lo copia. Luego, la cabeza de la primera cinta se desplaza a la derecha y la de la segunda cinta a la izquierda. El proceso se repite mientras la primera cabeza no lea el espacio en blanco.

Para comparar que las cintas son idénticas, situamos las cabezas de las cintas al inicio de las cadenas que guardan. A continuación, leemos ambos valores: si coinciden, ambas cabezas se mueven una posición a la derecha; si no coinciden, se rechaza la cadena. Cuando se lea un espacio en blanco, si no se ha rechazado antes, entonces se acepta la cadena.

■

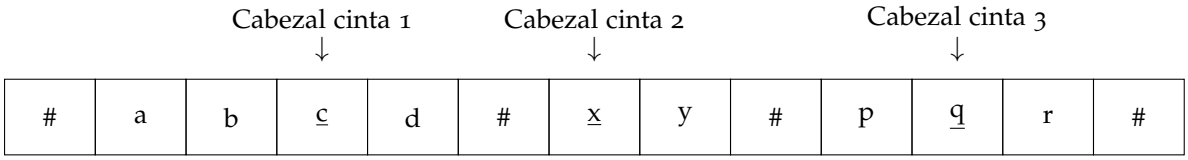
Teorema 1.11. *Toda máquina de Turing multicinta puede ser simulada por una máquina de Turing de una sola cinta equivalente.*

Demostración. Consideremos una máquina de Turing multicinta M con $k > 1$ cintas. Vamos a construir una máquina de Turing de una sola cinta S que simule el comportamiento de M.

La idea es codificar el contenido de las k cintas de M en una única cinta de S. Para ello:

- Usamos el símbolo especial # para separar los contenidos de las distintas cintas.
- En cada bloque delimitado por dos símbolos #, subrayamos exactamente un símbolo para marcar la posición de la cabeza de esa cinta.

De este modo, entre dos símbolos # siempre aparecerá un único símbolo subrayado que indica la posición actual de la cabeza en la cinta correspondiente. Por ejemplo, la configuración



representa una máquina con tres cintas, en las que las cabezas se encuentran sobre los símbolos c, x y q, respectivamente.

La función de transición de la máquina de cinta única S debe simular, paso a paso, la función de transición de la máquina multicinta M. Un paso de simulación de M por parte de S se realiza en dos fases:

1. Primero, S recorre toda su cinta para localizar, en cada uno de los k bloques delimitados por #, el símbolo subrayado. Esto permite determinar qué símbolo está leyendo cada una de las k cabezas de M.
2. A continuación, usando esta información y el estado actual, S calcula qué transición de M debe simular. Luego vuelve a recorrer la cinta para:
   - reemplazar cada símbolo subrayado por el nuevo símbolo que correspondería tras aplicar la transición,
   - mover la marca de subrayado una posición a la izquierda o a la derecha, o dejarla en la misma celda, según indique la dirección de movimiento de la cabeza de cada cinta.

Si en algún momento una de las cintas simuladas necesita escribir un símbolo en una celda que estaba en blanco (por ejemplo, más allá del contenido actual), la máquina S debe desplazarse y, en caso necesario, correr todos los símbolos situados a la derecha para crear espacio, de forma que la codificación siga siendo correcta.

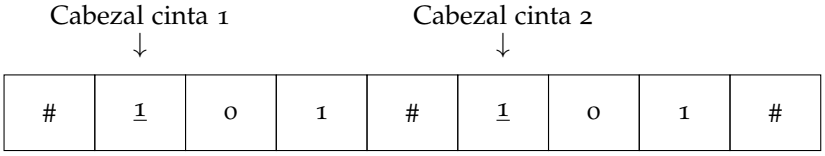
De esta manera, por cada paso de cómputo de M, la máquina S realiza un número finito de pasos propios y mantiene una codificación exacta del contenido de todas las cintas y de las posiciones de sus cabezas. Por tanto, S simula correctamente a M y reconoce el mismo lenguaje. □

Vemos que la máquina de cinta única tiene las mismas capacidades que la multicinta aunque requiere de muchos más pasos para escanear la cinta y actualizar las cabezas.

Ejemplo 1.12. Veamos el ejemplo anterior con una máquina de cinta única. Recordemos que queríamos una máquina que reconociera el lenguaje de las cadenas palíndromas,

$$L = \{w \in \{0,1\}^* : w = w^R\}.$$

Ahora las dos cintas no se encuentran separadas, sino en una cinta única dividida por #.



■

Definición 1.13. Una máquina de Turing no determinista es aquella en la que la función de transición puede tener múltiples posibles resultados para una misma configuración (estado y símbolo leído). Su función de transición tiene la forma

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times L, R).$$

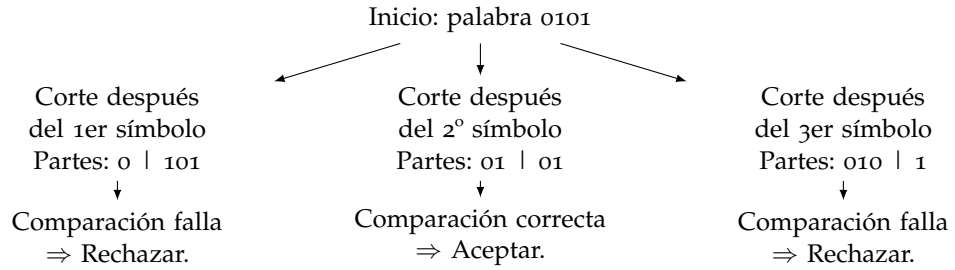
Se puede ver como un árbol donde un nodo es una configuración de la máquina y sus nodos hijos son las posibles configuraciones a las que se puede llegar mediante la función de transición. De forma que si alguna de las ramas acepta, la cadena es aceptada.

Ejemplo 1.14. Consideremos el lenguaje de las cadenas que están formadas por una subcadena repetida dos veces,

$$L = \{w \in \{0,1\}^* : w = xx \quad \text{con} \quad x \in \{0,1\}^*\}.$$

Por ejemplo, 0101 o 110110. Una máquina de Turing que reconozca este lenguaje puede ser aquella que cuenta la cantidad de símbolos de la cadena, divide por la mitad y calcula el punto de corte entre ambas subcadenas. Una vez conocido el punto de corte, solo tendría que comprobar que la cadena es la misma a ambos lados de este punto.

Sin embargo, una máquina no determinista no necesita calcular dicho punto de corte explícitamente, sino que puede probar paralelamente con todos los puntos de corte posibles (todas las posiciones de la cadena).



Como hay una rama del árbol que acepta, la cadena es aceptada. Independientemente de si el resto de las ramas rechazan o quedan en bucle.

■

Teorema 1.15. *Toda máquina de Turing no determinista tiene una máquina de Turing determinista equivalente.*

Demostración. Dada una máquina de Turing no determinista, podemos construir una máquina de Turing determinista que simule su comportamiento y acepte exactamente el mismo lenguaje.

La idea es que toda máquina no determinista puede verse como una que, en cada configuración, bifurca su ejecución en varias posibles ramas. La ejecución se ve como un árbol donde:

- Cada nodo representa una configuración de la máquina.
- Los hijos de un nodo corresponden a las diferentes posibles transiciones no deterministas desde esa configuración.

Numeramos las ramas de este árbol usando cadenas finitas de números naturales. Por ejemplo, la cadena 312 representa el camino que toma:

- la tercera opción desde la raíz,
- luego la primera opción desde el siguiente nodo,
- y después la segunda opción.

Ahora construiremos una máquina de Turing determinista (DTM) que explore todas las posibles ramas de este árbol. Para ello, usamos una máquina de Turing multicinta con 3 cintas (sabemos que es equivalente a una de cinta única):

- Cinta 1: contiene la entrada original, que no se modifica.
- Cinta 2: contiene una copia de la entrada que se irá modificando durante la simulación de una rama concreta del árbol.
- Cinta 3: contiene una cadena de enteros que codifica una rama del árbol (es decir, una secuencia de elecciones de transiciones no deterministas).

El funcionamiento es el siguiente:

1. La máquina copia la entrada de la cinta 1 a la cinta 2.
2. Luego simula la ejecución de la máquina no determinista siguiendo la ruta indicada por la cinta 3.
3. Si esta simulación llega a un estado de aceptación, la máquina determinista acepta.
4. Si la simulación se bloquea (transición inválida, rechazo o fin de la rama sin aceptar), se reinician las cintas 2 y 3, y se genera la siguiente rama.

De este modo, la máquina determinista explora todas las posibles ejecuciones de la máquina no determinista. Como acepta si alguna rama acepta, se asegura que acepta la misma entrada si y solo si la máquina original no determinista la habría aceptado. □

Lema 1.16. *Un lenguaje es decidible si y solo si existe una máquina de Turing no determinista que lo decide.*

Demostración. Probaremos ambas implicaciones por separado.

Para la implicación a la derecha, supongamos que L es un lenguaje decidible. Por definición, existe una máquina de Turing determinista M que decide L, es decir, para toda cadena de entrada w:

- M se detiene aceptando si $w \in L$,
- M se detiene rechazando si $w \notin L$.

Toda máquina determinista es un caso particular de máquina no determinista (basta considerar que en cada paso solo tiene una única transición posible). Por tanto, podemos ver a M como una máquina de Turing no determinista que también decide L.

Así, si L es decidible, entonces existe una máquina de Turing no determinista que lo decide.

Para la implicación a la izquierda, supongamos ahora que existe una máquina de Turing no determinista N que decide el lenguaje L. Esto significa que, para toda cadena de entrada w:

- todas las ramas de cómputo de N sobre w se detienen (no hay bucles infinitos),
- N acepta w si y solo si al menos una rama de cómputo termina en un estado de aceptación,
- N rechaza w si y solo si todas las ramas terminan en un estado de rechazo.

Construiremos una máquina de Turing determinista M que también decide L. La idea es que M simule de forma sistemática todas las posibles ramas de cómputo no deterministas de N sobre la entrada w.

Podemos ver la ejecución de N sobre w como un árbol de cómputo, donde:

- cada nodo representa una configuración de N (estado, contenido de la cinta, posición de la cabeza),
- la raíz es la configuración inicial con entrada w,
- los hijos de un nodo representan las configuraciones obtenidas tras aplicar una de las transiciones no deterministas posibles.

La máquina determinista M recorre este árbol de cómputo por niveles (búsqueda en anchura). En cada paso:

1. M mantiene una descripción codificada de todas las configuraciones del nivel actual.
2. A partir de ellas, genera todas las configuraciones del siguiente nivel aplicando las transiciones de N.
3. Si durante este proceso encuentra alguna configuración aceptadora, M acepta la entrada.
4. Si todas las configuraciones posibles que aparecen son de rechazo y no quedan más niveles por explorar, M rechaza la entrada.

Como N es un decisor, todas las ramas de cómputo son finitas. Por tanto, el árbol de cómputo sobre cualquier entrada w es finito, y la exploración por niveles que realiza M termina siempre en un número finito de pasos. Además:

- Si $w \in L$, por hipótesis existe al menos una rama de cómputo aceptadora de N sobre w, de modo que M la encontrará y aceptará w.
- Si $w \notin L$, todas las ramas terminan en rechazo, por lo que M acabará rechazando w.

En consecuencia, M se detiene para toda entrada y acepta exactamente las cadenas de L. Por tanto, M decide L y, en particular, L es decidible.

Hemos probado ambas implicaciones, luego un lenguaje es decidible si y solo si existe una máquina de Turing no determinista que lo decide. □

1.2. Complejidad en tiempo

En teoría de la computación no basta con saber que existe una máquina de Turing capaz de reconocer un lenguaje, en la práctica también interesa saber cuánto tarda en hacerlo. Dos máquinas pueden resolver el mismo problema, pero una podría necesitar un número de pasos exponencial respecto al tamaño de entrada, mientras que la otra podría tardar mucho menos. Esta diferencia es importante para decidir si un problema es abordable en la práctica.

Definición 1.17. Sea M una máquina de Turing determinista que se detiene para cualquier entrada. Definimos el tiempo de ejecución (o complejidad temporal) de M como la función

$$f : \mathbb{N} \to \mathbb{N}$$

donde $f(n)$ es el máximo número de pasos que realiza M sobre cualquier cadena de entrada de longitud n.

En general, la expresión exacta del tiempo de ejecución en función del tamaño de la entrada suele ser complicada. Por ello se utiliza la notación O para describir el crecimiento asintótico de la función de tiempo.

Cuando la función de tiempo es un polinomio, se considera únicamente el término de mayor grado (puesto que domina el comportamiento para entradas grandes), ignorando los coeficientes y los términos de menor grado. Si la función no es polinómica, la notación O permite expresar su orden de crecimiento comparándola con funciones conocidas, por ejemplo $O(\log n)$, $O(n \log n)$ u $O(2^n)$.

Ejemplo 1.18. Sea $f(n) = 7n^4 + 3n + 1$, decimos que una máquina de tiempo $f(n)$ es de orden $O(n^4)$.

■

Ejemplo 1.19. Veamos un ejemplo de cómo calcular la complejidad de tiempo de una máquina de Turing determinista de cinta única. Consideremos el lenguaje de las cadenas formadas por una cantidad par de ceros y que tienen un 1 en la mitad.

$$L = \{0^k 1 0^k : k > 1\}$$

Una máquina que decide este lenguaje copia la cadena de entrada en su cinta y ejecuta las siguientes operaciones:

1. Escanea toda la cinta y rechaza si se encuentran dos o más 1s.
2. Mientras queden 0s por marcar, escanea la cinta y marca un 0 a la izquierda por cada 0 a la derecha.
3. Si quedan 0s en alguno de los lados por marcar, después de haber marcado todos los del otro lado, entonces rechaza. Si solo queda el 1 por marcar, entonces acepta.

Analicemos cuántos pasos se tardan en ejecutar cada operación descrita. La primera operación trata de escanear la cinta completa, que tarda n pasos cuando la entrada es de tamaño n. Luego, la primera operación tarda $O(n)$ pasos.

Para la segunda operación se ejecutará como máximo $n/2$ veces, puesto que en cada escaneo de la cinta se marcan 2 símbolos. Y ya sabemos para escanear la cinta tardamos n pasos. Luego, la segunda operación tarda $(n/2)O(n) = O(n^2)$.

Finalmente, la tercera operación hace un último escaneo para decidir el resultado, luego tarda $O(n)$.

Tenemos entonces que la máquina tarda $O(n) + O(n^2) + O(n) = O(n^2)$ pasos en decidir el resultado.

■

Teorema 1.20. *Si $t(n)$ es una función con $t(n) \geq n$, entonces toda máquina de Turing multicinta que opera en tiempo $t(n)$ puede ser simulada por una máquina de Turing de cinta única que opera en tiempo $O(t^2(n))$.*

Demostración. Para demostrar este teorema recordamos el procedimiento de simulación de una máquina multicinta mediante una máquina de cinta única.

Codificamos el contenido de las k cintas en una única cinta, separando cada cinta con el símbolo especial '#' e indicando la posición de cada cabeza mediante un subrayado.

Inicialmente, la entrada de tamaño n debe ser transformada a este nuevo formato. Esta operación implica recorrer toda la entrada una vez, lo que requiere $O(n)$ pasos.

Durante la simulación:

- Cada paso de la máquina multicinta corresponde a dos escaneos completos de la cinta única: uno para leer las posiciones de las cabezas y decidir las acciones, y otro para actualizar los símbolos y mover las cabezas simuladas.
- Como la longitud de la cinta única es proporcional a $kt(n)$ (cada cinta puede usar hasta $t(n)$ celdas en el peor caso), y k es constante, escanear toda la cinta requiere $O(t(n))$ pasos.
- Dado que la máquina multicinta realiza $t(n)$ pasos, la simulación completa requiere $t(n)O(t(n)) = O(t^2(n))$ pasos.

Por tanto, la máquina de cinta única completa la simulación en un total de $O(n) + O(t^2(n))$ pasos.

Como asumimos que $t(n) \geq n$, el término $O(n)$ queda absorbido dentro de $O(t^2(n))$, y el tiempo total de simulación es $O(t^2(n))$.

Cabe destacar que la hipótesis $t(n) \geq n$ no es restrictiva, ya que leer toda la entrada requiere como mínimo n pasos. □

Teorema 1.21. *Si $t(n)$ es una función con $t(n) \geq n$, entonces toda máquina de Turing no determinista que opera en tiempo $t(n)$ puede ser simulada por una máquina de Turing determinista que opera en tiempo $2^{O(t(n))}$.*

Demostración. Para demostrar este teorema tenemos que recordar la forma de transformar una máquina de Turing no determinista en una determinista. Lo que hacíamos era simular la máquina no determinista explorando un árbol donde: cada nodo representa una configuración de la máquina y cada rama una decisión posible de la función de transición.

Cada rama del árbol tiene tamaño máximo $t(n)$ pasos, ya que la máquina no determinista termina a lo sumo en $t(n)$ pasos. Por otro lado, cada nodo tiene máximo b hijos, siendo b el número máximo de transiciones para una sola configuración. Luego,

$$\text{número de hojas} = b \times b \times ... \times b = b^{t(n)}$$

El árbol se explora en anchura, ya que no queremos quedarnos atrapados en una rama infinita. Y sabemos que viajar de la raíz a una hoja tarda máximo $t(n)$ pasos. Por tanto, recorrer el árbol tiene tiempo total $O(t(n)b^{t(n)}) = O(b^{t(n)})$, porque el crecimiento exponencial domina al polinómico. Por otro lado,

$$b^{t(n)} = 2^{t(n)\log_2 b} = 2^{O(t(n))}.$$

Concluimos con que la máquina determinista que simula la no determinista opera en tiempo $2^{O(t(n))}$.

□

Con estos dos teoremas anteriores vemos que hay una diferencia como mucho polinómica entre el tiempo de las máquinas de cinta única y las multicinta. Como los tiempos polinómicos no suelen tener un grado demasiado elevado, consideramos que esta diferencia no es grande.

Por otro lado, la diferencia entre una máquina no determinista y una determinista es como máximo exponencial. Esta diferencia sí se considera grande, ya que para una entrada de $n = 1000$, el número de pasos de 2^n es inmensa, más grande que la cantidad de átomos que hay en el Universo.

1.3. Complejidad de problemas

Una máquina de Turing es un modelo matemático que formaliza lo que entendemos por algoritmo, es decir, un procedimiento finito de pasos que determina si una entrada constituye o no una solución a un problema.

Definición 1.22. Un problema de decisión es un conjunto de entradas para las cuales la respuesta es "sí" o "no".

Cuando pensamos en problemas, suelen venirnos a la mente ejemplos como el problema del camino más corto en un grafo, o determinar si un número es primo. Lo que queremos ver es que estos problemas no son más que conjuntos de entradas para las cuales la respuesta es afirmativa; en otras palabras, son lenguajes.

Cada entrada de un problema (ya sea un grafo, un número o una ecuación) puede representarse mediante una cadena finita de símbolos. Por ejemplo, un grafo puede describirse

como una lista de vértices y aristas codificada en una cadena, un número puede representarse en binario, y una ecuación puede expresarse como un texto que describe su estructura.

Por otro lado, sabemos que las máquinas de Turing operan sobre cadenas de símbolos. Por este motivo, afirmamos que las máquinas de Turing pueden modelar algoritmos para cualquier problema de decisión: si la máquina implementa un algoritmo que resuelve el problema, aceptará las entradas que correspondan a soluciones válidas y rechazará las demás.

En conclusión, resolver un problema de decisión equivale a construir una máquina de Turing que acepte exactamente las entradas que pertenecen a un determinado lenguaje y rechace las que no.

Una vez aclarada esta conexión entre problemas, lenguajes y máquinas de Turing, vamos a estudiar las distintas clases de problemas que existen.

Definición 1.23. P es la clase de lenguajes decidibles en tiempo polinómico por una máquina de Turing determinista.

Esta es la clase de problemas que se pueden resolver por un ordenador en tiempo razonable, ya que agrupa los problemas para los cuales existe un método sistemático y determinista para resolverlos en un tiempo razonable, incluso para entradas grandes. Esta clase incluye problemas tradicionales como:

- Búsqueda en una lista ordenada: se puede realizar en $O(\log n)$ mediante búsqueda binaria.
- Ordenación de una lista:se puede realizar en $O(n \log n)$ con algoritmos como MergeSort o QuickSort.
- Cálculo de caminos mínimos en un grafo: se puede realizar en $O(n^2)$ con el algoritmo de Dijkstra en grafos densos.
- Multiplicación de matrices: se realiza en $O(n^3)$ con el algoritmo clásico.

Ejemplo 1.24. El problema de búsqueda en una lista ordenada se define como el lenguaje

$$L = \{(A, x) : A \text{ es una lista ordenada y } x \text{ esta en } A\}.$$

Una máquina que resuelve este problema puede simular el algoritmo de búsqueda binaria:

1. Marcamos dos índices $izq = 0$ y $der = n - 1$, que representan el principio y el final de la lista.
2. Calculamos el punto medio como $medio = (izq + der)/2$ y comparamos el valor de x. Si $A[medio] = x$, entonces aceptamos; si $A[medio] < x$, entonces descartamos la mitad izquierda y $izq = medio + 1$; si $A[medio] > x$, descartamos la mitad derecha y $der = medio - 1$.
3. Si el bucle termina sin encontrarlo, entonces rechazamos.

Si nos fijamos, este algoritmo divide el espacio de búsqueda a la mitad en cada paso. Esto quiere decir que: si la lista tiene 8 elementos, como máximo se hacen 3 pasos ($\log_2 8 = 3$) y si la lista tiene 1024 elementos, como máximo se hacen 10 pasos ($\log_2 1024 = 10$).

Por tanto, la cantidad de pasos está acotada por el logaritmo del tamaño de la lista $T(n) = O(\log n)$, que es menor tiempo que polinómico. Es decir, que este problema está en P.

■

Definición 1.25. NP es la clase de problemas de decisión que se pueden resolver por una máquina de Turing no determinista en tiempo polinomial.

Aunque la definición de NP se basa en máquinas de Turing no deterministas, otra forma intuitiva de describir este tipo de problemas es usando verificadores.

Definición 1.26. Un verificador de un lenguaje A es un algoritmo V, donde

$$A = \{w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c\}.$$

Si el verificador se ejecuta en tiempo polinómico, decimos que el lenguaje A es verificable polinómicamente.

De manera informal, un verificador es un algoritmo que te indica si una cadena dada pertenece al lenguaje, es decir, si es una solución del problema de decisión. Tenemos que c representa una posible solución para la instancia del problema w.

Ejemplo 1.27. Un ejemplo clásico es el problema SAT (Satisfiability Problem) que recibe una fórmula booleana en forma conjuntiva (AND de cláusulas OR). Supongamos:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_3).$$

El problema consiste en determinar si existe una asignación de valores de las variables que haga que la fórmula se evalúe como verdadera. Siguiendo con el ejemplo, una posible asignación sería

$$x_1 = 1, x_2 = 0, x_3 = 0.$$

Entonces la instancia del problema que hemos descrito con la fórmula sí que pertenece al lenguaje SAT porque existe una asignación de valores a las variables que evalúa la fórmula como verdadera.

Formalmente, el lenguaje SAT se define como:

$$SAT = \{w : w \text{ es una fórmula booleana satisfacible}\},$$

donde w es una entrada booleana que se dice satisfacible si existe una asignación de valores a sus variables que la evalúa como verdadera.

Un verificador para SAT es un algoritmo que recibe dos elementos:

- La entrada w es la instancia del problema. En este caso, la fórmula booleana.
- El certificado c es una posible asignación de valores para las variables.

El verificador lee la fórmula w y sustituye los valores del certificado c en la fórmula. Finalmente, comprueba si la fórmula es verdadera con esa asignación.

Este algoritmo es de tiempo polinómico porque simplemente sustituye los valores y evalúa la fórmula. Por tanto, SAT es verificable polinómicomente.

■

Con el siguiente teorema veremos que, equivalentemente, NP es la clase de lenguajes que tienen un verificador de tiempo polinómico.

Teorema 1.28. *Un lenguaje pertenece a NP si, y solo si, admite un verificador de tiempo polinómico.*

Demostración. Para la implicación a la derecha, sea $L \in NP$. Entonces existe una máquina de Turing no determinista que decide L en tiempo polinómico.

Dada una entrada w, una rama de la máquina no determinista de N puede aceptar w en tiempo polinómico. Se puede construir entonces un verificador V que, dada una pareja $\langle w, c\rangle$, interpreta c como la descripción de las elecciones no deterministas de N. Entonces el verificador simula N siguiendo dichas elecciones y acepta si la rama llega al estado de aceptación. Como N se ejecuta en tiempo polinómico, la simulación también lo hace, y V es un verificador polinómico.

Para la implicación a la izquierda, supongamos que existe un verificador polinómico V para un lenguaje L. Se puede construir una máquina de Turing no determinista N que, al recibir w, adivina de forma no determinista un certificado c de longitud polinómica y ejecuta V sobre $\langle w, c\rangle$. Como V se ejecuta en tiempo polinómico y c es de tamaño polinómico, N se detiene en tiempo polinómico. □

A continuación, vamos a desarrollar otro ejemplo crucial para este proyecto.

Ejemplo 1.29. El problema de la suma de subconjuntos (SSP, por sus siglas en inglés) busca responder si, dado un conjunto de enteros $S = \{s_1, \ldots, s_n\}$ y un entero t, existe un subconjunto de S cuyos elementos sumen t. El lenguaje asociado sería el siguiente:

$$SS = \{\langle S, t\rangle : S = \{x_1, ..., x_n\} \text{ y existe un subconjunto } S' = \{y_1, ..., y_m\} \subset S \text{ tal que } \sum y_i = t\}.$$

Veamos que el problema de la suma de subconjuntos está en NP. Buscamos un algoritmo V de tiempo polinómico que recibe $\langle\langle S, t\rangle, c\rangle$ y nos indica si c es solución de la instancia del problema $\langle S, t\rangle$. Para ello:

1. Comprueba si c es un subconjunto de S.
2. Comprueba si los elementos de c suman t.
3. Si ambos se cumplen, entonces acepta. De lo contrario, rechaza.

Como el verificador solo tiene que recorrer el subconjunto y sumar los elementos, ambas partes tienen coste $O(n)$, luego el verificador es de tiempo polinómico y $SS \in NP$.

■

Es importante destacar que los problemas en NP no necesariamente pueden resolverse rápidamente, pero si obtenemos una posible solución, podemos verificarla en tiempo polinómico. En el E.j. 1.27 hemos encontrado un algoritmo que nos dice en tiempo polinómico si una asignación de valores es solución de una instancia del problema SAT. Sin embargo, no se ha encontrado un algoritmo que, dada la fórmula booleana, concluya si existe una asignación de valores que la satisfaga.

1.4. Reducción polinómica

Reducir un problema A a otro problema B es equivalente a demostrar que una solución de B sirve para resolver A.

Definición 1.30. Una función $f : \Sigma^* \to \Sigma^*$ es una función computable de tiempo polinómico si existe una máquina de Turing de tiempo polinómico que se detiene con sólo $f(w)$ en su cinta, cuando se empieza con cualquier entrada w.

Definición 1.31. Un lenguaje A es polinómicamente reducible a B ($A \leq_P B$) si existe una función computable de tiempo polinómico $f : \Sigma^* \to \Sigma^*$ donde para cada w,

$$w \in A \iff f(w) \in B.$$

Esto quiere decir que si un lenguaje A es reducible polinómicamente a otro lenguaje B y sabemos que B se puede decidir en tiempo polinómico, entonces A también. Lo demostraremos en el siguiente teorema:

Teorema 1.32. *Si $A \leq_P B$ y $B \in P$, entonces $A \in P$.*

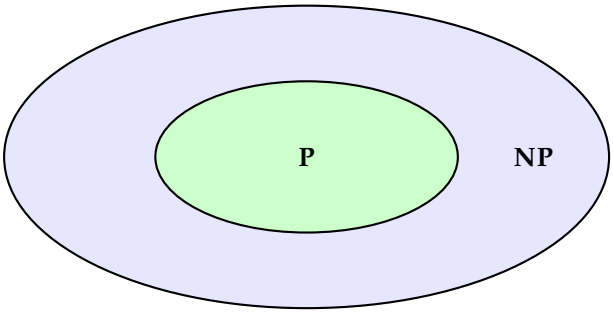
Demostración. Sea M la máquina de Turing de tiempo polinomial que decide B y f la función de tiempo polinomial de A a B. Buscamos una máquina N de tiempo polinomial que decida A.

Definimos N como la máquina con entrada w que calcula $f(w)$ y ejecuta M con entrada $f(w)$. Tenemos que calcular $f(w)$ tarda tiempo polinomial por la definición de f y que M también se ejecuta en tiempo polinomial. Luego, N tiene tiempo $O(n)$. □

1.5. NP-completitud

El problema P vs NP busca determinar si ambas clases son equivalentes, es decir, si todo problema cuya solución pueda verificarse en tiempo polinómico también puede resolverse en tiempo polinómico.

Actualmente, se conocen muchos problemas en NP para los cuales no se ha encontrado un algoritmo en P, pero tampoco se ha demostrado que no pertenezcan a P. Resolver esta cuestión es uno de los problemas abiertos más importantes en la teoría de la computación y las matemáticas contemporáneas.



Intuitivamente, el problema P vs NP busca responder si el dibujo anterior es cierto o si ambos subconjuntos son iguales.

Durante el estudio del problema P vs NP, se identificó un conjunto de problemas especialmente relevantes llamados problemas NP-completos. Estos problemas son clave porque su

complejidad está estrechamente ligada a la de todos los problemas en NP: si se encontrara un algoritmo en tiempo polinómico para resolver uno de ellos, entonces todos los problemas en NP podrían resolverse en tiempo polinómico, es decir, se probaría que $P = NP$.

Definición 1.33. Un problema de decisión L es NP-completo si cumple dos condiciones:

1. L pertenece a NP.
2. Todo problema en NP puede reducirse en tiempo polinómico a L.

La segunda condición implica que, si encontramos un algoritmo eficiente para resolver L, podemos utilizarlo para resolver cualquier otro problema en NP en tiempo polinómico. Este proceso se conoce como reducción polinómica y es la herramienta fundamental para demostrar la NP-completitud de un problema.

Equivalentemente, un problema se dice NP-completo si el hecho de encontrar un algoritmo de tiempo polinómico que lo decida, implicaría la existencia de un algoritmo de tiempo polinómico que decida todos los problemas en NP. Es decir, si A es NP-completo entonces

$$A \in P \implies P = NP.$$

Ejemplo 1.34. Recordemos que previamente hemos definido el problema de SAT, que consiste en determinar si existe una asignación de valores de verdad para las variables de una fórmula booleana que haga que la fórmula se evalúe como verdadera.

El problema $3SAT$ es un caso particular de SAT en el que la fórmula booleana está expresada en forma normal conjuntiva (CNF), y además, cada cláusula contiene exactamente tres literales.

Una cláusula en este contexto tiene la forma:

$$(x_1 \vee \neg x_2 \vee x_3),$$

donde x_1, x_2, x_3 son variables booleanas o su negación, y $\vee$ representa la disyunción lógica. Una fórmula en 3CNF es una conjunción (AND) de varias de estas cláusulas de tres literales.

El problema consiste en determinar si existe una asignación de valores de verdad para las variables de forma que todas las cláusulas sean verdaderas simultáneamente.

El Teorema de Cook-Levin establece que el problema SAT es NP-completo. Posteriormente, se demuestra que 3SAT también es NP-completo, ya que SAT se puede reducir en tiempo polinómico a 3SAT.

■

Como teorema fundamental y final de este apartado, demostraremos que el problema de la suma de subconjuntos es NP-completo utilizando la reducción polinómica. Para ello, explicaremos un nuevo problema de decisión y veremos un resultado importante suyo.

Teorema 1.35. *3SAT es NP-completo.*

Este resultado es fundamental y ampliamente conocido en teoría de la complejidad computacional. Puede encontrarse en referencias como [27].

Teorema 1.36. *El problema de la suma de subconjuntos es NP-completo.*

Demostración. Para que el problema sea NP-completo tiene que cumplir dos restricciones:

- La suma de subconjunto es NP. Esta parte ya la hemos demostrado en un ejemplo anterior.
- El problema $3SAT \leq_P SS$, es decir, se puede reducir polinomialmente $3SAT$ a la suma de subconjuntos. Esta condición es la que demostraremos a continuación con ayuda de un ejemplo.

Tenemos que considerar una fórmula booleana ϕ en forma 3CNF, es decir, cláusulas conectadas por $\wedge$ que contienen tres variables conectados por $\vee$. Esta fórmula ϕ tendrá n variables $x_1, ..., x_n$ y m cláusulas $c_1, ..., c_m$. Y queremos una función de reducción que convierta ϕ en una instancia del problema SS. Para construir la función utilizaremos una tabla.

Por cada variable x_i de ϕ, tenemos dos filas en la tabla representadas por y_i y z_i con $1 \leq n \leq n$. Además, por cada cláusula c_i de ϕ tenemos dos filas g_i y h_i con $1 \leq i \leq m$. Y cada fila representará un entero del conjunto S leído en base 10, excepto la última fila, que representará la suma objetivo t también leído en base 10.

Para rellenar la tabla haremos lo siguiente: cada vez que no se especifica un 1, hay un 0. Por cada variable x_i existen dos filas y_i y z_i que se rellenarán con 1 en la posición i con $1 \leq i \leq n$, es decir, en las n primeras columnas. Para las m siguientes columnas, pondremos un 1 en la fila y_i si la variable x_i aparece en la cláusula c_j y pondremos un 1 en la fila z_i si la variable x_i aparece negada en la cláusula c_j donde $1 \leq j \leq m$ y $1 \leq i \leq n$. Para las m últimas filas pondremos un 1 en la columna de la cláusula c_j de las filas g_j y h_j con $1 \leq j \leq m$.

	1	2	3	4	c_1	c_2	c_3
y_1	1	0	0	0	1	0	1
z_1	1	0	0	0	0	0	0
y_2	0	1	0	0	1	1	0
z_2	0	1	0	0	0	0	0
y_3	0	0	1	0	0	1	0
z_3	0	0	1	0	1	0	1
y_4	0	0	0	1	0	0	0
z_4	0	0	0	1	0	1	1
g_1	0	0	0	0	1	0	0
h_1	0	0	0	0	1	0	0
g_2	0	0	0	0	0	1	0
h_2	0	0	0	0	0	1	0
g_3	0	0	0	0	0	0	1
h_3	0	0	0	0	0	0	1
t	1	1	1	1	3	3	3

Tabla 1.1.: Tabla de reducción de $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$ al problema de Suma de Subconjuntos (SS)

Veamos ahora que ϕ es satisfacible si, y solo si, existe un subconjunto de S cuyos elementos suman t.

Para la implicación a la derecha, tenemos una asignación de valores 0 ó 1 para cada variable x_i de ϕ y queremos ver que existe un subconjunto de S que sume t. Seleccionaremos la fila y_i si la variable x_i está asignada a 1 y seleccionaremos la fila z_i si la variable x_i está asignada a 0. De esta forma, de las primeras $2n$ filas, se seleccionarán n y al sumar por columnas tendremos que los n primeros dígitos de t serán 1. En la parte derecha de las cláusulas, al sumar por columnas las filas seleccionadas, tenemos que cada columna suma

entre 1 y 3. Para llegar a 3 por cada columna de la derecha, podemos seleccionar las filas g_i y/o h_i que sirvan.

Véamoslo con la tabla ejemplo. Una posible asignación de las variables de ϕ que satisface la fórmula es

$$\{x_1 = 1, x_2 = 1, x_3 = 0, x_3 = 0\}.$$

Como $x_1 = 1$, se selecciona la fila y_1; como $x_2 = 1$, se selecciona la fila y_2; como $x_3 = 0$, se selecciona la fila z_3; y como $x_4 = 0$, se selecciona la fila z_4. Con esas filas seleccionadas, la suma de la columna de c_1 es 3, así que no seleccionamos ni g_1 ni h_1; la suma de la columna de c_2 es 2, así que seleccionamos g_2 o h_2; la suma de la columna de c_3 es 3, así que no seleccionamos ninguna fila más.

Hemos encontrado un subconjunto de S que suma $t = 1111333$:

$$S' = \{1000101, 100110, 10101, 1011, 10\}.$$

	1	2	3	4	c_1	c_2	c_3
y_1	1	0	0	0	1	0	1
z_1	1	0	0	0	0	0	0
y_2	0	1	0	0	1	1	0
z_2	0	1	0	0	0	0	0
y_3	0	0	1	0	0	1	0
z_3	0	0	1	0	1	0	1
y_4	0	0	0	1	0	0	0
z_4	0	0	0	1	0	1	1
g_1	0	0	0	0	1	0	0
h_1	0	0	0	0	1	0	0
g_2	0	0	0	0	0	1	0
h_2	0	0	0	0	0	1	0
g_3	0	0	0	0	0	0	1
h_3	0	0	0	0	0	0	1
t	1	1	1	1	3	3	3

Tabla 1.2.: Selección de subconjunto de S

Para la implicación a la izquierda, tenemos un subconjunto de S que suma t y queremos construir una asignación para las variables de ϕ que satisfaga la fórmula. Como t en las n primeras posiciones tiene 1, por cada variable x_i se debe haber seleccionado o bien la fila y_i o bien la fila z_i, pero no las dos (porque en ese caso la columna sumaría 2). Entonces, si se ha seleccionado la fila y_i, la asignación de la variable x_i será 1; si se ha seleccionado la fila z_i, la asignación de la variable x_i será 0. Y falta comprobar que dicha asignación satisface ϕ. Sabemos que los últimos m dígitos de t son 3, es decir, cada columna suma 3. Tenemos que como máximo dos de esos 3s vienen de seleccionar g_i y h_i, pero al menos un 1 aparece en las filas superiores, es decir, que cada cláusula tiene al menos un literal que da 1. Como se cumple para todas las cláusulas, la fórmula ϕ se satisface.

En el ejemplo se nos queda la misma tabla. Solamente que esta vez tenemos el subconjunto S', es decir, una selección de filas de la tabla y queremos calcular una asignación de las variables de ϕ que satisfagan la fórmula. Para ver si la variable x_i está asignada a 1 ó 0, tenemos que ver si se ha seleccionado la fila y_i o z_i, respectivamente.

Finalmente, debemos comprobar que la reducción de $3SAT$ a SS se hace en tiempo po-

linomial. Sea ϕ la fórmula booleana en $3SAT$ con n variables y m cláusulas, el tamaño de la entrada es $n' = n + m$. Por cada variable x_i, se generan dos números y_i y z_i; y por cada cláusula c_j, se generan dos números g_j y h_j. Cada número de longitud $n + m$. Para generar cada número se tarda $O(n + m)$ y hay $n + m$ números, luego $O((n + m)^2)$. Finalmente, para generar t se tarda $O(n + m)$.

Por tanto, sumando los tiempo de cada paso:

$$O((n + m)^2) + O(n + m) = O((n + m)^2) = O(n'^2),$$

donde n' es el tamaño de la entrada original. Luego, la reducción de $3SAT$ a SS tiene un coste polinomial de $O(n^2)$, siendo n el tamaño de la fórmula.

□

2. Criptografía postcuántica

2.1. Introducción a la criptografía

La palabra criptografía tiene raíces griegas *kryptos*, que siginifica oculto, y *graphikos*, que significa escritura. Podemos decir entonces, que la criptografía es la práctica de ocultar el contenido de los mensajes. Se basa en cuatro pilares:

1. Confidencialidad: solo el destinatario autorizado puede leer el contenido. Por ejemplo, encriptando un email.
2. Integridad: el mensaje no se ha modificado. Por ejemplo, usando un hash o firma digital en un documento.
3. Autenticación: verificar la identidad del remitente o destinatario. Por ejemplo, al iniciar sesión con contraseña o certificado.
4. No repudio: evitar que alguien niegue haber enviado o recibido algo. Por ejemplo, al firmar un contrato digital.

La criptología, por otro lado, es la ciencia que estudia la criptografía. Decimos que la criptografía se encarga de diseñar métodos de cifrado/descifrado, mientras que la criptología incluye el criptoanálisis, el estudio de cómo romper esos sistemas.

En las próximas secciones utilizaremos dos términos muy comunes: cifrar y descrifrar. El cifrado es el proceso de transformar un mensaje legible (texto plano) en uno ilegible (texto cifrado) usando una clave. El descifrado hace la operación inversa utilizando la clave adecuada. El objetivo es que, aunque se intercepte el mensaje cifrado y el método en el que se ha cifrado, el intruso no pueda descifrar el mensaje sin tener la clave.

Uno de los métodos más antiguos de la criptografía es el *Caesar Cipher*, utilizado por Julio Cesar en la Antigua Roma. En sus cartas tenía mensajes como el siguiente:

exegeviqswtsvipiwxi

Parecen letras aleatorias que no pueden tener ninguna información relevante. Sin embargo, tiene un mensaje oculto, que se puede descifrar moviendo cada letra cuatro posiciones en el alfabeto. De forma que la letra a se sustuirá por la letra e; la letra b, por la f; y la letra z, por la d. Si aplicamos esta técnica al mensaje obtenemos:

atacaremosporeleste

Si el enemigo consigue la carta que contiene el mensaje oculto y suponiendo que conoce el método de cifrado, solamente tendría que transformar el mensaje usando los 26 posibles desplazamientos para conocer la intención de Julio Cesar. Es por ello que este método no es demasiado seguro.

Existen otros métodos de cifrar mensajes mediante sustitución de letras sin un orden lógico. De forma que la letra a podría sustituirse por 26 letras diferentes y la letra b por

25, puesto que no puede ser el mismo sustituto que el de *a*. Luego, la cantidad de cifrados posibles por sustitución diferentes son:

$$26 \cdot 25 \cdot 24 \cdot ... \cdot 2 \cdot 1 = 26!$$

Supongamos que Alice le mandase un mensaje a Bob encriptado con uno de estos métodos, obviamente Bob ya debería conocer la asignación de cada letra para poder leerlo. Alice podría estar tranquila de que una tercera persona, digamos Eve, va a tardar millones de años en probar todas las posibles sustituciones hasta encontrar el contenido del mensaje. Sin embargo, el estudio de la criptografía da nuevos métodos para descifrar los mensajes y Eve podría tener en cuenta la frecuencia de cada letra, las combinaciones de letras más probables y otros trucos para reducir significativamente el tiempo de descifrado.

Otro avance importante fue el cifrado de Vigenère que, en vez de usar una clave numérica (que indica el desplazamiento), usa una palabra (donde cada letra indica el desplazamiento de cada posición).

Ejemplo 2.1. Veamos un ejemplo aclarativo de Vigenère en el que queremos cifrar el mensaje:

ayudaporfavor.

Y vamos a suponer que la clave es *clave*. Como el texto a cifrar es más largo que la clave, tendremos que alargarla como *claveclavecla*. De esta forma, sabemos que la primera letra del mensaje, la *a*, se tiene que desplazar 2 posiciones (porque a=0, b=1, c=2,...), sustituyéndose por la *c*. La segunda letra del mensaje se tiene que desplazar 11 posiciones, es decir, la *y* se sustituirá por la letra *k*; y así, sucesivamente. Finalmente, se nos queda el mensaje cifrado:

cjuyerzraexzr.

■

Este método, aunque se consideró irrompible durante mucho tiempo, hoy en día se puede romper si la clave se repite con frecuencia, aplicando técnicas estadísticas más avanzadas.

Ejemplo 2.2. Un último ejemplo para esta sección será el One-Time Pad, donde cada letra o bit del mensaje se combina con la letra o bit de la clave usando una operación (suma módulo 26 para letras, o XOR para bits). Luego, la clave será completamente aleatoria, tan larga como el mensaje y sólo se podrá usar una vez.

Imaginemos que queremos cifrar el siguiente mensaje:

enviartropasalnorte.

El mensaje tiene 19 letras, luego tenemos que generar una clave aleatoria de 19 letras. Por ejemplo *xqblncmvwpqzetihdgr*. Para cifrar, transformaremos cada letra del mensaje y de la clave en un número (a=0, b=1, c=2,...) y sumaremos letra a letra módulo 26.

La primera letra del mensaje es la *e*, que se convierte en 4; y la primera letra de la clave, *x*, se convierte en 23. Al sumarlas módulo 26, $(4 + 23) \mod 26 = 27 \mod 26 = 1$, se nos queda que la primera letra del mensaje cifrado será la *b*. Este proceso se repite para todas las letras del texto plano, quedándonos el siguiente mensaje cifrado:

bdwtnfmkeqreevvuzvc.

■

Este método es muy simple, se podría ejecutar con papel y lápiz, y además, se considera absolutamente irrompible. Sin embargo, una clave tan larga como el propio texto cifrado no es práctico a gran escala.

Estos ejemplos de criptosistemas simples siguen [14].

2.2. Sistemas de clave simétrica

En los ejemplos mencionados, Alice le manda un mensaje a Bob y, para hacerlo, usa una clave secreta k para ocultar su mensaje m y convertirlo en un mensaje cifrado c. Una vez Bob recibe el mensaje tiene que usar la clave secreta k para convertir el mensaje cifrado c en la reconstrucción de m. En este proceso, tanto Bob como Alice son conocedores de la misma clave k, que deberían haberla acordado con anterioridad mediante un canal seguro.

Los métodos de cifrado que se ajustan a los ejemplos se llaman cifrados simétricos, porque Bob y Alice comparten la misma clave. Formalmente, un cifrado simétrico usa una clave k elegida de un espacio de posibles claves $\mathcal{K}$ para encriptar un mensaje m elegido de un espacio de posibles mensajes $\mathcal{M}$. El resultado de la encriptación es un texto cifrado c perteneciente a un espacio de posibles textos cifrados $\mathcal{C}$.

La encriptación es una función

$$e : \mathcal{K} \times \mathcal{M} \to \mathcal{C},$$

cuyo dominio es el conjunto de pares clave y mensaje, (k, m), que tienen rango de cifrado $\mathcal{C}$. Asimismo, la decriptación es una función

$$d : \mathcal{K} \times \mathcal{C} \to \mathcal{M}.$$

Además, queremos que la decriptación sea la función de rehacer los resultados de la encriptación, es decir,

$$d(k, e(k, m)) = m \quad \forall k \in \mathcal{K}, m \in \mathcal{M}.$$

De ahora en adelante, usaremos la clave k como un subíndice. Para cada k tendremos entonces dos funciones

$$e_k : \mathcal{M} \to \mathcal{C} \quad \text{and} \quad d_k : \mathcal{C} \to \mathcal{M}.$$

Todos los ejemplos mencionados en el apartado anterior (Caesar Cipher, cifrado de Vigenère, One-Time Pad) eran cifrados simétricos, ya que la clave era compartida por el remitente y el destinatario. Aunque hay un algoritmo que es el estándar mundial para el cifrado de datos: AES (Advanced Encryption Standard). Este método encripta un mensaje de 128 bits usando una clave de 128, 192 o 256 bits. Es un cifrado iterativo, donde se aplican cuatro operaciones un número concreto de veces.

Una de las ventajas principales de este tipo de cifrados es su velocidad, ya que son mucho más rápidos y consumen menos recursos computacionales que los sistemas de clave pública. Es por eso que suelen usarse para el cifrado de grandes cantidades de datos. Sin embargo, su principal desventaja es la necesidad de compartir la clave de forma segura antes de la comunicación. Si un atacante logra interceptar la clave, puede descifrar todos los mensajes. Este problema de distribución segura de clave es precisamente lo que motivó el desarrollo de los sistemas de clave pública.

2.3. Sistemas de clave asimétrica o pública

En un cifrado asimétrico o de clave pública sólo el destinatario del mensaje cifrado tiene la clave para descifrarlo y el resto de la información es completamente pública. Podemos decir entonces que existen dos claves: la pública (compartida para todo el mundo) y la privada (que sólo el destinatario puede ver). Cuando un mensaje se cifra con la clave pública, solamente podrá descifrarse con la clave privada.

Formalmente, un elemento k del espacio de claves $\mathcal{K}$ es un par de claves:

$$k = (k_{priv}, k_{pub}),$$

que son la clave privada y pública, respectivamente. Entonces, para cada clave pública k_{pub} existe una función de encriptación

$$e_{k_{pub}} : \mathcal{M} \to \mathcal{C}$$

y para cada clave privada k_{priv} existe una función de desencriptación

$$d_{k_{priv}} : \mathcal{C} \to \mathcal{M}$$

Y estas funciones tienen la propiedad de que si el par $k = (k_{priv}, k_{pub})$ pertenece al espacio de claves $\mathcal{K}$, entonces

$$d_{k_{priv}}(e_{k_{pub}}(m)) = m \; \forall m \in \mathcal{M}.$$

Uno de los ejemplos típicos de clave pública es el criptosistema RSA, nombrado así por sus inventores Ron Rivest, Adi Shamir y Leonard Adleman. Su seguridad se basa en la dificultad de factorizar enteros grandes. Aunque la clave pública incluye un módulo grande y un exponente cifrado, para obtener la clave privada se necesita conocer la factorización de ese módulo, algo que es inviable para números suficientemente grandes. Este algoritmo se explicará en más detalle en Sección 2.5.

Otro ejemplo importante es el protocolo de Diffie-Hellman para intercambio de claves. Permite acordar una clave secreta a través de un canal público sin compartir previamente una clave secreta. Su seguridad se basa en la dificultad de resolver el logaritmo discreto en un grupo finito.

También cabe mencionar que existen variantes basadas en el logaritmo discreto sobre curvas elípticas, que ofrecen la misma seguridad pero con claves más pequeñas.

La principal ventaja de los sistemas de clave pública es que resuelven el problema de la distribución de claves: no hace falta un canal seguro previo para acordar una clave. Sin embargo, suelen ser más lentos y computacionalmente costosos que los sistemas simétricos, por lo que en la práctica se usan juntos: la clave pública establece un canal seguro para intercambiar una clave simétrica, que luego cifra los datos de forma eficiente.

2.4. Firmas digitales

Una firma digital es un sistema criptográfico que permite verificar la autenticidad y la integridad de un mensaje o documento digital. El objetivo es simular una firma manuscrita: demostrar que un mensaje fue creado o aprobado por una persona concreta. Además, una firma digital también protege contra modificaciones no autorizadas, es decir, si el mensaje

cambia, la firma deja de ser válida.

Los sistemas de clave pública y de firma digital tienen esquemas muy similares, que incluyen claves pública y privada, pero con roles invertidos. En cifrado asimétrico, el emisor cifra con la clave pública del receptor, y solo el receptor puede descifrar con su clave privada. En firma digital, en cambio, el emisor firma con su clave privada, y cualquiera puede verificar la firma usando la clave pública del emisor.

Formalmente, un esquema de firma digital consta de tres algoritmos:

1. Generación de claves: produce un par de claves (k_{priv}, k_{pub}), donde k_{priv} se usa para firmar y k_{pub} para verificar.
2. Firma: Dado un mensaje m, la clave privada y un algoritmo de firma $Sign$, se produce una firma $s = Sign_{k_{priv}}(m)$.
3. Verificación: cualquiera con acceso a la clave pública puede comprobar la firma con un algoritmo $Verify$, evaluando si $Verify_{k_{pub}}(m, s)$ devuelve $true$.

Puesto que muchos esquemas de firma solo funcionan sobre mensajes cortos (entre 80 y 1000 bits), en la práctica se firma el resumen (digest) del mensaje, obtenido mediante una función hash criptográfica segura. De este modo, la firma es más eficiente porque se plica sobre un resumen de tamaño fijo, y se protege todo el contenido del mensaje.

Existen distintos algoritmos clásicos para implementar firmas digitales. Los algoritmos de firmas digitales RSA utiliza el mismo esquema que el cifrado RSA, pero se aplica en sentido inverso. El emisor usa su clave privada para firmar el mensaje, y el verificador comprueba la firma usando la clave pública. La seguridad se basa también en la dificultad de la factorización de enteros grandes.

Otro algoritmo típico es ElGamal Digital Signature, basado en el problema del logaritmo discreto en grupos finitos. Este algoritmo serviría como base para diseñar el algoritmo DSA (Digital Signature Algorithm), adoptado como estándar internacional en Estados Unidos. Su eficiencia fue mejorada usando el logaritmo discreto sobre curvas elípticas en la variante ECDSA.

Las firmas digitales aseguran autenticidad, integridad, no repudio y, además, cualqueira puede validar la firma sin acceso a datos secretos. Sin embargo, su seguridad depende del mantenimiento seguro de la clave privada, y de la dificultad de resolver los problemas matemáticos en los que se basan los algoritmos.

2.5. Problemas matemáticos difíciles

Los sistemas de clave pública y de firma digital que hemos mencionado en anteriores apartados (como RSA y Diffie-Hellman) se basan en ciertos problemas matemáticos considerados difíciles de resolver con los recursos computacionales actuales. La seguridad de estos algoritmos depende de que no se conozcan métodos eficientes (en tiempo polinómico) para resolver estos problemas en instancias grandes. A continuación, se describen brevemente algunos de estos problemas matemáticos difíciles:

Factorización de enteros. El problema de la factorización de enteros consiste en, dado un número grande N, determinar sus factores primos. En el caso de RSA, N se genera como el producto de dos primos grandes p y q. Entonces, calcular N es trivial, pero la factorización de N es un problema para el que no se conoce un algoritmo en tiempo polinómico.

La clave pública en RSA incluye N y un exponente de cifrado e, elegido tal que $\gcd(e, \varphi(N)) = 1$, donde $\varphi(N) = (p-1)(q-1)$ por el Teorema de Euler. Para descifrar, se necesita el exponente privado d, que cumple:

$$e \cdot d \equiv 1 \mod \varphi(N).$$

La seguridad de RSA se basa en que, sin conocer p y q, es computacionalmente inviable calcular $\varphi(N)$ y por lo tanto, el inverso d. Esto convierte la factorización de enteros en un problema fundamental: si se lograra factorizar N rápidamente, todo el sistema se rompería.

Logaritmo discreto. El problema del logaritmo discreto en un grupo finito es otro básico de la criptografía. Sea p un número primo grande y g un generador del grupo multiplicativo $\mathbb{Z}_p^*$. Dado un entero h, resolver:

$$h \equiv g^x \pmod{p}$$

significa encontrar el exponente secreto x. La exponenciación modular es computacionalmente eficiente, pero no se conoce ningún algoritmo eficiente para el logaritmo discreto cuando p es suficientemente grande.

El protocolo de intercambio de claves Diffie-Hellman utiliza esta ecuación. Las dos partes, Alice y Bob, eligen números secretos a y b, y comparten $g^a \pmod{p}$ y $g^b \pmod{p}$ públicamente. Cada uno puede entonces calcular la clave compartida:

$$K = (g^b)^a = (g^a)^b = g^{ab} \pmod{p},$$

mientras que un atacante necesita resolver el logaritmo discreto para obtener a o b.

Otro uso del problema del logaritmo discreto aparece en los sistemas de firma digital como ElGamal y DSA. En estos esquemas, el firmante elige una clave privada a y publica $A = g^a \pmod{p}$ como clave pública. Para firmar un mensaje, se utiliza un valor aleatorio k y se calculan dos componentes (S_1, S_2), donde $S_1 = g^k \pmod{p}$ y S_2 depende de a, k y el mensaje. La verificación requiere comprobar que

$$A^{S_1} \cdot S_1^{S_2} \equiv g^D \pmod{p},$$

una identidad que se basa en propiedades algebraicas del grupo multiplicativo. La seguridad del esquema reside en la dificultad de recuperar a a partir de $A = g^a \pmod{p}$, es decir, en la imposibilidad práctica de resolver el logaritmo discreto.

Logaritmo discreto elíptico. Una variante más avanzada se basa en el problema del logaritmo discreto sobre curvas elípticas. En este caso, los cálculos se hacen en el grupo de puntos de una curva elíptica definida sobre un campo finito. Dado un punto generador P y otro punto $Q = kP$, el problema consiste en hallar el entero k. Aunque la operación de multiplicar es sencilla, calcular el logaritmo discreto elíptico es muy difícil con los algoritmos conocidos. La principal ventaja de esta variante es que logra la misma seguridad con claves más pequeñas.

En resumen, la criptografía de clave pública depende de la dificultad computacional de problemas como los mencionados anteriormente. Toda la seguridad de los sistemas actuales se basa en que no se conocen algoritmos eficientes para resolverlos. Esto motiva la investigación en criptografía postcuántica, dado que la aparición de ordenadores cuánticos podría hacer que algunos de estos problemas dejen de ser intratables.

2.6. Motivación de la criptografía postcuántica

El desarrollo de la computación cuántica plantea una amenaza directa a muchos sistemas criptográficos que se usan actualmente. Mientras que la criptografía clásica se basa en problemas difíciles para ordenadores convencionales, los algoritmos cuánticos permiten resolver algunos de estos problemas de forma mucho más eficiente.

Un computador cuántico es un modelo de cómputo que se basa en las propiedades de la mecánica cuántica, como la superposición y la interferencia. En lugar de bits clásicos que son 0 y 1, utiliza qubits, que pueden representarse como una combinación lineal de los estados 0 y 1:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \text{con } |\alpha|^2 + |\beta|^2 = 1.$$

Es decir, un qubit puede ser algo así como un poco 0 y un poco 1 al mismo tiempo. A esto se le llama superposición y permite al computador cuántico procesar muchas posibilidades en paralelo.

La otra propiedad es la interferencia, que permite combinar estas superposiciones de forma que las respuestas incorrectas se cancelen y la respuesta correcta se refuerce. Gracias a la interferencia, los algoritmos cuánticos se pueden diseñar para dirigir el resultado hacia la solución deseada.

En resumen, un computador cuántico aprovecha la superposición para explorar muchas soluciones a la vez, y la interferencia para aumentar la probabilidad de encontrar la respuesta correcta al medir. Es posible que la computación cuántica no escale lo suficiente por limitaciones técnicas desconocidas, pero todavía no hay evidencia de tales límites. Por lo tanto, lo prudente es asumir que esa amenaza es real y prepararse con criptografía postcuántica, como se explica en [6].

El algoritmo de Shor. En 1994, Peter Shor presentó un algoritmo cuántico que factoriza enteros grandes de forma polinómica. Recordemos que la seguridad de RSA se basa en la dificultad de factorizar un número $N = pq$, producto de dos primos grandes. Mientras que en un ordenador clásico la mejor factorización conocida es subexponencial (el *General Number Field Sieve*), el algoritmo de Shor logra factorizar N con complejidad $O((\log N)^3)$.

El algoritmo de Shor usa la transformada cuántica de Gourier para resolver problemas de búsqueda de períodos en funciones moduladas, reduciendo la factorización a un problema de hallar el período de una función. El resultado es que, con un computador cuántico suficientemente grande, sistemas como RSA quedarían rotos.

Además, Shor generalizó su técnica para resolver el problema del logaritmo discreto, en el que se basa Diffie-Hellman y los sistemas de curva elíptica (ECC). Por tanto, con un ordenador cuántico se rompería la mayoría de los sistemas de clave pública usados actualmente.

El algoritmo de Grover. En 1996, Grover presentó otro algoritmo cuántico que reduce el número de evaluaciones necesarias en la búsqueda no estructurada de N a $\sqrt{N}$.

Para los cifrados simétricos, como AES, esto significa que un ataque de fuerza bruta sobre una clave de 128 bits, pasaría de necesitar 2^{128} operaciones en un ordenador convencional a 2^{64} operaciones en un ordenador cuántico. Este ataque no rompe el método por completo, pero obliga a aumentar el tamaño de la clave para mantener la seguridad.

Vemos que el impacto de la computación cuántica en la criptografía podría ser significativo. Mientras que los sistemas de clave simétrica pueden adaptarse relativamente fácil

aumentando el tamaño de las claves, los sistemas de clave pública actuales quedarían rotos frente a un atacante con computador cuántico.

2.7. Qué es la criptografía postcuántica

En el apartado anterior hemos explicado la importancia de investigar sobre la criptografía postcuántica, que consiste en sistemas criptográficos diseñados para resistir ataques de adversarios con computador cuántico. La criptografía postcuántica no usa qubits ni canales cuánticos, sino que sigue siendo criptografía clásica que puede implementarse en hardware y software actuales. Su seguridad se basa en problemas matemáticos que se cree que son difíciles incluso para algoritmos cuánticos conocidos, como los problemas de retículos, códigos de error, o funciones hash.

Es importante diferenciar la criptografía postcuántica de la criptografía cuántica. La criptografía cuántica, como el Quantum Key Distribution (QKD), utiliza las leyes de la física cuántica para garantizar la seguridad de la distribución de claves. Por tanto, necesita canales cuánticos para llevarse a cabo. En cambio, la criptografía postcuántica se diseña para poder reemplazar los algoritmos actuales, usando solo un procesamiento clásico.

2.8. Tipos de problemas resistentes

Para resistir todos los ataques conocidos, incluyendo variantes del algoritmo de Shor, se han estudiado algunas propuestas candidatas a reemplazar RSA y ECC en el futuro. Estas son las propuestas que han sobrevivido décadas de estudio y siguen manteniéndose como seguras. Algunas de ellas están diseñadas para cifrado (como McEliece), otras para firmas digitales y algunas permiten ambos usos.

2.8.1. Criptografía basada en códigos

La criptografía basada en códigos es una familia de sistemas criptográficos que usa códigos correctores de errores como base para proteger la información.

Un código corrector de errores es un método para transmitir información por un canal ruidoso (teléfono, internet, satélite,...), de manera que aunque algunos bits se corrompan por el camino, el receptor pueda detectarlos y corregirlos automáticamente. La idea es en lugar de mandar el mensaje sin añadidos, se manda una versión codificada más larga, que incluye redundancia.

Ejemplo 2.3. Pongamos un ejemplo famoso: el código de Hamming (7,4). Con Hamming nos referimos a que los bits de paridad (de redundancia) se colocan en las posiciones potencias de 2 y con (7,4) nos referimos a un mensaje de 4 bits que se codifica en una palabra de 7 bits. En este caso somos capaces de detectar y corregir 1 error de bit.

Supongamos un mensaje de 4 bits,

$$m = (m_1, m_2, m_3, m_4) = (1, 0, 1, 1)$$

Se transformará en 7 bits: $c = (c_1, c_2, c_3, c_4, c_5, c_6, c_7)$, donde c_1, c_2, c_4 son los bits de paridad y, el resto, son los datos originales. Tenemos por ahora que:

$$c = (\_, \_, 1, \_, 0, 1, 1)$$

Calculemos los bits de paridad usando la suma bit a bit (XOR):

- c_1 cubre posiciones 1,3,5,7 $\rightarrow c_1 = c_3 \oplus c_5 \oplus c_7 = 1 \oplus 0 \oplus 1 = 0$.
- c_2 cubre posiciones 2,3,6,7 $\rightarrow c_2 = c_3 \oplus c_6 \oplus c_7 = 1 \oplus 1 \oplus 1 = 1$.
- c_4 cubre posiciones 4,5,6,7 $\rightarrow c_4 = c_5 \oplus c_6 \oplus c_7 = 0 \oplus 1 \oplus 1 = 0$.

Así obtenemos el código transmitido:

$$c = (0, 1, 1, 0, 0, 1, 1)$$

Supongamos que en la transimisión se corrompe el bit 5, y el receptor recibe:

$$c' = (0, 1, 1, 0, 1, 1, 1)$$

El receptor recalcula las paridades con el síndrome y verifica que hay inconsistencias al verificar c_1 y c_4, lo que indica que el error se encuentra en la posición 5. Con este método puede corregir el bit 5 y recuperar el mensaje original.

■

En el ejemplo hemos usado el método de Hamming, donde el código (las posiciones de paridad, las reglas y la matriz de paridad H) es público. De esta forma, cualquiera puede codificar y descodificar, pero no sirve para cifrar. La criptografía basada en códigos introduce un secreto en el código:

- Una clave privada secreta: un código corrector de errores estructurado que el dueño sabe decodificar de forma eficiente.
- Clave pública: una matriz generadora de código disfrazada (mezclada con permutaciones y transformaciones lineales).

Desde fuera, la clave pública parece un código lineal aleatorio, para el cual la corrección de errores es NP-difícil.

El sistema McEliece fue propuesto en 1978 y es el ejemplo más famoso de criptografía basada en códigos. Como clave secreta usa un código Goppa binario y, como clave pública, una matriz generadora de código, a la que se le aplica permutación para ocultar su estructura interna.

McEliece PKC (Public Key Cryptosystem). Es un sistema de cifrado asimétrico, pero basado en códigos correctores de errores. El algoritmo se basa en ocultar un código corrector de errores (un código Goppa) tras dos transformaciones: una permutación de las columnas, que oculta la estructura del código; y una matriz aleatoria invertible, S, que resuelve las filas.

El emisor puede usar la clave pública, pero solo el receptor conoce la estructura oculta y puede corregir los errores introducidos intencionadamente para recuperar el mensaje.

Para el algoritmo McEliece se utilizan tres parámetros:

- n es la longitud del código (número total de bits del mensaje cifrado).
- t es el número máximo de errores que el código puede corregir.
- k es la dimensión del código, es decir, la longitud del mensaje original.

El proceso de generar claves trata de elegir un código corrector de errores C de distancia mínima $d \geq 2t + 1$, cuya matriz generadora es G de tamaño $k \times n$. Se generan la matriz binaria aleatoria S, que es invertible de tamaño $k \times k$ y la matriz de permutación P, de tamaño $n \times n$. Y calcula la clave pública como:

$$G_{pub} = S \cdot G \cdot P.$$

La clave pública es G_{pub} y t, mientras que la clave privada está formada por S, G, P y el algoritmo de decodificación eficiente de G (llamado D_G).

Ejemplo 2.4. Veamos un ejemplo para entender cómo funciona el algoritmo. Supongamos que trabajamos sobre $\mathbb{F}_2$, con la longitud del mensaje $k = 2$, la longitud del código $n = 4$ y que podemos corregir hasta $t = 1$ errores. Una matriz G generadora de código podría ser la siguiente:

$$G = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \in \mathbb{F}_{2\times 4}^2$$

La matriz de permutación P intercambiará las columnas 1 y 4, y las columnas 2 y 3. Será de la siguiente forma:

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \in \mathbb{F}_{4\times 4}^2$$

Y finalmente, generamos una matriz binaria aleatoria de tamaño 2×2:

$$S = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \in \mathbb{F}_{4\times 4}^2$$

Podemos calcular la clave pública $G_{pub} = S \cdot G \cdot P$, que nos quedaría:

$$G_{pub} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \in \mathbb{F}_{2\times 4}^2$$

Entonces para cifrar un mensaje $m \in \mathbb{F}_2^k$ se elige un vector de errores $z \in \mathbb{F}_2^n$ que tenga exactamente t unos y se calcula el cifrado como:

$$c = m \cdot G_{pub} \oplus z$$

donde $\oplus$ es la suma bit a bit (XOR).

Supongamos en nuestro ejemplo que queremos cifrar el mensaje:

$$m = \begin{pmatrix} 1 & 0 \end{pmatrix} \in \mathbb{F}_{1\times 2}^2$$

Elegimos un vector de errores con 1 uno:

$$z = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \in \mathbb{F}_{1\times 4}^2$$

El mensaje cifrado será calculado como $c = m \cdot G_{pub} \oplus z$, es decir:

$$m \cdot G_{pub} = \begin{pmatrix} 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$$

$$c = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \oplus \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix}$$

Para descifrar c, el receptor aplica la inversa de la permutación:

$$c' = c \cdot P^{-1} = m \cdot S \cdot G \oplus z \cdot P^{-1}$$

Como conoce el código original G y su decodificador D_G, puede corregir los errores del vector c' y obtener mS. Le queda aplicar la inversa de S para recuperar el mensaje original:

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

En nuestro ejemplo, debemos multiplicar el mensaje cifrado c por la inversa de la permutación P, que es su traspuesta. Nos queda:

$$c \cdot P^{-1} = \begin{pmatrix} 1 & 0 & 0 & 1 \end{pmatrix} \oplus \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix}$$

Como el receptor conoce G, buscamos una combinación lineal de filas de esta matriz que esté a distancia 1 del resultado anterior. Es decir, probamos:

$$\begin{aligned} m = \begin{pmatrix} 1 & 0 \end{pmatrix} &\implies mG = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \\ m = \begin{pmatrix} 0 & 1 \end{pmatrix} &\implies mG = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix} \\ m = \begin{pmatrix} 1 & 1 \end{pmatrix} &\implies mG = c \cdot P^{-1} = \begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix} \end{aligned}$$

El decodificador nos devuelve:

$$m \cdot S = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Finalmente, aplicamos S^{-1} para recuperar el mensaje original:

$$S^{-1} = S = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Nos queda entonces que:

$$m = \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot S^{-1} = \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

Efectivamente, éste era nuestro mensaje original.

■

Como hemos visto, atacar McEliece significa descifrar sin conocer el código original, solamente conociendo G_{pub}. Aunque el código G tiene estructura (porque es de Goppa), el

público ve una versión camuflada, por lo que es muy difícil deducir cómo corregir los errores.

En resumen, se puede ver el problema McEliece como: dado un $G_{pub} \in \{0,1\}^{k\times n}$, un vector $c \in \{0,1\}^n$ y un entero t, encuentra el único $m \in \{0,1\}^k$ tal que:

$$\text{peso de } (mG_{pub} - c) = t$$

El criptosistema GLN, que estudiaremos en detalle en el Capítulo 3, también se inspira en estas ideas. Igual que McEliece oculta un código corrector de errores tras transformaciones lineales, GLN oculta un problema de suma de subconjuntos (MSSP/WMSSP) tras operaciones aritméticas modulares. En ambos casos, la seguridad proviene de un problema de corrección/dificultad algorítmica considerado NP-duro (esto es, al menos tan difícil como cualquier problema en NP), mientras que la clave privada permite una decodificación eficiente.

2.8.2. Criptografía basada el Subset Sum Problem (SSP).

La idea mochila en criptografía se basa en aprovechar la dificicultad del Subset Sum Problem (SSP) como base de seguridad. Este problema, demostrado NP-completo en el E.j. 1.29, consiste en determinar si un número entero puede puede expresarse como la suma de un subconjunto de una lista dada de enteros.

En el contexto de la criptografía, la idea original es la siguiente: el mensaje se representa como un vector binario que selecciona ciertos elementos de la secuencia, y el texto cifrado se obtiene como la suma ponderada de esos elementos. Para un observador externo, invertir esta operación (es decir, recuperar el mensaje a partir de una suma) equivale a resolver una instancia de SSP, un problema difícil en el caso general.

De esta forma se obtiene un esquema de cifrado de clave pública:

- La clave pública es una secuencia de enteros aparentemente aleatoria, sobre la cual resolver SSP resulta difícil.
- La clave privada es información adicional que da estructura a la secuencia y permite al receptor descifrar el mensaje eficientemente.

Merkle-Hellman. El primer criptosistema de mochila fue propuesto por Merkle y Hellman en 1978. Su idea era utilizar una secuencia súpercreciente como trampa secreta.

Una secuencia $(a_1, \ldots, a_n)$ es súpercreciente si cada término es mayor que la suma de todos los anteriores. Por ejemplo, $(2, 3, 7, 15, 31)$ es súpercreciente. Con esta propiedad, resolver el problema de la suma de subconjuntos resulta trivial usando un algoritmo voraz: basta restar el mayor elemento posible y repetir el proceso hasta llegar a cero.

En el esquema de Merkle-Hellman, la clave privada es una secuencia súpercreciente, que permite descifrar fácilmente. La clave pública se obtiene a partir de una transformación modular que oculta la estructura: cada elemento de la secuencia se multiplica por un entero secreto w y se reduce módulo m, donde w es coprimo con m. El resultado es una secuencia que ya no parece súpercreciente y frente a la cual el problema de la suma de subconjuntos vuelve a ser difícil.

Para cifrar, el emisor representa el mensaje como un vector binario que selecciona ciertos elementos de la clave pública, y el texto cifrado es la suma de esos elementos. Para descifrar,

el receptor multiplica el cifrado por el inverso de w módulo m, con lo que recupera una suma en la secuencia privada súpercreciente.

Sin embargo, el sistema fue roto por Shamir (1984), quien mostró que la baja densidad de la clave pública podía explotarse con técnicas de programación lineal. Desde entonces, Merkle-Hellman dejó de considerarse seguro, pero su propuesta abrió la puerta a una línea de investigación en criptografía basada en problemas de mochila.

Ejemplo 2.5. Como ejemplo vamos a elegir una secuencia súpercreciente de 6 números:

$$w = (2, 3, 7, 14, 30, 57)$$

De parámetros extra necesitamos un módulo M mayor que la suma total: $M = 200 > 2 + 3 + 7 + 14 + 30 + 57 = 113$ y número a coprimo con M, por ejemplo, $a = 31$.

Para generar la clave pública calcularemos

$$b_i = a \cdot w_i \pmod{M}.$$

Calculamos,

$$\begin{array}{c|c} b\_1 = 31 \cdot 2 \pmod{200} = 62 & b_2 = 31 \cdot 3 \pmod{200} = 93 \\ b\_3 = 31 \cdot 7 \pmod{200} = 17 & b_4 = 31 \cdot 14 \pmod{200} = 34 \\ b\_5 = 31 \cdot 30 \pmod{200} = 130 & b_6 = 31 \cdot 57 \pmod{200} = 167. \end{array}$$

La clave pública nos queda

$$b = (62, 93, 17, 34, 130, 167).$$

Es el momento de cifrar el mensaje, supongamos el vector binario

$$m = (1, 0, 1, 0, 1, 1).$$

Esto significa que elegimos b_1, b_3, b_5, b_6 y calculamos el cifrado que envía el emisor

$$c = 62 + 17 + 130 + 167 = 376.$$

El receptor conoce la clave privada a y M. Para descifrar el mensaje debe calcular el inverso modular de a módulo M

$$a^{-1} \equiv 31^{-1} \pmod{200}.$$

Que con el algoritmo de extendido Euclides,

$$31 \cdot 71 - 200 \cdot 11 = 1.$$

Así que $a^{-1} = 71$. Ahora calcula,

$$\begin{aligned} c' &= a^{-1} \cdot c \pmod{M} \\ &= 71 \cdot 376 \pmod{200} \\ &= 71 \cdot 176 \pmod{200} \\ &= 12496 \pmod{200} \\ &= 96. \end{aligned}$$

Para recuperar el mensaje debemos expresar $c' = 96$ como suma de la secuencia privada $w = (2, 3, 7, 14, 30, 57)$. Usamos el algoritmo voraz:

1. El mayor que no excede 104 es 57 $\rightarrow$ queda 39.
2. El mayor que no excede 39 es 30 $\rightarrow$ queda 9.
3. El mayor que no excede 9 es 7 $\rightarrow$ queda 2.
4. El mayor que no excede 2 es 2 $\rightarrow$ queda 0.

Hemos usado $57, 30, 14, 3$, lo que corresponde a posiciones (w_1, w_3, w_5, w_6). En binario,

$$m = (1, 0, 1, 0, 1, 1).$$

Y hemos recuperado el mensaje original.

■

El criptosistema GLN utiliza un esquema de mochila: representa el mensaje como una suma de subconjuntos, pero lo hace en una versión sofisticada, usando el problema de la suma de subconjuntos modular ponderada (WMSSP).

A diferencia del esquema de Merkle-Hellman, que se basaba en secuencias súpercrecientes fácilmente explotables, GLN usa únicamente aritmética modular sobre enteros. Su mecanismo de descifrado usa la identidad de Bézout y el algoritmo de Euclides extendido truncado, eso lo hace mucho más robusto frente a ataques estructurales y corrige las debilidades del sistema de Merkle-Hellman.

2.9. Estado actual y estandariazación

Aunque un sistema criptográfico esté matemáticamente bien fundamentado, su seguridad práctica depende de muchos factores del mundo real: dispositivos físicos, entornos operativos, implementaciones concretas y usuarios humanos. En muchas ocasiones, los modelos matemáticos con los que se diseñan criptografía no siempre reflejan cómo funcionan las cosas en el mundo físico. Por ejemplo, existen los ataques por canales laterales (side-channel attacks), donde el atacante no rompe las matemáticas del algoritmo, sino que observa los efectos físicos del dispositivo, como cuánto tarda en calcular o cuánta energía consume, para deducir claves secretas.

Por tanto, la seguridad no solo debe evaluarse a nivel teórico, sino también desde una perspectiva práctica, incluyendo resistencia frente a ataques físicos y facilidad de implementación segura.

Por otro lado, no basta con que un sistema sea seguro: todas las personas y dispositivos deben ponerse de acuerdo sobre qué algoritmos usar y cómo usarlos exactamente. Eso se consigue con la estandarización, un proceso para definir normas comunes y necesario para que distintos sistemas puedan interoperar de forma segura.

2.9.1. Estado actual

Los estándares de seguridad modernos se fundamentan en la dificultad computacional de ciertos problemas matemáticos. En este capítulo se han descrito algunos criptosistemas actuales importantes: RSA, ECC y AES.

La seguridad de estos criptosistemas depende de la dificultad computacional frente al mejor ataque clásico y realista que se conoce hoy en día. Se mide con el número de operaciones básicas que debe realizar un atacante para romper el sistema usando este ataque. Este nivel se expresa habitualmente en bits de seguridad.

Formalmente, si el mejor ataque conocido contra un criptosistema necesita aproximadamente 2^n bits operaciones elementales, entonces se dice que el sistema ofrece n bits de seguridad. Por ejemplo, un ataque que necesita del orden de 2^{128} operaciones corresponde a 128 bits de seguridad.

El Instituto Nacional de Estándares y Tecnología de EE. UU. (NIST) mantiene guías oficiales sobre niveles mínimos de seguridad aceptables para cifrado, firmas, KEM y otros mecanismos criptográficos. Una de ellas establece, según el período de tiempo y según el uso, qué niveles de seguridad deben considerarse válidos.

Security strength		Through 2030	2031 and beyond
Strength	Applying / Processing		
< 112	Applying	Disallowed	Disallowed
	Processing	Legacy-use	Legacy-use
112	Applying	Acceptable	Disallowed
	Processing	Acceptable	Legacy-use
128	Applying/Processing	Acceptable	Acceptable
192	Applying/Processing	Acceptable	Acceptable
256	Applying/Processing	Acceptable	Acceptable

Tabla 2.1.: Aceptación de fuerza de seguridad para aplicar y procesar protección criptográfica.

La Tabla 2.1, extraída de la documentación oficial del NIST en [5], resume los requisitos. Se distinguen dos escenarios:

- *Applying*: se refiere a proteger datos actuales, como cifrar mensajes nuevos.
- *Processing*: se refiere a descifrar datos cifrados en el pasado. En este caso no se exige el mismo nivel de seguridad porque el algoritmo pudo ser seguro cuando se cifraron esos datos, pero ya no lo es para su uso futuro. Sin embargo, es necesario usarlo para recuperar la información almacenada.

El NIST marca los mínimos:

- Para cifrar datos nuevos hasta 2030, un sistema debe ofrecer ≥ 112 bits de seguridad.

- A partir de 2031, los sistemas considerados seguros deben ofrecer ≥ 128 bits de seguridad.
- Los algoritmos con seguridad < 112 bits están prohibidos para cifrar datos nuevos, pero pueden usarse solo para descifrar datos antiguos (*Legacy-use*).

2.9.2. Estandarización postcuántica

Para anticipar el impacto de los ordenadores cuánticos, las organizaciones internacionales ya han reconocido la urgencia de prepararse para ataques cuánticos. En 2017, el NIST inició un proceso competitivo internacional para seleccionar algoritmos criptográficos postcuánticos que serán estandarizados en el futuro [10]. Tras varias rondas de evaluación, en 2022 el NIST anunció los primeros algoritmos seleccionados para estandarización:

- CRYSTALS-Kyber, para cifrado de clave pública e intercambio de claves, basado en retículos.
- CRYSTALS-Dilithium, para firmas digitales, también basado en retículos.
- FALCON, otro esquema de firmas digitales basado en retículos, con parámetros más compactos.
- SPHINCS+, un esquema de firmas basado en funciones hash, elegido como alternativa por no depender de retículos.

En 2023 se publicaron los borradores de los primeros estándares oficiales y se sigue trabajando en un segundo grupo de candidatos para hacer el portafolio de algoritmos postcuánticos más diverso.

La transición hacia criptografía postcuántica será progresiva y exigente. Existen ya sistemas candidatos sólidos, pero aún se necesita tiempo para afinar parámetros, garantizar implementaciones robustas y adaptar los sistemas actuales. Aun así, lo que está claro es que el futuro de la criptografía depende de anticiparse al impacto de la computación cuántica.

3. Criptosistema GLN

El criptosistema GLN (Gómez Torrecillas-Lobillo-Navarro) es una propuesta reciente dentro del campo de la criptografía postcuántica, que combina ideas de la criptografía basada en códigos (McEliece) y el problema de mochila, utilizando solamente aritmética entera. Su seguridad se basa en una versión del problema de la suma de subconjuntos (E.j. 1.29), conocido como el problema de la suma modular ponderada de subconjuntos (WMSSP, por sus siglas en inglés), y logra construir un esquema de cifrado de clave pública seguro frente a ataques cuánticos, eficiente en generación de claves y con operaciones simples.

En este capítulo se describirá el funcionamiento del criptosistema GLN, incluyendo su motivación matemática, los algoritmos implicados en cada fase (generación de claves, cifrado y descifrado) y la demostración de su corrección.

Toda la información de este capítulo se ha obtenido a partir del paper oficial publicado por los autores del criptosistema [13].

3.1. Motivación y contexto

El primer criptosistema de clave pública basado en el Subset Sum Problem (SSP) fue el esquema de Merkle-Hellman (Subsección 2.8.2). En él, la clave privada consistía en una secuencia súpercreciente, que permite resolver instancias del SSP de forma eficiente mediante un algoritmo voraz. La clave pública se construía aplicando a la secuencia una transformación modular y multiplicativa para esconder la estructura. Sin embargo, pronto se demostró que esta transformación no eliminaba por completo los patrones internos: se podía recuperar la secuencia súpercreciente usando reducción de retículos, es decir, se podía descifrar el sistema.

El criptosistema GLN retoma el SSP, pero añade modificaciones para evitar las debilidades que se han atacado anteriormente. En lugar de trabajar directamente con sumas simples, lo hace sobre sumas módulo un número g, con coeficientes ponderados y parámetros seleccionados cuidadosamente. De esta forma se construyen instancias mucho más resistentes del problema conocido como WMSSP (Weighted Modular Subset Sum Problem), para el que no se han encontrado algoritmos clásicos ni cuánticos eficientes que lo resuelvan.

Al igual que en los sistemas de McEliece/Niederreiter, la idea es que:

- Para un atacante externo, resolver la instancia del problema subyacente es difícil.
- Para el receptor (que contiene la clave privada), existe un procedimiento que permite invertir la operación de forma eficiente.

El procedimiento para invertir la operación no es un decodificador de códigos (como en McEliece), sino una reconstrucción aritmética basada en la identidad de Bézout y en el algoritmo de Euclides extendido truncado (TrEEA), que permite recuperar los parámetros internos y, con ello, el mensaje original.

El objetivo de este capítulo es demostrar la corrección del esquema: cifrar y descifrar con GLN recupera siempre el mensaje. Para su uso práctico, se sigue la construcción estándar KEM-DEM:

- El esquema GLN encapsula la clave de sesión k (KEM).
- Con la clave de sesión se protege la información mediante un cifrado simétrico conocido (DEM).

En términos de implementación, GLN sólo utiliza aritmética entera básica (suma, productos y un algoritmo euclídeo), lo que facilita su eficiente y entendimiento. Además, admite alfabetos no binarios de manera natural, lo que mejora su seguridad.

3.2. Descripción general del sistema

El criptosistema GLN es un sistema de clave pública basado en el problema NP-completo de la suma de subconjuntos (SSP), pero reformulado con una estructura inspirada en el esquema clásico de McEliece. Se trata de un sistema híbrido compuesto por dos capas: un esquema de cifrado de clave pública (PKE, Public Key Encryption) y un esquema de intercambio de claves (KEM, Key Encapsulation Mechanism).

En la práctica, el criptosistema GLN se emplea como mecanismo de encapsulación de claves (KEM). Dado que esta capa no es necesaria para entender el funcionamiento interno del sistema, su construcción detallada se ha trasladado al Apéndice A, donde se describe cómo se usaría GLN en un esquema KEM completo.

El esquema de cifrado de clave pública (PKE), Π, se define con los siguientes componentes:

1. Parámetros:
   - z es el tamaño del alfabeto $\{0, 1, ..., z-1\}$ de los mensajes.
   - n es el tamaño del texto plano.
   - t es el peso del texto plano, es decir, el número de componentes no nulas en el vector del mensaje.
2. La clave pública es una lista de n enteros no negativos: $(t_1, ..., t_n)$.
3. La clave privada es una lista de $n+2$ enteros positivos: $(p_1, ..., p_n, g, u)$.
4. El texto plano es un vector $e = (e_1, ..., e_n)$ de longitud n cuyas coordenadas toman valores del alfabeto $\{0, 1, ..., z-1\}$ y contiene exactamente t entradas no nulas.
5. El texto cifrado es un par de enteros (c_1, c_2). En el contexto del KEM, el texto cifrado transmitido se denota como (c_0, c_1) que son equivalentes a (c_1, c_2) del PKE.

De manera general, el sistema GLN funciona de la siguiente manera:

1. Generación de claves (`KeyGen`): este proceso es muy eficiente, ya que se basa exclusivamente en aritmética de enteros. Se selecciona una lista de n enteros p_i (preferiblemente primos para simplificar los cálculos), un entero grande g (que debe cumplir ciertas restricciones para garantizar la unicidad de la solución), y un entero aleatorio u. A partir de estos, se calculan los inversos modulares h_i de p_i con respecto a g, y finalmente, se calculan los enteros t_i como

$$t_i \equiv h_i - u \pmod{g}.$$

2. Cifrado (`Encrypt`): para cifrar un mensaje, que es un vector $e = (e_1, ..., e_n)$ con un número fijo t de componentes no nulas, el emisor utiliza la clave pública $(t_1, ..., t_n)$ para cifrar. Calcula dos valores: c_1 es la suma ponderada de las componentes no nulas del mensaje por los elementos de la clave pública,

$$c_1 = \sum_{i=1}^{n} e_i \cdot t_i.$$

   Por otro lado, c_2 es la suma de las componentes no nulas,

$$c_2 = \sum_{i=1}^{n} e_i.$$

   El texto cifrado es el par (c_1, c_2). Es crucial que se envíe c_2 junto con c_1 para permitir la recuperación del texto plano.

3. Descifrado (`Decrypt`): el receptor utiliza la clave privada $(p_1, ..., p_n, g, u)$ y el texto cifrado (c_1, c_2). Primero, calcula un valor s usando c_1, c_2 y u,

$$s \equiv c_1 + u \cdot c_2 \pmod{g}.$$

   Luego, aplica una versión truncada del algoritmo de Euclides extendido (TrEEA) con g y s como entrada para obtener el par (λ, ω). Este par (λ, ω) está relacionado con los elementos p_i del mensaje original. Finalmente, al encontrar los p_k que comparten un factor primo con λ y resolver un sistema de congruencias, se recuperan las componentes no nulas del texto plano.

Este criptosistema destaca por su resistencia contra amenazas cuánticas. Además, a diferencia de los criptosistemas basados en mochila típicos, esta propuesta es adaptable para operar con alfabetos no binarios, lo que mejora su seguridad contra ataques de baja densidad cuando se usan alfabetos suficientemente grandes.

3.3. Sistema de cifrado determinista (PKE)

El criptosistema se define como un esquema PKE determinista:

$$\Pi = (\text{PubKeys}, \text{PrivKeys}, \text{Plaintexts}, \text{Ciphertexts}, \text{KeyGen}, \text{Encrypt}, \text{Decrypt})$$

Su seguridad se basa en la tratabilidad del Problema de la Suma de Subconjuntos Modular Ponderada (WMSSP). En el apartado anterior ya hemos explicado el algoritmo de forma general, ahora detallemos los pasos.

3.3.1. Generación de claves

El algoritmo de generación de claves `KeyGen` produce un par de claves: una clave pública para el cifrado y una privada para el descifrado. Este proceso se basa exclusivamente en aritmética de enteros, lo que contribuye a su eficiencia.

En primer lugar, se seleccionan aleatoriamente una lista ordenada

$$p = (p_1, ..., p_n)$$

formada por n enteros positivos, mayores que $z - 1$ y relativamente primos entre sí. La idea es que estos enteros formen parte de la clave secreta. Aunque no es estrictamente obligatorio, la implementación del criptosistema se simplifica enormemente cuando los p_i son primos entre sí, como veremos más adelante.

Por otro lado, se selecciona aleatoriamente un entero positivo g que debe ser relativamente primo a cada p_i, para $i = 1, ..., n$. Además, g debe satisfacer dos condiciones de tamaño cruciales para la unicidad y corrección del descifrado:

- $g > 4\omega_{\max}^2$
- $g \geq 4\lambda_{\max}^2$

Donde,

$$\lambda_{\max} = p_{k_1} \cdots p_{k_t}, \quad \omega_{\max} = (z-1) \sum_{j=1}^{t} \frac{\lambda_{\max}}{p_{k_j}}.$$

Los $p_{k_1}, \ldots, p_{k_t}$ son los t enteros más grandes de la lista p.

El entero g actúa como el módulo principal en el criptosistema modular, por lo que las condiciones sobre su tamaño son fundamentales. Estas condiciones implican $g > \lambda_{\max}\omega_{\max}$, lo que, según el Teorema 3.16 garantiza que existe a lo sumo un par admisible (λ, ω) que satisface la ecuación clave:

$$s\lambda \equiv \omega \pmod{g}.$$

Además, estas condiciones aseguran que el TrEEA (Subsección 3.4.1) aplicado durante el descifrado devuelva directamente los valores λ y ω.

Para cada $i = 1, ..., n$ se calcula el entero positivo h_i menor que g tal que $p_i h_i \equiv 1 \pmod{g}$. Estos h_i son los inversos modulares de los p_i respecto a g y son la clave para resolver el problema de la suma de subconjuntos, convirtiéndolo de un problema NP-duro a uno eficientemente resoluble una vez que g y p_i sean conocidos.

También se selecciona el entero u, un valor secreto aleatorio cuya función principal es esconder la relación directa entre los elementos de la clave pública t_i y los inversos modulares h_i. Si la clave pública estuviera formada directamente por los h_i, un atacante podría intentar explotar propiedades de las relaciones entre p_i y h_i para inferir g y, por tanto, la clave privada. Al introducir u y definir

$$t_i \equiv h_i - u \pmod{g},$$

se ocultan los h_i, haciendo más difícil para un atacante reconstruir la clave privada o g a partir de la clave pública.

Tenemos entonces que la clave pública se forma por la lista de enteros $(t_1, \ldots, t_n)$, mientras que la clave privada se forma por la lista $(p_1, \ldots, p_n, g, u)$.

Ejemplo 3.1. A continuación, presentamos un ejemplo de generación de claves usando este algoritmo. Supongamos los siguientes parámetros:

- $n = 6$ es la longitud del texto plano.
- $t = 2$ es el número de componentes no nulas del texto plano.

- $z = 5$ es el tamaño del alfabeto.

El primer paso es seleccionar una lista ordenada de n enteros positivos que sean mayores que $z - 1 = 4$ y relativamente primos entre sí. Es más conveniente seleccionar números primos para simplificar cálculos y acelerar el proceso de generación de claves:

$$p = (5, 7, 11, 13, 17, 19).$$

A continuación, indentficamos que los t valores más grandes en la lista p. Este caso, $t = 2$, y los dos valores más grandes de la lista son son $p_{k_1} = 19$ y $p_{k_2} = 17$, con lo que podemos calcular los valores auxiliares

$$\lambda_{max} = 19 \cdot 17 = 323, \quad \omega_{max} = (z-1)\sum_{j=1}^{t} \frac{\lambda_{\max}}{p_{n_j}} = 4 \cdot (\frac{323}{19} + \frac{323}{17}) = 144.$$

Se selecciona aleatoriamente un entero positivo g que sea coprimo a cada p_i para $i = 1, ..., n$ y que cumpla con las condiciones de tamaño:

1. $g > 4\omega_{\max}^2 = 4 \cdot 144^2 = 82944$
2. $g \geq 4\lambda_{\max}^2 = 4 \cdot 323^2 = 417316$

Un buen candidato para g que cumpla estas condiciones y sea comprimos con los p_i (podemos usar un número primo para simplificar) es $g = 417331$.

Para cada $i = 1, ..., n$, se calcula el entero positivo h_i tal que $p_i \cdot h_i \equiv 1 \pmod{g}$. Esto se realiza usando el Algoritmo Extendido de Euclides para encontrar el inverso modular.

$$\begin{array}{l|l} h_1 = 5^{-1} \pmod{417331} = 333865 & h_2 = 7^{-1} \pmod{417331} = 238475 \\ h_3 = 11^{-1} \pmod{417331} = 189696 & h_4 = 13^{-1} \pmod{417331} = 160512 \\ h_5 = 17^{-1} \pmod{417331} = 220940 & h_6 = 19^{-1} \pmod{417331} = 109824. \end{array}$$

Se selecciona aleatoriamente un entero positivo u menor que g y diferente a los enteros $h_1, ..., h_n$. Este valor u es crucial para la seguridad del sistema, ya que evita los ataques basados en conocimiento de los h_i originales. Tomemos, por ejemplo, $u = 123456$.

Para cada $i = 1, ..., n$, se calcula el entero positivo t_i tal que $t_i \equiv h_i - u \pmod{g}$. Estos valores t_i formarán la clave pública.

$$\begin{aligned} t_1 &= (h_1 - u) \pmod{g} = (333865 - 123456) \pmod{417331} = 210409 \\ t_2 &= (h_2 - u) \pmod{g} = (238475 - 123456) \pmod{417331} = 115019 \\ t_3 &= (h_3 - u) \pmod{g} = (189696 - 123456) \pmod{417331} = 66240 \\ t_4 &= (h_4 - u) \pmod{g} = (160512 - 123456) \pmod{417331} = 37056 \\ t_5 &= (h_5 - u) \pmod{g} = (220940 - 123456) \pmod{417331} = 97484 \\ t_6 &= (h_6 - u) \pmod{g} = (109824 - 123456) \pmod{417331} = 109824. \end{aligned}$$

- La clave pública está formada por los enteros $t_1, ..., t_n$.

$$PubKeys = (210409, 115019, 66240, 37056, 97484, 109824).$$

- La clave privada está formada por los enteros $p_1, ..., p_n, g$ y u.

$$PrivKeys = (5, 7, 11, 13, 17, 19, 417331, 123456).$$

■

3.3.2. Cifrado

El algoritmo `Ecrypt` toma un texto plano y utiliza la clave pública para generar el texto cifrado. El texto plano es un vector $e = (e_1, ..., e_n)$ de longitud n. Sus coordenadas e_i toman valores del alfabeto $\{0, ..., z-1\}$, y exactamente t coordenadas son no nulas.

El primer paso es calcular c_1 y c_2 como

$$c_1 = \sum_{i=1}^{n} e_i t_i, \quad c_2 = \sum_{i=1}^{n} e_i$$

La interpretación de c_1 es la suma ponderada de las componentes del mensaje por los elementos de la clave pública. Por otro lado, c_2 es simplemente la suma de las componentes del mensaje (que coincide con el peso t, si el alfabeto es binario, o con la suma real de los valores no nulos si el alfabeto es mayor). El envío de c_2 es esencial para el descifrado, ya que permite al receptor deshacer el efecto de u.

Como resultado obtenemos el texto cifrado, que es el par de enteros (c_1, c_2).

Ejemplo 3.2. Retomamos el ejemplo asumiendo que el paso de generación de claves ha sido completado. Recordemos los parámetros utilizados:

- $n = 6$ es la longitud del texto plano.
- $t = 2$ es el número de componentes no nulas del texto plano.
- $z = 5$ es el tamaño del alfabeto $\{0, 1, 2, 3, 4\}$.

Además, sabemos que para cifrar el mensaje necesitamos usar la clave pública, que es:

$$PubKeys = (210409, 115019, 66240, 37056, 97484, 109824).$$

El algoritmo de cifrado toma como entrada un vector de texto plano $e = (e_1, ..., e_n)$. Este vector debe tener una longitud n, sus coordenadas deben tomar valores del alfabeto $\{0, 1, 2, 3, 4\}$ y exactamente t de ellas deben ser no nulas. Para este ejemplo, elijamos el mensaje

$$e = (0, 3, 0, 4, 0, 0).$$

A continuación, se calcula c_1 como la suma de los productos de cada componente e_i del texto plano por su correspondiente t_i de la clave pública,

$$c_1 = \sum_{i=1}^{n} e_i t_i.$$

Sustituyendo los valores de nuestro ejemplo:

$$c_1 = e_2 \cdot t_2 + e_4 \cdot t_4 = 3 \cdot 115019 + 4 \cdot 37056 = 493281.$$

Se calcula c_2 como la suma de todas las componentes del texto plano,

$$c_2 = \sum_{i=1}^{n} e_i = 3 + 4 = 7.$$

Finalmente, el texto cifrado es el par de enteros (c_1, c_2), es decir,

$$Ciphertext = (493281, 7).$$

■

3.3.3. Descifrado

El algoritmo `Decrypt` utiliza la clave privada y el texto cifrado para recuperar el texto plano. El receptor calcula el entero positivo s (menor que g) tal que

$$s \equiv c_1 + u \cdot c_2 \pmod{g}.$$

La idea es cancelar los términos introducidos durante el cifrado, quedándonos con una combinación lineal de los valores h_i asociados a las posiciones no nulas del mensaje. Dicho de otra forma, transforma el difícil de la suma de subconjuntos Ponderada moduar (WMSSP), en un Problema de la Suma de Subconjuntos Modular (MSSP) que se basa en los h_i y sí puede resolverse conociendo la clave privada.

El siguiente paso es utilizar el Algoritmo Extendido de Euclides Truncado (TrEEA) con entrada (g, s). Este algoritmo produce (v, r) como salida y, bajo las condiciones de g, se cumple que

$$(v, r) = \frac{\lambda}{\gcd(\lambda, \omega)}, \frac{\omega}{\gcd(\lambda, \omega)}.$$

Básicamente, el algoritmo TrEEA actúa como un mecanismo para calcular (λ, ω), usados para codificar el mensaje.

Conocido λ, el receptor del mensaje puede identificar qué posiciones del vector son no nulas: basta con comprobar, para cada p_i de la clave privada, si divide a λ. Los p_i que aparecen como factores de λ corresponden a las posiciones del mensaje original con valores distintos de cero.

Finalmente, una vez localizadas estas posiciones, los valores concretos e_j se recuperan resolviendo sencillas congruencias módulo los p_{k_j} correspondientes. Estas congruencias provienen de la definición de ω y determinan de manera única los coeficientes del mensaje.

Ejemplo 3.3. Continuando con el ejemplo anterior, procedemos a aplicar los pasos del descifrado. Recordemos:

- El texto cifrado es $(c_1, c_2) = (493281, 7)$.
- La clave privada es $(p_1, ..., p_n, g, u) = (5, 7, 11, 13, 17, 19, 417331, 123456)$.

El primer paso es calcular un entero s tal que $s \equiv c_1 + u \cdot c_2 \pmod{g}$, tenemos

$$s = 493281 + 123456 \cdot 7 \mod 417331 = 1357473 \mod 417331 = 105480.$$

El TrEEA produce v y r que corresponden a λ y ω divididos por $\gcd(\lambda, \omega)$. Dado que los p_i son primos, $\gcd(\lambda, \omega)$ será 1, lo que significa que el algoritmo directamente nos dará λ y ω. Además, se busca que $|\lambda|$ y $|\omega|$ sean pequeños, específicamente menores que $\sqrt{g}/2 = \sqrt{417331}/2 \approx 323$.

Aplicamos el Algoritmo Extendido de Euclides a $(g, s) = (417331, 105480)$:

Cociente	r_0	r_1	v_0	v_1	(r, v)
(Inicio)	417331	105480	0	1	-
3	105480	100891	1	-3	(100891,-3)
1	100891	4589	-3	4	(4589,4)
21	4589	4522	4	-87	(4522,-87)
1	4522	67	-87	91	(67,91)
67	67	33	91	-6184	(33,-6184)
2	33	1	-6184	12459	(1,12459)

Buscamos el par (r, v) donde $|r|, |v| < 323$. El par $(67, 91)$ cumple estas condiciones. Confirmamos que

$$\begin{aligned} s \cdot v &\equiv r \pmod{g} \\ 105480 \cdot 91 &\equiv 67 \pmod{417331} \\ 9598680 &\equiv 67 \pmod{417331}. \end{aligned}$$

La anterior ecuación es correcta porque $9598680 - 29 \cdot 417331 = 67$. Por lo tanto, los valores obtenidos son: $\omega = 67$ y $\lambda = 91$.

Ahora debemos encontrar los $p_{k_1}, ..., p_{k_t}$, los t enteros p_i de la clave privada P que tienen un máximo común divisor distinto de 1 con $\lambda = 91$.

$$\begin{array}{c|c|c} \gcd(5, 91) = 1 & \gcd(7, 91) = 7 & \gcd(11, 91) = 1 \\ \gcd(13, 91) = 13 & \gcd(17, 91) = 1 & \gcd(19, 91) = 1. \end{array}$$

Hemos encontrado dos p_i con gcd distinto de 1: $p_{k_1} = p_2 = 7$ y $p_{k_2} = p_4 = 13$. Esto indica que el texto plano original tenía dos componentes no nulas ($t = 2$), que corresponden a las posiciones 2 y 4.

Ahora, calculamos el producto de los p_k encontrados:

$$\gamma = (p_2 \cdot p_4)/\lambda = (7 \cdot 13)/91 = 91/91 = 1.$$

Actualizamos $\lambda = p_2 \cdot p_4 = 91$ y $\omega = \omega \cdot \gamma = 67 \cdot 1 = 67$. En este caso, como $\gamma = 1$, los valores de λ y ω se mantienen.

Ahora debemos resolver las congruencias para las componentes no nulas del texto plano, $e_2 = e_{k_1}$ y $e_4 = e_{k_2}$, utilizando la fórmula $(\lambda/p_{k_i}) \cdot e_{k_i} \equiv \omega \pmod{p)_{k_i}}$ con $0 < e_{k_i} < p_{k_i}$. Para e_{k_1} usando $p_{k_1} = 7$:

$$\begin{aligned} (\lambda/p_{k_1}) \cdot e_{k_1} &\equiv \omega \pmod{p_{k_1}} \\ (91/7) \cdot e_2 &\equiv 67 \pmod{7} \\ 13 \cdot e_2 &\equiv 67 \pmod{7} \\ -1 \cdot e_2 &\equiv 4 \pmod{7} \\ e_2 &\equiv -4 \pmod{7} \\ e_2 &= 3. \end{aligned}$$

Para e_{k_2} usando $p_{k_2} = 13$:

$$
\begin{aligned}
(\lambda/p_{k_2}) \cdot e_{k_2} &\equiv \omega \pmod{p_{k_2}} \\
(91/13) \cdot e_4 &\equiv 67 \pmod{13} \\
7 \cdot e_4 &\equiv 67 \pmod{13} \\
7 \cdot e_4 &\equiv 2 \pmod{13} \\
7^{-1} \cdot 7 \cdot e_4 &\equiv 7^{-1} \cdot 2 \pmod{13} \\
2 \cdot 7 \cdot e_4 &\equiv 2 \cdot 2 \pmod{13} \\
e_4 &\equiv 4 \pmod{13} \\
e_4 &= 4.
\end{aligned}
$$

El texto plano e es un vector de longitud $n = 6$, donde las componentes e_2 y e_4 son 3 y 4 respectivamente, y las demás son 0. Así, el texto plano descifrado es

$$e = (0, 3, 0, 4, 0, 0).$$

Este resultados es consistente con el texto plano que hemos utilizado en el ejemplo de cifrado anterior, confirmando que hemos descifrado el mensaje con éxito.

■

3.4. Demostración de corrección del criptosistema

Antes de comenzar la demostración de la corrección del criptosistema, presentamos algunas definiciones y resultados básicos de teoría de números.

Definición 3.4. Un entero a divide a otro entero b, y escribiremos $a|b$, si existe un entero k tal que $b = ak$.

Definición 3.5. El máximo común divisor de dos enteros a, b denotado $\gcd(a, b)$, es el mayor entero positivo que divide simultáneamente a a y b.

Definición 3.6. Dos enteros a, b se dicen primos relativos o coprimos cuando $\gcd(a, b) = 1$.

Teorema 3.7 (Identidad de Bézout)**.** *La identidad de Bézout establece que, dados dos enteros a, b, existen siempre enteros u, v tales que*

$$au + bv = \gcd(a, b).$$

Esta identidad es la base del algoritmo de Euclides extendido (EEA), que sirve para calcular el máximo común divisor de dos enteros y, además, devuelve explícitamente un par de coeficientes (u, v) que satisfacen la identidad de Bézout.

En criptografía, esta propiedad se utiliza para calcular inversos modulares: si $\gcd(a, v) = 1$, entonces el coeficiente u obtenido del EEA verifica

$$au \equiv 1 \pmod{v}.$$

En el resto del capítulo utilizaremos la siguiente notación:

- n: longitud del texto plano (cantidad de elementos del vector).
- t: peso del texto plano (número de posiciones no nulas).
- z: tamaño del alfabeto $\{0, 1, ..., z-1\}$.
- g: entero positivo grande utilizado en la generación de claves, coprimo con todos los p_i.
- $p_1, .., p_n$: enteros positivos mayores que $z-1$ y mutuamente coprimos, que forman parte de la clave privada.
- h_i: inversos modulares de los p_i módulo g.
- u: entero aleatorio menor que g, usado para enmascarar la clave pública.
- t_i: enteros que forman parte de la clave pública, obtenidos a partir de los h_i y de u.

3.4.1. Algoritmo de Euclides extendido truncado (TrEEA)

El algoritmo de Euclides extendido permite expresar el máximo común divisor de dos enteros a, b como una combinación lineal de ellos. En particular, devuelve enteros (u, v) tales que $ua + vb = \gcd(a, b)$.

Algorithm 1 Algoritmo de Euclides extendido (EEA)

Require: Dos enteros a, b con $a \geq b > 0$
Ensure: $\gcd(a, b)$ y coeficientes (u, v) tales que $ua + vb = \gcd(a, b)$

$r_0 \leftarrow a,\ r_1 \leftarrow b$
$u_0 \leftarrow 1,\ v_0 \leftarrow 0$
$u_1 \leftarrow 0,\ v_1 \leftarrow 1$
for $i = 1, 2, \ldots$ **do**
  $q_i \leftarrow \lfloor r_{i-1}/r_i \rfloor$
  $r_{i+1} \leftarrow r_{i-1} - q_i r_i$
  $u_{i+1} \leftarrow u_{i-1} - q_i u_i$
  $v_{i+1} \leftarrow v_{i-1} - q_i v_i$
  if $r_{i+1} = 0$ **then**
    salir del bucle
  end if
end for
return (r_i, u_i, v_i) ▷ $r_i = \gcd(a, b)$ y $u_i a + v_i b = r_i$

En nuestro criptosistema se emplea una variante truncada de este algoritmo, denominada TrEEA (Truncated Extended Euclidean Algorithm). Su objetivo no es calcular el máximo común divisor, sino proporcionar una par de enteros (v, r) a partir de una entrada (g, s) que satisfagan una relación lineal del tipo $ug + vs = r$, donde r es un resto suficientemente pequeño. Es decir, el TrEEA se detiene antes, en el momento en que el resto r es lo bastante pequeño con relación con g.

Algorithm 2 Algoritmo de Euclides extendido truncado (TrEEA)

Require: $a, b \in \mathbb{Z}$ con $a \geq b > 0$
Ensure: $v, r \in \mathbb{Z}$
1: $r_0 \leftarrow a,\ r_1 \leftarrow b$
2: $v_0 \leftarrow 0,\ v_1 \leftarrow 1$
3: $c \leftarrow r_0$ quo r_1
4: $r \leftarrow r_0 - cr_1,\ r_0 \leftarrow r_1,\ r_1 \leftarrow r$
5: $v \leftarrow -c,\ v_0 \leftarrow v_1,\ v_1 \leftarrow v$
6: **while** $r^2 \geq a$ **do**
7: $\quad c \leftarrow r_0$ quo r_1
8: $\quad r \leftarrow r_0 - cr_1,\ r_0 \leftarrow r_1,\ r_1 \leftarrow r$
9: $\quad v \leftarrow v_0 - cv_1,\ v_0 \leftarrow v_1,\ v_1 \leftarrow v$
10: **end while**
11: **return** (v_1, r_1)

Además, en el análisis usaremos dos propiedades fundamentales:

Lema 3.8 (Relación lineal)**.** *Los valores devueltos cumplen*

$$u_I g + v_I s = r_I$$

para ciertos coeficientes (u_I, v_I) *y un resto* r_I*.*

Demostración. Queremos ver que en cada iteración del EEA (y, por tanto, también del TrEEA) se cumple la relación

$$u_i g + v_i s = r_i$$

Lo haremos por inducción sobre i. El caso base es la inicialización en la que:

$$r_0 = g, \quad r_1 = s,$$

con

$$(u_0, v_0) = (1, 0), \quad (u_1, v_1) = (0, 1.)$$

Claramente,

$$u_0 g + v_0 s = 1 \cdot g + 0 \cdot s = g = r_0,$$
$$u_1 g + v_1 s = 0 \cdot g + 1 \cdot s = s = r_1.$$

Supongamos que para ciertos índices $i - 1$ e i se cumple

$$u_{i-1} g + v_{i-1} s = r_{i-1}, \quad u_i g + v_i s = r_i.$$

Por definición del algoritmo

$$r_{i+1} = r_{i-1} - q_i r_i, \quad u_{i+1} = u_{i-1} - q_i u_i, \quad v_{i+1} = v_{i-1} - q_i v_i.$$

Sustituyendo la hipótesis de inducción,

$$\begin{aligned} u_{i+1}g + v_{i+1}s &= (u_{i-1} - q_i u_i)g + (v_{i-1} - q_i v_i)s = \\ &= (u_{i-1}g + v_{i-1}s) - q_i(u_i g + v_i s) \\ &= r_{i-1} - q_i r_i = r_{i+1}. \end{aligned}$$

Por inducción, la relación

$$u_{i-1}g + v_{i-1}s = r_{i-1}$$

se cumple para todos los i. En particular, cuando el algoritmo truncado se detiene en el índice I, obtenemos la identidad que afirma el lema. □

Lema 3.9. *En el Algoritmo de Euclides extendido truncado (TrEEA), si se detiene en el índice I porque $r_I < \sqrt{g}$, entonces el coeficiente v_I satisface*

$$|v_I| \leq \sqrt{g}.$$

Demostración. Recordemos que en el Algoritmo de Euclides extendido se construyen sucesiones de enteros (r_i, u_i, v_i) tales que, para todo i,

$$u_i g + v_i s = r_i,$$

y que los restos cumplen $r_{i-1} = q_i r_i + r_{i+1}$ con $0 \leq r_{i+1} < r_i$.

De esta relación se deduce, por inducción, que cada resto r_i puede expresarse como una combinación lineal de g y s con coeficientes (u_i, v_i). Además, los cocientes q_i controlan la actualización de los v_i mediante la regla

$$v_{i+1} = v_{i-1} - q_i v_i.$$

Un hecho fundamental del análisis del EEA es que los valores $|v_i|$ crecen a medida que decrecen los restos r_i, de modo que en la iteración donde r_I se hace estrictamente menor que $\sqrt{g}$ se cumple la cota

$$|v_I| \leq \frac{g}{r_{I-1}}.$$

En efecto, esta desigualdad puede obtenerse de la identidad $u_i g + v_i s = r_i$: tomando valores absolutos y observando que $r_{i-1} \geq \sqrt{g}$ (pues el algoritmo se detiene precisamente cuando $r_I < \sqrt{g}$), se deduce que

$$|v_I| \leq \frac{g}{r_{I-1}} \leq \frac{g}{\sqrt{g}} = \sqrt{g}.$$

□

3.4.2. Problemas combinatorios

El funcionamiento del criptosistema GLN se apoya en la dificultad de resolver ciertos problemas conocidos por ser NP-duros. En particular, la seguridad se basa en dos versiones del Subset Sum Problem (SSP) : el Modular Subset Sum Problem (MSSP) y el Weighted Modular Subset Sum Problem (WMSSP).

El Subset Sum Problem es NP-completo, como vimos en Teorema 1.36 a través de la reducción desde 3SAT. El criptosistema de Merkle-Hellman (Subsección 2.8.2) fue la primera propuesta en basar su seguridad en SSP, aunque más tarde resultó vulnerable a ataques estructurales (Shamir).

Definición 3.10 (Modular Subset Sum Problem)**.** Dados enteros positivos $p_1, ..., p_n$ coprimos entre sí y un módulo g coprimo con todos los p_i anteriores, definimos para cada p_i con $i = 1, ..., n$ un único inverso h_i módulo g, que existe por la identidad de Bézout.

Sea s un entero positivo, el Modular Subset Sum Problem consiste en determinar si existen únicos índices distintos $1 \leq k_1 < ... < k_t \leq n$ y enteros $0 < e_1, ..., e_t < z$ tales que:

$$s \equiv \sum_{j=1}^{t} e_j h_{k_j} \pmod{g}.$$

Este problema aparece en GLN porque el cifrado se forma como una suma modular de los h_i correspondientes a las posiciones no nulas del mensaje. Resolver el MSSP sin información adicional equivale a romper el cifrado, pero el receptor que conoce la clave privada puede transformar esa suma modular en un par (λ, ω) y descifrar.

Para reforzar la seguridad frente a ciertos ataques, el criptosistema GLN introduce un parámetro secreto u. En lugar de publicar directamente los inversos h_i, se publica la secuencia disfrazada

$$t_i \equiv h_i - u \pmod{g}.$$

Con esta modificación, el cifrado resulta ser

$$s' = \sum_{i=1}^{n} t_i e_i = \sum_{i=1}^{n} h_i e_i - u \sum_{i=1}^{n} e_i.$$

Para descifrar, el receptor necesita conocer dos valores:

$$s = \sum_{i=1}^{n} h_i e_i, \quad w = \sum_{i=1}^{n} e_i,$$

donde w es el peso del mensaje (número de posiciones no nulas). Por eso el emisor transmite junto con el criptograma, el valor de w.

De este modo, el problema duro al que se enfrenta un adversario externo es el WMSSP.

Definición 3.11 (Weighted Modular Subset Sum Problem)**.** Dados enteros c_1, c_2, determinar si existen coeficientes $0 < e_j < z$ y un subconjunto de índices $1 \leq k_1 < \dots k_t \leq n$ tales que

$$c_1 \equiv \sum_{j=1}^{t} e_j t_{k_j} \pmod{g}, \quad c_2 = \sum_{j=1}^{t} e_j.$$

Esta variante es más robusta que MSSP al añadir la ecuación de peso, lo que hace el problema más intratable.

3.4.3. Construcción algebraica

Para justificar que el descifrado del criptosistema GLN recupera siempre el mensaje original, necesitamos traducir el problema de reconstruir el texto plano a un lenguaje algebraico. Re-

cordemos que, al cifrar, lo que se transmite son un par de enteros (c_1, c_2) que corresponden a combinaciones lineales de los valores del mensaje con los parámetros de la clave pública. Desde el punto de vista del receptor, descifrar consiste en identificar qué posiciones del vector $(e_1, \ldots, e_n)$ eran no nulas y con qué valores.

El reto está en que los enteros (c_1, c_2) no dan directamente los e_j, sino que aparecen enmascarados en una suma modular. Para deshacer esta suma, introducimos una representación intermedia: los pares admisibles (λ, ω):

- λ es el producto de los enteros privados p_i correspondientes a las posiciones no nulas del mensaje. Por tanto, λ codifica qué posiciones del vector forman parte del mensaje.
- ω combina los enteros privados p_i con los valores e_j, de manera que almacena la información de los coeficientes.

Con esta construcción, el proceso de descifrado se reduce en demostrar que:

- Existencia: el par (λ, ω) asociado al mensaje siempre satisface una cierta congruencia clave.
- Unicidad: bajo condiciones adecuadas sobre g, este par es único, lo que evita ambigüedad.
- Cálculo: usando el TrEEA, el receptor puede calcular efectivamente (λ, ω) y, a partir de ahí, recuperar los e_j.

Definición 3.12 (Par admisible)**.** Un par de enteros (λ, ω) se dice admisible con respecto a $\{n, t, z, g, p_1, ..., p_n\}$ si existen enteros $0 < e_1, ..., e_t < z$ y $1 \leq k_1 < ... < k_t \leq n$ tales que

$$\lambda = p_{k_1} \cdot \ldots \cdot p_{k_t}, \quad w = \sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}}.$$

En lugar de buscar directamente los e_j y las posiciones k_j del mensaje cifrado, buscamos este par (λ, ω).

Proposición 3.13. *Sea el par de enteros (λ, ω) admisible, entonces se cumple que*

$$\gcd(\lambda, \omega) = \prod_{i=1}^{t} \gcd(e_i, p_{k_i}).$$

Demostración. Demostraremos la igualdad en dos pasos.

(i) Demostración de $\gcd(\lambda, \omega) \mid \prod_{i=1}^{t} \gcd(e_i, p_{k_i})$. Supongamos que un primo (o potencia de primo) p divide a $\gcd(\lambda, \omega)$. Entonces $p \mid \lambda$ y $p \mid \omega$.

Como $\lambda = p_{k_1} \cdots p_{k_t}$ y los p_{k_i} son coprimos entre sí, debe existir un único índice j tal que $p \mid p_{k_j}$. Escribamos

$$\omega = \sum_{i=1}^{t} e_i \frac{\lambda}{p_{k_i}} = e_j \frac{\lambda}{p_{k_j}} + \sum_{i \neq j} e_i \frac{\lambda}{p_{k_i}}.$$

Observamos que, para $i \neq j$, los términos λ / p_{k_i} son múltiplos de p_{k_j}, y por tanto también de p. De ahí se deduce que

$$\omega \equiv e_j \frac{\lambda}{p_{k_j}} \pmod{p}.$$

Pero como $p \mid \omega$, se sigue que $p \mid e_j \frac{\lambda}{p_{k_j}}$. Ahora bien, como p no puede dividir a λ / p_{k_j} (recordemos que p divide a p_{k_j} y los p_{k_i} son coprimos), necesariamente $p \mid e_j$. Concluimos que $p \mid \gcd(e_j, p_{k_j})$.

Por tanto, todo divisor primo de $\gcd(\lambda, \omega)$ divide a alguno de los factores $\gcd(e_i, p_{k_i})$, lo que prueba esta primera inclusión.

(ii) Demostración de $\prod_{i=1}^{t} \gcd(e_i, p_{k_i}) \mid \gcd(\lambda, \omega)$. Sea $d_j = \gcd(e_j, p_{k_j})$ para algún j. Entonces $d_j \mid p_{k_j}$ y $d_j \mid e_j$. De aquí se sigue inmediatamente que

$$d_j \mid p_{k_j} \quad \Rightarrow \quad d_j \mid \lambda,$$

y también

$$d_j \mid e_j \quad \Rightarrow \quad d_j \mid \sum_{i=1}^{t} e_i \frac{\lambda}{p_{k_i}} = \omega.$$

Por tanto, cada d_j divide a $\gcd(\lambda, \omega)$, y en consecuencia su producto también lo hace.

Como ambas inclusiones están probadas, se concluye que

$$\gcd(\lambda, \omega) = \prod_{i=1}^{t} \gcd(e_i, p_{k_i}),$$

tal como se afirmaba. □

Una vez establecido que los pares admisibles están bien definidos y no pierden información, el siguiente paso es mostrar la conexión entre un par admisible y el proceso de cifrado. En particular, veremos que todo par admisible (λ, ω) asociado a un mensaje cumple una congruencia clave con el valor transmitido s.

Teorema 3.14 (Existencia)**.** *Los enteros $0 < e_1, \ldots, e_t < z$ y $1 \leq k_1 < \ldots < k_t \leq n$ son solución del MSSP si, y solo si, el par admisible (λ, ω) asociado cumple*

$$s\lambda \equiv \omega \pmod{g}.$$

Demostración. La demostración se divide en dos partes.

(i) Implicación directa. Supongamos que los enteros e_i y k_i para $i = 1, \ldots, t$ son solución del MSSP, es decir,

$$s \equiv \sum_{j=1}^{t} e_j h_{k_j} \pmod{g}.$$

Sea (λ, ω) el par admisible correspondiente. Entonces,

$$s\lambda \equiv \Big(\sum_{j=1}^{t} e_j h_{k_j} \Big) \lambda \pmod{g}.$$

Ahora bien, como $h_{k_j} p_{k_j} \equiv 1 \pmod{g}$, se tiene

$$h_{k_j} \lambda = h_{k_j} p_{k_j} \cdot \frac{\lambda}{p_{k_j}} \equiv \frac{\lambda}{p_{k_j}} \pmod{g}.$$

Sustituyendo en la suma:

$$s\lambda \equiv \sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}} \equiv \omega \pmod{g}.$$

Así se obtiene la congruencia buscada.

(ii) Implicación recíproca. Supongamos ahora que (λ, ω) es un par admisible que satisface

$$s\lambda \equiv \omega \pmod{g}.$$

Como $\lambda = p_{k_1} \cdots p_{k_t}$ y cada p_{k_j} es coprimo con g, también λ lo es, y por tanto existe su inverso módulo g, que es precisamente

$$\lambda^{-1} \equiv h_{k_1} \cdots h_{k_t} \pmod{g}.$$

Multiplicando la congruencia por λ^{-1} obtenemos:

$$s \equiv \omega\lambda^{-1} \pmod{g}.$$

Sustituyendo la definición de ω:

$$s \equiv \Big(\sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}} \Big) \lambda^{-1} \equiv \sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}} \lambda^{-1} \pmod{g}.$$

Finalmente, como $\frac{\lambda}{p_{k_j}} \lambda^{-1} \equiv h_{k_j} \pmod{g}$, se concluye que

$$s \equiv \sum_{j=1}^{t} e_j h_{k_j} \pmod{g},$$

lo cual demuestra que los e_j y k_j constituyen una solución del MSSP.

Con ambas implicaciones probadas, queda demostrado el teorema. □

Una vez explicada la ecuación clave que caracteriza los pares admisibles, debemos acotar sus posibles valores. Para ello introduciremos las cantidades λ_{max} y ω_{max}, que garantizan que la solución del descifrado sea única. Denotaremos por

$$\lambda_{max} = p_{n_1} \dots p_{n_t}, \quad \omega_{max} = (z-1) \sum_{j=1}^{t} \frac{\lambda_{max}}{p_{n_j}}.$$

Donde $p_{n_1}, \dots, p_{n_t}$ son los t elementos más grandes de la clave pública $p_1, \dots, p_n$. Tendremos entonces que cada par admisible (λ, ω) está acotado superiormente por $(\lambda_{max}, \omega_{max})$.

Antes de probar la unicidad de la solución, es necesario asegurarnos de que la ecuación clave no se satisface de manera trivial. Es decir, tenemos que descartar el caso $s = 0$, que invalidaría la recuperación del mensaje. Para ello, aplicaremos la condición del siguiente lema.

Lema 3.15. *Sea (λ, ω) un par admisible que satisface la ecuación*

$$s\lambda \equiv \omega \pmod{g}.$$

Si $g > \omega_{\max}$, entonces necesariamente $s \neq 0$.

Demostración. Procedamos por contradicción. Supongamos que $s = 0$. Sustituyendo en la congruencia obtenemos

$$\omega \equiv 0 \pmod{g},$$

lo que equivale a decir que $g \mid \omega$.

Por definición del par admisible,

$$\omega = \sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}},$$

donde $0 < e_j < z$ para cada $j = 1, \ldots, t$, y $\lambda / p_{k_j} \in \mathbb{Z}^+$ ya que $\lambda = p_{k_1} \cdots p_{k_t}$. De aquí se deduce que $\omega > 0$.

Además, por construcción, $\omega \leq \omega_{\max}$. En consecuencia,

$$0 < \omega \leq \omega_{\max} < g.$$

Esto contradice el hecho de que $g \mid \omega$, ya que ningún múltiplo positivo de g puede ser estrictamente menor que g.

Por tanto, la hipótesis $s = 0$ es imposible y debe cumplirse $s \neq 0$. □

El siguiente paso es asegurarnos de que la ecuación clave no puede ser satisfecha por más de un par distinto. Es decir, necesitamos garantizar la unicidad de la solución admisible.

Teorema 3.16 (Unicidad). *Si $g > \lambda_{\max}\, \omega_{\max}$, entonces existe a lo sumo un par admisible (λ, ω) que satisfaga*

$$s\lambda \equiv \omega \pmod{g}.$$

Demostración. Supongamos, por contradicción, que existen dos pares admisibles distintos (λ, ω) y (λ', ω') tales que

$$s\lambda \equiv \omega \pmod{g} \qquad \text{y} \qquad s\lambda' \equiv \omega' \pmod{g}.$$

Como λ y λ' son coprimos con g, ambas son invertibles módulo g, de modo que

$$s \equiv \omega\, \lambda^{-1} \equiv \omega'\, (\lambda')^{-1} \pmod{g} \implies \omega\, \lambda' \equiv \omega'\, \lambda \pmod{g}.$$

Por definición de las cotas, $0 < \omega \leq \omega_{\max}$ y $0 < \lambda' \leq \lambda_{\max}$, y análogamente $0 < \omega' \leq \omega_{\max}$ y $0 < \lambda \leq \lambda_{\max}$. Por tanto,

$$0 < \omega\lambda',\ \omega'\lambda \ \leq\ \omega_{\max}\lambda_{\max} \ <\ g,$$

donde la última desigualdad usa la hipótesis $g > \lambda_{\max}\omega_{\max}$. En consecuencia, la congruencia $\omega\lambda' \equiv \omega'\lambda \pmod{g}$ implica en realidad la igualdad en enteros:

$$\omega\lambda' \ =\ \omega'\lambda.$$

Sea $d = \gcd(\lambda, \omega)$ y $d' = \gcd(\lambda', \omega')$, y escribamos

$$\lambda = d\, A, \quad \omega = d\, B, \qquad \lambda' = d'\, A', \quad \omega' = d'\, B',$$

con $\gcd(A, B) = \gcd(A', B') = 1$ por construcción. La igualdad $(*)$ se reescribe como

$$(dB)(d'A') \;=\; (d'B')(dA) \;\Longrightarrow\; A\,B' \;=\; A'\,B.$$

Como $\gcd(A, B) = 1$, se deduce que $A \mid A'$. Simétricamente, de $\gcd(A', B') = 1$ se obtiene $A' \mid A$. Por tanto,

$$A = A' \quad \text{y entonces} \quad B = B'.$$

En consecuencia,

$$\frac{\lambda}{\gcd(\lambda, \omega)} \;=\; \frac{\lambda'}{\gcd(\lambda', \omega')} \quad \text{y} \quad \frac{\omega}{\gcd(\lambda, \omega)} \;=\; \frac{\omega'}{\gcd(\lambda', \omega')}.$$

Bajo la elección estándar de parámetros de GLN (en particular, $p_i > z - 1$ y, si además los p_i son primos), la Proposición 3.13 garantiza

$$\gcd(\lambda, \omega) = \prod_{j=1}^{t} \gcd(e_j, p_{k_j}) = 1 \quad \text{y} \quad \gcd(\lambda', \omega') = 1,$$

ya que $0 < e_j < z < p_{k_j}$. Aplicando esto en la igualdad anterior concluimos

$$\lambda = \lambda' \quad \text{y} \quad \omega = \omega',$$

lo que contradice que los pares fueran distintos. Por tanto, existe a lo sumo un par admisible que satisface la ecuación clave. □

Teorema 3.17 (Cálculo). *Sea (λ, ω) un par admisible que satisface la ecuación $s\lambda \equiv \omega \pmod{g}$. Supongamos que*

$$g \geq 4\lambda_{\max}^2 \qquad y \qquad g > 4\omega_{\max}^2.$$

Si (v, r) es la salida del TrEEA con entrada (g, s), entonces

$$v = \frac{\lambda}{\gcd(\lambda, \omega)}, \qquad r = \frac{\omega}{\gcd(\lambda, \omega)}.$$

Demostración. Sea $(r_i, u_i, v_i)_{0 \leq i \leq l+1}$ la sucesión generada por el EEA aplicado a (g, s). Definamos I como el índice en que se cumple $r_I^2 < g$ y $r_{I-1}^2 \geq g$; por construcción del TrEEA, la salida es $(v, r) = (v_I, r_I)$.

Definimos ahora

$$a = \frac{v_I}{\gcd(v_I, \lambda)}, \qquad b = \frac{\lambda}{\gcd(v_I, \lambda)}.$$

Por definición $b > 0$ y $\gcd(a, b) = 1$, además a tiene el mismo signo que v_I. De las hipótesis y de los lemas previos sabemos:

$$|a| \leq |v_I| \leq \sqrt{g}, \qquad |b| \leq \lambda \leq \frac{\sqrt{g}}{2}.$$

Por el lema de la relación lineal del EEA, se cumple

$$u_I g + v_I s = r_I.$$

Multiplicando por b:

$$(bu_I)g + (bv_I)s = br_I. \tag{3.1}$$

Por otro lado, de la ecuación clave $s\lambda \equiv \omega \pmod{g}$, existe un entero κ tal que

$$\lambda s + \kappa g = \omega.$$

Multiplicando por a:

$$(a\lambda)s + (a\kappa)g = a\omega. \tag{3.2}$$

Restando (3.1) y (3.2) y usando que $a\lambda = bv_I$, obtenemos

$$(a\kappa - bu_I)g = a\omega - br_I. \tag{3.3}$$

Analicemos el valor absoluto del lado derecho:

$$|a\omega - br_I| \leq \max\{\, |a\omega|, |br_I| \,\}.$$

- Si domina $|a\omega|$: como $|\omega| \leq \omega_{\max} < \frac{\sqrt{g}}{2}$ y $|a| \leq \sqrt{g}$, se cumple $|a\omega| < g$.
- Si domina $|br_I|$: como $r_I < \sqrt{g}$ y $|b| \leq \frac{\sqrt{g}}{2}$, se cumple $|br_I| < g$.

En ambos casos, $|a\omega - br_I| < g$. Pero en (3.3) el lado izquierdo es múltiplo de g. La única posibilidad es que sea nulo, es decir:

$$a\kappa = bu_I \qquad \text{y} \qquad a\omega = br_I. \tag{3.4}$$

De la segunda igualdad en (3.4) deducimos

$$r_I = \frac{a}{b}\,\omega.$$

Como $b \mid \lambda$ y $b \mid \omega$, se sigue que $b \mid \gcd(\lambda, \omega)$ y, por tanto,

$$r_I = \frac{\omega}{\gcd(\lambda, \omega)}. \tag{3.5}$$

Análogamente, de la definición de a y b obtenemos

$$v_I = \frac{a}{b}\,\lambda.$$

Usando de nuevo que $b \mid \gcd(\lambda, \omega)$, se concluye

$$v_I = \frac{\lambda}{\gcd(\lambda, \omega)}. \tag{3.6}$$

Por (3.5) y (3.6), la salida del TrEEA coincide exactamente con

$$(v, r) = \left(\frac{\lambda}{\gcd(\lambda, \omega)},\ \frac{\omega}{\gcd(\lambda, \omega)} \right),$$

lo que completa la demostración. □

Teorema 3.18 (Corrección del criptosistema GLN). *Sea $e = (e_1, \dots, e_n)$ un mensaje con $0 < e_j < z$ y soporte $\{k_1, \dots, k_t\} \subseteq \{1, \dots, n\}$. Sean $p_1, \dots, p_n$ enteros positivos coprimos entre sí, cada uno de ellos coprimo con g, y $p_i > z - 1$ para todo i. Definimos*

$$\lambda = \prod_{j=1}^{t} p_{k_j}, \qquad \omega = \sum_{j=1}^{t} e_j \frac{\lambda}{p_{k_j}}.$$

Si s es el valor obtenido en el cifrado y se cumplen las cotas

$$g > \lambda_{\max}\, \omega_{\max}, \qquad g \geq 4\lambda_{\max}^2, \qquad g > 4\omega_{\max}^2,$$

entonces el algoritmo de descifrado recupera exactamente el mensaje e a partir del criptograma.

Demostración. Para la demostración usaremos todos los resultados anteriores.

Por el Teorema 3.14, el par admisible (λ, ω) asociado al mensaje e satisface

$$s\lambda \equiv \omega \pmod{g}.$$

Además, de $g > 4\omega_{\max}^2$ (y por tanto $g > \omega_{\max}$) se deduce por el Lema 3.15 de no trivialidad que $s \neq 0$.

Bajo la cota $g > \lambda_{\max}\, \omega_{\max}$, el Teorema 3.16 garantiza que existe a lo sumo un par admisible que verifica la ecuación clave. En particular, el par asociado al mensaje e es el único compatible con s.

Aplicando el TrEEA a (g, s) y usando el Teorema 3.17, se obtiene como salida (v, r) con

$$v = \frac{\lambda}{d}, \qquad r = \frac{\omega}{d}, \quad \text{donde } d = \gcd(\lambda, \omega).$$

Es decir, el receptor obtiene el par $(A, B) := (v, r) = \big(\lambda/d,\ \omega/d\big)$.

Escribimos, para cada $j = 1, \dots, t$,

$$g_j := \gcd(e_j, p_{k_j}), \qquad \tilde{p}_{k_j} := \frac{p_{k_j}}{g_j}, \qquad \tilde{e}_j := \frac{e_j}{g_j}.$$

Por la Proposición 3.13,

$$d = \gcd(\lambda, \omega) = \prod_{j=1}^{t} g_j, \qquad A = \frac{\lambda}{d} = \prod_{j=1}^{t} \tilde{p}_{k_j}, \qquad B = \frac{\omega}{d} = \sum_{j=1}^{t} \tilde{e}_j \frac{A}{\tilde{p}_{k_j}},$$

con $\gcd(\tilde{e}_j, \tilde{p}_{k_j}) = 1$ para todo j. Como los p_i son coprimos entre sí, también lo son sus divisores $\tilde{p}_{k_j}$ y, además, cada $\tilde{p}_{k_j}$ divide únicamente a p_{k_j} y a ningún otro p_i.

El receptor conoce la clave privada $\{p_i\}$. Por tanto, a partir de A identifica el soporte del mensaje del modo siguiente:

$$\text{para cada } i, \quad t_i := \gcd(A, p_i) = \begin{cases} \tilde{p}_{k_j} & \text{si } i = k_j \text{ para algún } j, \\ 1 & \text{en caso contrario.} \end{cases}$$

De este modo se recupera exactamente el conjunto de índices no nulos $\{k_1, \dots, k_t\}$.

Fijado un índice k_j del soporte, consideremos la identidad

$$B = \sum_{\ell=1}^{t} \tilde{e}_\ell \, \frac{A}{\tilde{p}_{k_\ell}}.$$

Reduciendo módulo $\tilde{p}_{k_j}$ y usando que, para $\ell \neq j$, el valor $A/\tilde{p}_{k_\ell}$ es múltiplo de $\tilde{p}_{k_j}$ (porque los factores son coprimos entre sí), obtenemos

$$B \equiv \tilde{e}_j \, \frac{A}{\tilde{p}_{k_j}} \pmod{\tilde{p}_{k_j}}.$$

Como $\gcd\left(\frac{A}{\tilde{p}_{k_j}},\, \tilde{p}_{k_j}\right) = 1$, el inverso $\left(\frac{A}{\tilde{p}_{k_j}}\right)^{-1}$ existe módulo $\tilde{p}_{k_j}$, y por consiguiente

$$\tilde{e}_j \equiv B \left(\frac{A}{\tilde{p}_{k_j}}\right)^{-1} \pmod{\tilde{p}_{k_j}}.$$

Dado que $0 < \tilde{e}_j < \tilde{p}_{k_j}$, esta congruencia determina de forma única el entero $\tilde{e}_j$.

Finalmente, como p_{k_j} y $\tilde{p}_{k_j}$ son conocidos por el receptor, se recupera

$$g_j = \frac{p_{k_j}}{\tilde{p}_{k_j}} \qquad \text{y, por tanto,} \qquad e_j = g_j \, \tilde{e}_j.$$

Observemos que $0 < e_j < z$ por hipótesis, luego no hay ambigüedad por reducción modular.

Hemos recuperado exactamente el soporte $\{k_1, \ldots, k_t\}$ y los valores e_{k_j} del mensaje. Para los índices restantes se tiene $e_i = 0$. Por tanto, el descifrado reconstruye el vector e original. Esto prueba la corrección del esquema. □

4. Ataques al sistema

En el capítulo anterior hemos visto que el criptosistema GLN se basa en la dificultad del Weighted Modular Subset Sum Problem (WMSSP), una variante del problema clásico Subset Sum Problem (SSP). Como se vio en Teorema 1.36, el SSP es NP-completo y fue la base del criptosistema Merkle-Hellman, que fue atacado con éxito posteriormente.

Los ataques más conocidos a los criptosistemas basado en el SSP se dividen en dos tipos de categorías:

- Ataques combinatorios, que se dividen a su vez en dos ataques diferentes:
  - Ataques la clave privada: buscan recuperar los parámetros secretos a partir de la clave pública.
  - Ataque por fuerza bruta: buscan romper el esquema tratando de adivinar cuáles son las posiciones no nulas del mensaje hasta convertir la instancia en un SSP clásico.
- Ataques al problema SSP: tratan de romper el esquema aplicando algoritmos generales, especialmente mediante ataques de baja densidad, que permiten convertir el problema en uno de retículos resoluble.

El primer ataque explota posibles debilidades estructurales en la generación de claves, el segundo ataque trata de jugar con los parámetros hasta crear una instancia del problema atacable fácilmente, mientras que los terceros dependen de la dificultad algorítmica de las instancias del problema matemático en que se fundamenta la seguridad.

En el caso del GLN, se han incluido medidas específicas que mitigan los tres tipos de ataques: se incluye el parámetro secreto u en la construcción de la clave pública para evitar los primeros ataques, y se eligen parámetros específicos para evitar que los ataques por baja densidad y fuerza bruta sean efectivos. En este capítulo se estudiarán los ataques, cómo afectan a nuestro criptosistema y qué condiciones de parámetros son necesarias para garantizar seguridad frente a ellos.

4.1. Ataques a la clave privada

Este tipo de ataques tratan de romper la estructura de la generación de claves para, a partir de la clave pública, reconstruir la clave privada. Recordemos que en el criptosistema GLN las claves son las siguientes:

- La clave pública es una lista de n enteros no negativos: $(t_1, \ldots, t_n)$.
- La clave privada es una lista de $n+2$ enteros positivos: $(p_1, \ldots, p_n, g, u)$.

El atacante conoce $(t_1, \ldots, t_n)$ y los parámetros públicos del esquema (es decir, n, t, z). De ahí obtiene $l = \lceil \log_2 t \rceil$ y $b = \lceil \log_2 z \rceil$, y puede deducir el rango de elección de los primos privados p_i: se toman de entre los primos con $b+1, \ldots, b+\beta$ bits, donde $\beta \geq 1$ se fija para

que haya un rango de primos suficientemente grande. La cardinalidad de este conjunto puede acotarse inferiormente usando el Teorema de los Números Primos, que permite estimar cuántos candidatos tiene el atacante para intentar reconstruir la clave privada a prueba y error.

Teorema 4.1 (Teorema de los Números Primos de [32]). *Para x grande,*

$$\pi(x) \sim \frac{x}{\ln x},$$

donde $\pi(x)$ cuenta la cantidad de primos menores o iguales que x.

Por tanto, el número aproximado de primos con entre $b+1$ y $b+\beta$ bits es:

$$N_{primos} \approx \pi(2^{b+\beta}) - \pi(2^b) \approx \frac{2^{b+\beta}}{(b+\beta)\ln 2} - \frac{2^b}{b \ln 2}$$

4.1.0.1. Idea general del ataque

Sea h_i el inverso modular de los p_i módulo g. Si el sistema publicara directamente $\{h_i\}$, esta identidad permitiría a un atacante formar enteros divisibles por g del tipo $h_i p_i - 1$ y, probando parejas de posibles primos (p_a, p_b), calcular $\gcd(h_a p_a - 1, h_b p_b - 1)$ y reconstruir g como un divisor común grande. Esta sería la forma de conseguir g si la clave pública estuviese formada por $\{h_i\}$ y el conjunto de primos candidatos fuera pequeño.

Ejemplo 4.2. Supongamos los parámetros privados.

$$g = 97, \quad p_1 = 11, \quad p_2 = 13.$$

Calculamos los inversos módulo g de forma que:

$$11 \cdot h_1 \equiv 1 \pmod{97}, \quad 13 \cdot h_2 \equiv 1 \pmod{97}.$$

Probamos:

$$11 \cdot 53 = 583 \equiv 1 \pmod{97}, \quad 13 \cdot 15 = 195 \equiv 1 \pmod{97}.$$

Si el esquema publicara $h_1 = 53$ y $h_2 = 15$, un atacante podría formar los siguientes enteros:

$$X_1 = h_1 \cdot p_1 - 1 = 53 \cdot 11 - 1 = 582$$
$$X_2 = h_2 \cdot p_2 - 1 = 15 \cdot 13 - 1 = 194$$

Observemos que:

$$X_1 = 2 \cdot 3 \cdot 97, \quad X_2 = 2 \cdot 97.$$

Ahora el atacante calcula el máximo común divisor:

$$\gcd(X_1, X_2) = \gcd(582, 194) = 194.$$

Una vez conseguido 194, el atacante puede detectar el factor grande 97 dividiendo por pequeños primos, y así recuperar g.

■

Aunque en el ejemplo anterior asumimos que el atacante conoce los parámetros privados p_1 y p_2, en la práctica son secretos. Sin embargo, si el conjunto de primos posibles es pequeño (por el tamaño de bits de rango limitado), el atacante puede probar candidatos para p_i. Por lo tanto, si el espacio de candidatos es reducido, esta búsqueda por pares es factible; si es grande, se vuelve impráctica.

Además, nuestro criptosistema no publica $\{h_i\}$, sino $\{t_i\}$ con:

$$t_i \equiv h_i - u \pmod{g}$$

Esta fórmula evita el ataque gcd por parejas explicado, porque t_i ya no cumple una congruencia lineal simple con p_i y g que el atacante puede utilizar con gcd. El coste del atacante pasa a depender de recuperar g y u al mismo tiempo.

4.1.0.2. Ataque por tríos

Aunque el desplazamiento evita el ataque por parejas, sigue habiendo relaciones entre t_i, p_i y g y que el adversario podría utilizar si el rango de primos es pequeño.

Partiendo de $p_i(h_i) = 1 \pmod{g}$ y $h_i \equiv t_i + u \pmod{g}$, para dos índices i, j:

$$p_i(t_i + u) \equiv 1, \quad p_j(t_j + u) \equiv 1 \pmod{g}.$$

Restando, se obtiene la congruencia:

$$(p_i p_j)(t_i - t_j) \equiv p_j - p_i \pmod{g}.$$

Con tres primos candidatos (p_1, p_2, p_3), se pueden definir dos valores múltiplos de g:

$$X_{12} := (p_1 p_2)(t_1 - t_2) - (p_2 - p_1), \quad X_{13} := (p_1 p_3)(t_1 - t_3) - (p_3 - p_1)$$

Por tanto, $\gcd(X_{12}, X_{13})$ es un múltiplo pequeño de g , y permite aislar g o un múltiplo pequeño, que luego puede refinarse usando aritmética de Euclides y el hecho de que los parámetros son pequeños comparados con g. Este sería el ataque por tríos sobre $\{t_i\}$.

Una vez con un candidato g, se pueden resolver pequeñas congruencias lineales para recomponer los p_j y, reiterando, recuperar la clave privada completa.

Ejemplo 4.3. Consideremos $g = 97$ y los primos secretos $p_1 = 11, p_2 = 13, p_3 = 17$, con un desplazamiento secreto $u = 7$. Primero calculamos los inversos h_i de cada p_i módulo g:

- Para $p_1 = 11$, su inverso es $h_1 = 53$.
- Para $p_2 = 13$, su inverso es $h_2 = 15$.
- Para $p_3 = 17$, su inverso es $h_3 = 40$.

El esquema publica $t_i \equiv h_i - u \pmod{g}$. Calculamos los t_i:

- $t_1 = h_1 - u = 53 - 7 = 46$.
- $t_2 = 15 - 7 = 8$.
- $t_3 = 40 - 7 = 33$.

El atacante ve $(t_1, t_2, t_3) = (46, 8, 33)$ y puede probar con diferentes primos candidatos. Supongamos que el atacante prueba con el trío $(11, 13, 17)$. Y calcula

$$X_{12} = (p_1 p_2)(t_1 t_2) - (p_2 - p1) = 5432, \quad X_{13} = (p_1 p_3)(t_1 - t_3) - (p_3 - p1) = 2425.$$

Ahora el atacante calcula usando el Algoritmo de Euclides

$$\gcd(X_{12}, X_{13}) = \gcd(5432, 2425) = 97.$$

Es decir, probando con candidatos de primos, se puede llegar a obtener los primos correctos, que servirían para recuperar g.

Con g descubierto, el atacante vuelve a las congruencias

$$p_i(t_i + u) \equiv 1 \pmod{g}.$$

Si prueba un candidato para p_1, como $p_1 = 11$, puede despejar u con

$$p_1 u \equiv 1 - p_1 t_1 \pmod{g}.$$

Multiplica por el inverso modular de p_1 módulo g para obtener u

$$u \equiv (1 - p_1 t_1) \cdot p_1^{-1} \pmod{g}.$$

Con los parámetros $p_1 = 11, t_1 = 46, g = 97$:

$$1 - p_1 t_1 = 1 - 11 \cdot 46 = 1 - 506$$
$$1 - 506 \equiv -505 \equiv 77 \pmod{97}.$$

El inverso de $p_1 = 11 \pmod{97}$ es $p_1^{-1} = 53$. Luego,

$$u \equiv 77 \cdot 53 \pmod{97} \equiv 7.$$

Efectivamente, hemos recuperado el desplazamiento u correctamente. Con g y u ya encontrados, es sencillo calcular los primos candidatos y completar la reconstrucción de la clave privada completa.

■

4.1.0.3. Solución al ataque por tríos

La viabilidad del ataque por tríos depende del tamaño del rango de primos. Definamos N_{primos} al número de primos de $b+1, \ldots, b+\beta$ bits (estimable inferiormente con el Teorema de los Números Primos). Un atacante que prueba tríos debe explorar en torno a $\binom{N_{primos}}{3}$ combinaciones.

Ejemplo 4.4. Consideremos dos casos diferentes:

- $z = 2^5$, luego $b = \lceil \log_2 z \rceil = 5$ y $\beta = 3$.
- $z = 2^{10}$, luego $b = \lceil \log_2 z \rceil = 5$ y $\beta = 3$.

Usando la aproximación del Teorema de los Números Primos tenemos que:

1. Para $b = 5, \quad \beta = 3 : N_{primos} \approx 37$. Por tanto, el número de tríos de primos que el atacante tendría que probar son:

$$\binom{N}{3} = \binom{37}{3} = 7770 \approx 2^{13}.$$

2. Para $b = 10, \quad \beta = 3 : N_{primos} \approx 762$. Por tanto, el número de tríos de primos que el atacante tendría que probar son:

$$\binom{N}{3} = \binom{762}{3} = 73451720 \approx 2^{26}.$$

Este ejemplo muestra que el ataque crece extremadamente rápido con el tamaño del alfabeto. Basta incrementar b unos pocos bits para multiplicar por órdenes de magnitud el espacio de búsqueda del atacante.

■

4.2. Ataques al problema SSP

4.2.1. Ataques basados en densidad

En esta sección veremos los ataques más conocidos a los criptosistemas basados en mochila originales se basaban en la dificultad de resolver otra versión del problema.

Definición 4.5 (Binary Subset Sum Problem (BSSP))**.** Dados $a_1, \ldots, a_n \in \mathbb{Z}^+$ y un objetivo s, hallar $x \in \{0,1\}^n$ tal que

$$\sum_{i=1}^{n} a_i x_i = s$$

Es la forma binaria del SSP y ha experimentado distintos ataques durante su historia. Estos ataques afectaban a los criptosistemas originales que utilizaban secuencias súpercrecientes, como Merkle-Hellman y, por tanto, no afecta a nuestro caso. Sin embargo, en criptografía de tipo mochila, esto llevó a desarrollar nuevos ataques basados en densidad.

En criptografía de tipo mochila, un parámetro fundamental para medir la dificultad de una instancia es la densidad.

Definición 4.6 (Densidad según [22])**.** La densidad de una instancia del BSSP con n elementos se define como

$$d = \frac{n}{\log_2(max(a_1, \ldots, a_n))}.$$

El denominador mide el número de bits necesarios para representar el mayor peso a_i, es decir, la magnitud de los coeficientes de la mochila. Así, la densidad compara la dimensión n del problema con el tamaño en bits de los números que lo forman. Intuitivamente:

- Si d es pequeño (densidad baja), los pesos son grandes en comparación con n, lo que tiende a hacer más única la solución.
- Si d es grande (densidad alta), hay muchos subconjuntos distintos que producen sumas cercanas, lo que dificulta distinguir la solución correcta.

Ejemplo 4.7. Supongamos una suma objetivo $s = 20$ y $n = 5$ posibles sumandos. Consideremos dos subconjuntos de enteros distintos:

- $A = \{1, 2, 4, 8, 16\}$
- $A' = \{6, 7, 8, 9, 10\}$

Para conseguir la suma $s = 20$ usando A podemos empezar probando: $1 + 2 + 4 + 8 = 15 < 20$. Por lo que interpretamos que se necesita el 16. Luego, $16 + 4 = 20$, y ya hemos encontrado la solución.

Sin embargo, el mismo objetivo usando A' requiere mucho más esfuerzo: $6 + 7 + 8 = 21 > 20$, lo que no nos da ninguna pista. Seguimos probando $10 + 9 = 19 < 20$, que tampoco nos da ninguna pista. Y tenemos que seguir probando.

Vemos que en el caso de A no hay solapamiento fuerte entre pesos, es decir, los números son muy distintos entre sí, lo que permite encontrar una estructura geométrica en el conjunto. Sin embargo, en el caso de A', los pesos están muy agrupados, y pequeñas variaciones crean múltiples sumas similares, lo que hace muy difícil encontrar una estructura. Comparemos las densidades de ambas instancias del problema:

- Para A: $d = \dfrac{5}{\log_2(16)} = \dfrac{5}{4} = 1.25$, que se considera baja-media densidad.
- Para A': $d = \dfrac{5}{\log_2(10)} \approx \dfrac{5}{3.32} \approx 1.50$, con densidad mayor.

Las densidades confirman nuestra hipótesis: el caso de A tiene una densidad más pequeña que la instancia de A', es decir, encontrar la suma usando los valores de A resulta más fácil que usando los valores de A'.

■

En 1985, Lagarias–Odlyzko demostró que si la densidad d de una instancia del problema del BSSP es menor que 0.645, casi todas las instancias pueden resolverse mediante algoritmos de reducción de retículos [16]. Posteriormente, se demostró que si $d < 0.9408$, entonces el vector más corto en el retículo asociado a la instancia corresponde con alta probabilidad a una solución válida [8].

Intuitivamente, en baja densidad, el subconjunto que suma s es demasiado único y puede identificarse a través de la geometría de los números: se construye un retículo a partir de los a_i. Y resolver el SSP se convierte en buscar el vector más corto en ese retículo (Shortest Vector Problem, SVP). Como el espacio de soluciones es reducido (baja densidad), el vector correspondiente a la solución verdadera destaca y es recuperable.

En densidad alta, en cambio, hay muchos subconjuntos distintos que pueden sumar valores cercanos a s, lo que esconde mejor la solución.

Ejemplo 4.8. Supongamos un problema con $A = \{3, 7, 11, 19, 42\}$, $n = 5$. El máximo es $\max A = 42$, y en binario se necesitan $\log_2(42) \approx 5.39$ bits. Entonces la densidad es:

$$d = \frac{n}{\log_2(\max A)} \approx \frac{5}{5.39} \approx 0.93.$$

Como $d < 1$, estamos en un caso de baja densidad, y esto significa que el problema es más vulnerable: un atacante podría usar reducción de retículos para encontrar la combinación de elementos que da s.

■

4.2.2. Reducción a problemas de retículos (SVP)

Una de las principales ideas para atacar criptosistemas de tipo mochila es reducir el Subset Sum Problem (SSP) al problema de encontrar un vector más corto en un retículo (Shortest Vector Problem, SVP). La complejidad exacta de cómo encontrar un vector más corto en retículos no se conoce, pero se conocen métodos de tiempo polinomial para aproximarlo, como el algoritmo de reducción de bases LLL de Lenstra, Lenstra y Lovász.

Pero antes de entrar en más detalle, recordemos algunas nociones matemáticas, siguiendo [1].

Definición 4.9 (Retículo)**.** Un retículo $L \subset \mathbb{R}^n$ es el conjunto de todas las combinaciones lineales con coeficientes enteros de un conjunto de m vectores linealmente independientes

$$B = \{b_1, \ldots, b_m\} \subset \mathbb{R}^n, \quad m \leq n.$$

Es decir,

$$L = L(B) = \{a_1 b_1 + \cdots + a_m b_m : a_i \in \mathbb{Z}\}.$$

El conjunto B se denomina base de L. La dimensión del espacio es n y el rango del retículo es m.

De ahora en adelante, asumimos que cualquier base B tiene vectores en $\mathbb{Z}^n$.

Sea B la matriz $n \times m$ cuyas columnas son los vectores de la base $b_1, \ldots, b_m$. Entonces

$$B = \begin{bmatrix} b_1 & b_2 & \ldots & b_m \end{bmatrix}$$

y el retículo generado por B puede escribirse de forma compacta como

$$L(B) = \{Bv : v \in \mathbb{Z}^m\}.$$

Ejemplo 4.10. En $\mathbb{R}^2$, con la base estándar $b_1 = (1,0)$, $b_2 = (0,1)$ se tiene

$$L = \{(z_1, z_2) : z_1, z_2 \in \mathbb{Z}\} = \mathbb{Z}^2,$$

es decir, todos los puntos con coordenadas enteras del plano.

En cambio, si usamos la base $b_1 = (2,0)$ y $b_2 = (1,3)$, el retículo es

$$L = \{z_1(2,0) + z_2(1,3) : z_1, z_2 \in \mathbb{Z}\}.$$

En este caso, los puntos del retículo forman una rejilla inclinada y estirada respecto a la base estándar.

■

Ejemplo 4.11. Veamos ahora un ejemplo en $\mathbb{R}^3$. Sea $B = \{b_1, b_2\}$, donde

$$b_1 = (0,0,1), \qquad b_2 = (1,0,1).$$

La matriz de la base es

$$B = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix},$$

y el retículo generado es

$$L(B) = \left\{ \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} x : x \in \mathbb{Z}^2 \right\}.$$

El vector $v = (2, 0, 3)$ pertenece a $L(B)$, porque $v = b_1 + 2b_2$. En efecto,

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 3 \end{bmatrix} = v.$$

Sin embargo, el vector $v' = (0, 1, 0)$ no pertenece a $L(B)$. Si escribimos

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ 0 \\ x + y \end{bmatrix},$$

vemos que la segunda coordenada es siempre 0 para cualquier $x, y \in \mathbb{Z}$, por lo que nunca podremos obtener la segunda coordenada igual a 1. Luego $v' \notin L(B)$.

■

Cuando $m = n$, el rango y la dimensión de un retículo L coinciden y se dice que L es de rango completo. De ahora en adelante, asumiremos que hablamos de retículos de este tipo.

Definición 4.12. Sea $L = L(B)$ un retículo de rango completo en $\mathbb{R}^n$, donde $B = \{b_1, \ldots, b_n\}$. El paralelepípedo fundamental de L es

$$P(B) = \{a_1 b_1 + \cdots + a_n b_n : a_i \in [0, 1)\} = \{Ba : a \in [0, 1)^n\}.$$

El paralelepípedo fundamental $P(B)$ se puede usar para particionar $\mathbb{R}^n$ en regiones no solapadas (también llamadas paralelepípedos). Cada una de estas regiones es una copia trasladada de $P(B)$ y sus esquinas son precisamente los elementos del retículo $L(B)$.

Definición 4.13. La longitud de un vector $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ es su norma euclídea:

$$\|a\| = \sqrt{a_1^2 + \cdots + a_n^2}.$$

Definición 4.14. Decimos que una matriz $U \in \mathbb{Z}^{n \times n}$ es unimodular si

$$\det(U) = \pm 1.$$

Lema 4.15. *Sea $U \in \mathbb{Z}^{n \times n}$ unimodular. Entonces U es invertible y U^{-1} tiene entradas enteras.*

Demostración. La matriz inversa de U se define como

$$U^{-1} = \frac{1}{\det(U)} \operatorname{adj}(U),$$

donde $\operatorname{adj}(U)$ es la matriz adjunta de U, es decir, la traspuesta de la matriz de cofactores.

Para una matriz cuadrada A, el cofactor C_{ij} del elemento a_{ij} se define como

$$C_{ij} = (-1)^{i+j} M_{ij},$$

donde M_{ij} es el determinante de la submatriz que se obtiene al eliminar la fila i y la columna j de A.

Si todos los elementos de U son enteros, cada submatriz tiene determinante entero y, por tanto, cada cofactor C_{ij} es entero. Esto implica que $\text{adj}(U)$ tiene entradas enteras.

Como $\det(U) = \pm 1$, se tiene

$$U^{-1} = \pm \,\text{adj}(U),$$

que existe y tiene también entradas enteras. □

Lema 4.16. *Sea $U \in \mathbb{Z}^{n\times n}$. Entonces U es unimodular si y solo si*

$$U(\mathbb{Z}^n) = \mathbb{Z}^n.$$

Demostración. Supongamos primero que U es unimodular. Por el Lema 4.15, U es invertible y U^{-1} tiene entradas enteras.

Sea $z \in \mathbb{Z}^n$. Entonces Uz es una combinación lineal de las columnas de U con coeficientes enteros, por lo que $Uz \in \mathbb{Z}^n$ y se tiene

$$U(\mathbb{Z}^n) \subseteq \mathbb{Z}^n.$$

Recíprocamente, sea $w \in \mathbb{Z}^n$. Como U es invertible, la ecuación $Uz = w$ tiene solución única

$$z = U^{-1}w.$$

Dado que U^{-1} tiene entradas enteras y w es un vector de enteros, también z es un vector de enteros, por lo que $w = Uz \in U(\mathbb{Z}^n)$. Así obtenemos

$$\mathbb{Z}^n \subseteq U(\mathbb{Z}^n),$$

y por tanto $U(\mathbb{Z}^n) = \mathbb{Z}^n$.

Para la implicación en el otro sentido, supongamos que

$$U(\mathbb{Z}^n) = \mathbb{Z}^n.$$

Entonces, en particular, cada vector de la base canónica e_i pertenece a $U(\mathbb{Z}^n)$. Por tanto, para cada i existe $z^{(i)} \in \mathbb{Z}^n$ tal que

$$Uz^{(i)} = e_i.$$

Definimos la matriz $V \in \mathbb{Z}^{n\times n}$ cuyas columnas son los vectores $z^{(i)}$. Por construcción se cumple

$$UV = I_n.$$

Es decir, V es una inversa por la derecha de U. Una matriz cuadrada que tiene inversa por la derecha es invertible, y esa inversa coincide con la inversa usual; por tanto $V = U^{-1}$.

Como V tiene entradas enteras, $U^{-1} \in \mathbb{Z}^{n\times n}$. En particular, su determinante es un entero. Además,

$$\det(U)\ \det(U^{-1}) = \det(I_n) = 1.$$

Por tanto, $\det(U)$ es un entero cuyo producto con otro entero da 1, lo que implica

$$\det(U) = \pm 1.$$

Luego U es unimodular. □

Teorema 4.17. *Un retículo L tiene infinitas bases distintas.*

Demostración. Sea $B \in \mathbb{Z}^{n\times n}$ una base de rango completo de L y sea $U \in \mathbb{Z}^{n\times n}$ una matriz unimodular. Definimos

$$B' = BU.$$

Consideramos el retículo generado por B':

$$L' = \{B'z : z \in \mathbb{Z}^n\} = \{BUz : z \in \mathbb{Z}^n\}.$$

Como $U(\mathbb{Z}^n) = \mathbb{Z}^n$, se tiene

$$L' = \{Bw : w \in \mathbb{Z}^n\} = L.$$

Por tanto, $B' = BU$ es también una base del mismo retículo L.

Para concluir, basta observar que existen infinitas matrices unimodulares en $\mathbb{Z}^{n\times n}$. Por ejemplo, para cada $k \in \mathbb{Z}$, la matriz

$$U_k = \begin{bmatrix} 1 & 0 & \dots & 0 & k \\ 0 & 1 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}$$

es unimodular y todas ellas son distintas. De este modo se obtienen infinitas bases diferentes para L. □

Ejemplo 4.18. Consideremos la base

$$B = \{(0,0,1),(1,0,1),(1,1,1)\}$$

con matriz asociada

$$B = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

y la matriz unimodular

$$U = \begin{bmatrix} 2 & 3 & 2 \\ 4 & 2 & 3 \\ 9 & 6 & 7 \end{bmatrix}, \qquad \det(U) = 1.$$

Entonces,

$$B' = BU = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 2 \\ 4 & 2 & 3 \\ 9 & 6 & 7 \end{bmatrix} = \begin{bmatrix} 13 & 8 & 10 \\ 9 & 6 & 7 \\ 15 & 11 & 12 \end{bmatrix}.$$

Las columnas de B' forman otra base distinta del mismo retículo $L(B)$.

■

Definición 4.19. Sea $L = L(B) \subseteq \mathbb{Z}^n$ un retículo de rango completo. El volumen de L se define como

$$\mathrm{vol}(L) = |\det(B)|$$

y coincide con el volumen del paralelepípedo fundamental $P(B)$.

Lema 4.20. *El volumen es un invariante del retículo, es decir, no depende de la base B elegida para generar L.*

Demostración. Sean B y B' dos bases de L. Queremos ver que

$$|\det(B)| = |\det(B')|.$$

Como $L = B\mathbb{Z}^n = B'\mathbb{Z}^n$, cada columna de B' es una combinación lineal de las columnas de B con coeficientes enteros. Esto implica la existencia de una matriz $U \in \mathbb{Z}^{n\times n}$ tal que

$$B' = BU.$$

Entonces

$$B'\mathbb{Z}^n = BU\mathbb{Z}^n = B\mathbb{Z}^n.$$

De aquí se deduce que

$$U(\mathbb{Z}^n) = \mathbb{Z}^n,$$

así que U es unimodular. Por tanto,

$$\det(B') = \det(BU) = \det(B)\ \det(U) = \pm\det(B),$$

y al tomar valor absoluto se obtiene

$$|\det(B')| = |\det(B)|,$$

como queríamos probar. □

Definición 4.21. Sea $L \subseteq \mathbb{Z}^n$ un retículo de rango n. Para cada $i \in \{1,\dots,n\}$, el mínimo sucesivo i, denotado por $\lambda_i(L)$, es el valor real más pequeño $r > 0$ tal que L contiene i vectores linealmente independientes, todos ellos de norma a lo sumo r.

En particular, se cumple

$$\lambda_1(L) \leq \lambda_2(L) \leq \cdots \leq \lambda_n(L),$$

y $\lambda_1(L)$ es la longitud del vector no nulo más corto de L.

Teorema 4.22 (Minkowski)**.** *Sea L un retículo de rango n. Entonces*

$$\lambda_1(L) \leq \sqrt{n} \cdot \mathrm{vol}(L)^{1/n}.$$

Con estas definiciones ya podemos presentar el problema clave de esta sección.

Definición 4.23 (Shortest Vector Problem, SVP)**.** Dado un retículo $L \subset \mathbb{Z}^n$ y una norma, el problema del vector más corto consiste en encontrar un vector no nulo $v \in L$ de longitud mínima:

$$\|v\| = \lambda_1(L),$$

donde $\lambda_1(L)$ es el primer mínimo sucesivo.

En general, encontrar exactamente ese vector es un problema NP-difícil en dimensión grande. Sin embargo, para dimensiones moderadas es posible obtener soluciones aproximadas de forma eficiente, lo que resulta suficiente en muchos ataques criptográficos. El algoritmo LLL (debido a Lenstra, Lenstra y Lovász) fue el primero en proporcionar un método en tiempo polinomial para aproximar el vector más corto mediante una base casi reducida, y sus mejoras se han utilizado para romper diversos esquemas criptográficos.

Para convertir el SSP en un problema de buscar un vector corto en un retículo, podemos verlo como una ecuación lineal con bits. Tenemos un vector de pesos $A = (a_1, \ldots, a_n) \in \mathbb{Z}^n$, una suma objetivo $S \in \mathbb{Z}$ y buscamos $X = (x_1, \ldots, x_n)^\top \in \{0,1\}^n$ tal que:

$$AX = S.$$

En el contexto de problemas de tipo mochila, A es la clave pública y X es el mensaje cifrado.

La idea es fabricar una base de retículo (matriz M) tal que el vector

$$W = \begin{bmatrix} X \\ 0 \end{bmatrix}$$

pertenezca al retículo $L = L(M)$ y sea significativamente más corto que los vectores típicos de L. Si logramos eso, un algoritmo de reducción de retículos tenderá a sacarlo o acercarse mucho.

Para conseguir esta base, tenemos que escribir la ecuación $AX = S$ como el sistema bloque:

$$MV = W, \text{ con } M = \begin{bmatrix} I_n & 0 \\ A & -S \end{bmatrix}, V = \begin{bmatrix} X \\ 1 \end{bmatrix}, W = \begin{bmatrix} X \\ 0 \end{bmatrix}.$$

Donde I_n es la identidad $n \times n$. La última fila

$$\begin{bmatrix} A & -S \end{bmatrix}$$

codifica $AX - S = 0$. Si X resuelve el SSP, entonces existe $V = [X; 1]$ tal que $MV = W$ y, por tanto, W está en el retículo generado por las columnas de M.

Ejemplo 4.24. Veamos un ejemplo simple pero completo del proceso de convertir un SSP en un SVP. Imaginemos que un atacante intercepta la siguiente comunicación que se está realizando usando el criptosista de Merkle-Hellman:

- La clave pública es $A = (10, 25, 43, 9)$.
- El texto cifrado es $S = 35$.

El atacante sabe que el mensaje original es un vector X de 0s y 1s, y que $AX = S$. Su objetivo es encontrar X.

El primer paso del ataque es convertir el SSP en un SVP, es decir, debe reformular el problema para poder usar la reducción de retículos. Contruye una matriz M que combina la clave pública, el texto cifrado y la matriz identidad. En este caso, como A tiene cuatro

elementos, la matriz M será de 5×5:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 10 & 25 & 43 & 9 & -35 \end{bmatrix}$$

Las columnas de esta matriz M forman la base del retículo en el que el atacante buscará la solución. El vector solución W que busca dentro de este retículo tendrá una estructura muy específica y será inusualmente corto: sus primeras componentes serán los 0s y 1s del mensaje original X, y su última componente será 0.

El vector objetivo es

$$W = [x_0, \ldots, x_3, 0]^\top$$

con longitud máxima $\sqrt{4} = 2$, si todos los x_i fueran 1, lo que lo hace significativamente más corto que un vector típico en este retículo.

El siguiente paso del atacante sería introducir la matriz M en el algoritmo LLL. El algoritmo ejecutará sus procesos internos para encontrar una nueva base M' para el mismo retículo, pero compuesta por vectores mucho más cortos.

Para este ejemplo, al introducir la matriz M como entrada en la implementación del algoritmo LLL, nos devuelve la siguiente matriz M':

$$M' = \begin{bmatrix} 1 & -1 & 0 & 0 & -2 \\ 1 & 0 & 0 & -1 & 2 \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 1 & -1 & 1 & -1 \\ 0 & -1 & -1 & 0 & 2 \end{bmatrix}$$

El último paso del atacante es examinar las columnas de la matriz M' en busca de un vector que coincida con la estructura de W: primeras 4 entradas con 0s y 1s, y la última es 0. Al revisarlo, el atacante encuentra que la primera columna es $(1, 1, 0, 0, 0)^\top$ y, por tanto, puede concluir con que el mensaje original es:

$$X = (1, 1, 0, 0).$$

Para verificar que la solución es correcta, simplemente comprueba que:

$$AX = S$$

$$1 \cdot 10 + 1 \cdot 25 + 0 \cdot 43 + 0 \cdot 9 = 35.$$

■

Una vez construida la base M, la idea es buscar el vector W que pertenezca a $L(M)$ y sea más corto que los vectores típicos suyos. Queremos que sea más corto porque los primeros n componentes de W son bits $0/1$ y el último es 0, de modo que

$$\|W\| = \sqrt{x_1^2 + \cdots + x_n^2} \leq \sqrt{n},$$

que es mucho menor que la longitud típica de un vector a azar del retículo $L(M)$. Estas

características hacen a W un objetivo ideal para los algoritmos de reducción de base.

4.2.3. Ortogonalización de Gram-Schmidt

Puesto que podemos elegir distintas bases para un mismo retículo, es natural preguntarse qué bases son mejores desde el punto de vista algorítmico. En particular, para muchos problemas sobre retículos resulta más sencillo trabajar con bases formadas por vectores cortos y casi ortogonales que con bases cuyos vectores sean largos y muy inclinados entre sí.

El proceso de reducción de base tiene precisamente este objetivo: transformar una base arbitraria en otra base del mismo retículo con vectores lo más cortos y cercanos a ortogonales posible.

El algoritmo de Lenstra–Lenstra–Lovász (LLL), propuesto en 1982, es el método clásico para la reducción de bases de retículos en varias dimensiones. Este algoritmo recibe una base cualquiera

$$B = \{b_1, \dots, b_n\}$$

y la transforma mediante operaciones elementales (reducciones y permutaciones de vectores) hasta obtener una base reducida. Para describir correctamente este algoritmo, necesitamos introducir primero la ortogonalización de Gram–Schmidt.

Definición 4.25. Sean $u = (u_1, \dots, u_n)$ y $v = (v_1, \dots, v_n)$ dos vectores de $\mathbb{R}^n$. Se define el producto escalar como

$$\langle u, v \rangle = u_1 v_1 + \cdots + u_n v_n.$$

En particular, el cuadrado de la longitud de un vector $u \in \mathbb{R}^n$ es

$$\|u\|^2 = u_1^2 + \cdots + u_n^2 = \langle u, u \rangle.$$

Lema 4.26. *El producto escalar es simétrico y lineal.*

Demostración. Para la simetría, basta observar que, para $u = (u_1, \dots, u_n)$ y $v = (v_1, \dots, v_n)$, se cumple

$$\langle u, v \rangle = u_1 v_1 + \cdots + u_n v_n = v_1 u_1 + \cdots + v_n u_n = \langle v, u \rangle,$$

ya que el producto de números reales es conmutativo.

Para la linealidad, sea $\alpha \in \mathbb{R}$ y $u, v, w \in \mathbb{R}^n$. Demostramos que el producto escalar es lineal en la segunda componente:

$$\langle u, v + w \rangle = u_1(v_1 + w_1) + \cdots + u_n(v_n + w_n) = (u_1 v_1 + \cdots + u_n v_n) + (u_1 w_1 + \cdots + u_n w_n) = \langle u, v \rangle + \langle u, w \rangle.$$

Asimismo,

$$\langle u, \alpha v \rangle = u_1(\alpha v_1) + \cdots + u_n(\alpha v_n) = \alpha(u_1 v_1 + \cdots + u_n v_n) = \alpha \langle u, v \rangle.$$

Como el producto escalar es simétrico, estas propiedades implican también la linealidad en la primera componente. Por tanto, el producto escalar es simétrico y lineal. □

Recordemos que un subespacio vectorial de $\mathbb{R}^n$ es un subconjunto de $\mathbb{R}^n$ cerrado para la suma y para el producto por escalares.

Definición 4.27. Sea V un subespacio vectorial de dimensión k de $\mathbb{R}^n$ con base ordenada $B = \{b_1, \ldots, b_k\}$. Decimos que B es una base ortogonal de V si

$$\langle b_i, b_j \rangle = 0, \quad \forall\, 1 \leq j < i \leq k.$$

Definición 4.28. Sea V un subespacio vectorial de dimensión k de $\mathbb{R}^n$ con base ordenada $B = \{b_1, \ldots, b_k\}$. El complemento ortogonal de V se define como

$$V^{\perp} = \{x \in \mathbb{R}^n : \langle x, v \rangle = 0 \text{ para todo } v \in V\}.$$

Es un subespacio de dimensión $n - k$.

Una base $B = \{b_1, \ldots, b_k\}$ de un subespacio V de dimensión k de $\mathbb{R}^n$ puede extenderse a una base de $\mathbb{R}^n$ del tipo

$$\{b_1, \ldots, b_k, b_{k+1}, \ldots, b_n\},$$

donde $\{b_{k+1}, \ldots, b_n\}$ es una base de $V^{\perp}$.

Dado cualquier vector $w \in \mathbb{R}^n$, podemos escribirlo de forma única como

$$w = c_1 b_1 + \cdots + c_n b_n, \quad c_i \in \mathbb{R},$$

y separarlo en dos componentes:

$$w_1 = c_1 b_1 + \cdots + c_k b_k, \qquad w_2 = c_{k+1} b_{k+1} + \cdots + c_n b_n,$$

de modo que

$$w = w_1 + w_2,$$

con $w_1 \in V$ y $w_2 \in V^{\perp}$.

Definición 4.29. La proyección de $w \in \mathbb{R}^n$ sobre V se define como

$$\Pi_V(w) = w_1.$$

La proyección de w sobre $V^{\perp}$ es

$$\Pi_{V^{\perp}}(w) = w_2 = w - w_1 = w - \Pi_V(w).$$

En consecuencia,

$$w = \Pi_V(w) + \Pi_{V^{\perp}}(w).$$

Lema 4.30. *Sea $B = \{b_1, \ldots, b_k\}$ una base ortogonal de un subespacio $V \subset \mathbb{R}^n$ y sea $w \in \mathbb{R}^n$. Entonces*

$$\Pi_V(w) = \sum_{j=1}^{k} \frac{\langle w, b_j \rangle}{\|b_j\|^2}\, b_j.$$

Demostración. Extendemos la base ortogonal B a una base de $\mathbb{R}^n$:

$$\{b_1, \ldots, b_k, b_{k+1}, \ldots, b_n\},$$

donde $\{b_{k+1}, \ldots, b_n\}$ es una base de $V^{\perp}$. Escribimos

$$w = c_1 b_1 + \cdots + c_k b_k + c_{k+1} b_{k+1} + \cdots + c_n b_n, \quad c_i \in \mathbb{R}.$$

Para cada $1 \leq j \leq k$ se tiene

$$\begin{aligned}\langle w, b_j \rangle &= \Big\langle \sum_{i=1}^{n} c_i b_i,\ b_j \Big\rangle \\ &= \sum_{i=1}^{n} c_i \langle b_i, b_j \rangle \\ &= c_j \langle b_j, b_j \rangle,\end{aligned}$$

pues $\langle b_i, b_j \rangle = 0$ si $i \neq j$ por ortogonalidad. De aquí

$$c_j = \frac{\langle w, b_j \rangle}{\langle b_j, b_j \rangle} = \frac{\langle w, b_j \rangle}{\|b_j\|^2}.$$

La proyección de w sobre V es

$$\Pi_V(w) = c_1 b_1 + \cdots + c_k b_k = \sum_{j=1}^{k} \frac{\langle w, b_j \rangle}{\|b_j\|^2}\, b_j.$$

□

Definición 4.31. Sea $B = \{b_1, \ldots, b_k\}$ una familia de vectores de $\mathbb{R}^n$. El subespacio generado por B (o expansión de B) se define como

$$\mathrm{Span}(b_1, \ldots, b_k) = \{c_1 b_1 + \cdots + c_k b_k : c_i \in \mathbb{R},\ i = 1, \ldots, k\}.$$

Definición 4.32 (Gram–Schmidt)**.** Sea V un subespacio de $\mathbb{R}^n$ con base

$$B = \{b_1, \ldots, b_k\}.$$

Definimos los vectores ortogonales de Gram–Schmidt

$$B^* = \{b_1^*, \ldots, b_k^*\}$$

de forma iterativa:

$$b_1^* = b_1,$$

y, para cada $2 \leq i \leq k$,

$$V_{i-1}^* = \mathrm{Span}(b_1^*, \ldots, b_{i-1}^*), \quad b_i^* = \Pi_{(V_{i-1}^*)^\perp}(b_i).$$

Equivalentemente, definimos los coeficientes de Gram–Schmidt como

$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}, \quad 1 \leq j < i \leq k,$$

de modo que

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j}\, b_j^*.$$

De ahora en adelante, utilizaremos la notación Π_i para referirnos a la proyección sobre el

complemento ortogonal de $V_i^* = \text{Span}(b_1^*, \ldots, b_i^*)$, es decir,

$$\Pi_i = \Pi_{(V_i^*)^\perp}.$$

Lema 4.33. *Para cada $1 \leq i \leq k$, definimos*

$$V_i = \text{Span}(b_1, \ldots, b_i), \quad V_i^* = \text{Span}(b_1^*, \ldots, b_i^*).$$

Entonces $V_i^* = V_i$ y $\{b_1^*, \ldots, b_i^*\}$ es una base ortogonal de V_i^*.

Demostración. Procedemos por inducción sobre i.

Para $i = 1$, se tiene $b_1^* = b_1$, luego

$$V_1^* = \text{Span}(b_1^*) = \text{Span}(b_1) = V_1,$$

y $\{b_1^*\}$ es una base ortogonal de V_1^*.

Supongamos ahora que para algún $i \geq 2$ se cumple la hipótesis de inducción:

$$V_{i-1}^* = V_{i-1} \quad \text{y} \quad \{b_1^*, \ldots, b_{i-1}^*\} \text{ es una base ortogonal de } V_{i-1}^*.$$

Por definición,

$$b_i^* = b_i - \Pi_{V_{i-1}^*}(b_i).$$

Como $V_{i-1}^* = V_{i-1}$, la proyección $\Pi_{V_{i-1}^*}(b_i)$ pertenece a V_{i-1} y, por tanto, también a V_i. Además, $b_i \in V_i$, así que

$$b_i^* = b_i - \Pi_{V_{i-1}^*}(b_i) \in V_i.$$

En consecuencia,

$$V_i^* = \text{Span}(b_1^*, \ldots, b_{i-1}^*, b_i^*) \subseteq V_i.$$

Por otro lado, de la expresión anterior se obtiene

$$b_i = b_i^* + \Pi_{V_{i-1}^*}(b_i),$$

donde $\Pi_{V_{i-1}^*}(b_i) \in V_{i-1}^* \subseteq V_i^*$. Por tanto, $b_i \in V_i^*$ y, como $V_{i-1} \subseteq V_{i-1}^* \subseteq V_i^*$, se tiene

$$V_i = \text{Span}(b_1, \ldots, b_i) \subseteq V_i^*.$$

De las dos inclusiones se deduce que $V_i^* = V_i$.

Finalmente, por construcción b_i^* es ortogonal a V_{i-1}^*. En particular,

$$\langle b_i^*, b_j^* \rangle = 0 \quad \text{para todo } 1 \leq j \leq i-1.$$

Esto demuestra que $\{b_1^*, \ldots, b_i^*\}$ es una base ortogonal de V_i^*. □

Este lema garantiza que el proceso de Gram–Schmidt, aplicado a una base B, produce una familia B^* que es una base ortogonal del mismo subespacio generado por B.

El pseudocódigo del algoritmo de Gram–Schmidt es el siguiente:

Algorithm 3 Algoritmo de Gram–Schmidt

Require: Vectores $\{b_1, b_2, \ldots, b_n\} \subset \mathbb{R}^m$
Ensure: Vectores ortogonales $\{b_1^*, b_2^*, \ldots, b_n^*\} \subset \mathbb{R}^m$
1: $b_1^* \leftarrow b_1$
2: **for** $i = 2$ **to** n **do**
3: $\quad v \leftarrow b_i$
4: $\quad$ **for** $j = i - 1$ **downto** 1 **do**
5: $\quad\quad \mu_{i,j} \leftarrow \dfrac{\langle b_i,\, b_j^* \rangle}{\langle b_j^*,\, b_j^* \rangle}$
6: $\quad\quad v \leftarrow v - \mu_{i,j}\, b_j^*$
7: $\quad$ **end for**
8: $\quad b_i^* \leftarrow v$
9: **end for**
10: **return** $\{b_1^*, b_2^*, \ldots, b_n^*\}$

Ejemplo 4.34. Consideremos el conjunto de vectores en $\mathbb{R}^3$:

1. $b_1 = (1, 1, 1)$
2. $b_2 = (2, 1, 0)$
3. $b_3 = (3, 1, 2)$

Queremos encontrar un conjunto ortogonal de vectores $B' = \{b_1^*, b_2^*, b_3^*\}$.

El primer vector ortogonal es:

$$b_1^* = b_1 = (1, 1, 1).$$

Para calcular el segundo vector ortogonal calculamos el coeficiente de proyección:

$$\mu_{2,1} = \frac{\langle b_2, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \frac{2 \cdot 1 + 1 \cdot 1 + 0 \cdot 1}{1^2 + 1^2 + 1^2} = 1.$$

Ahora calculamos b_2^*:

$$b_2^* = b_2 - \mu \cdot b_1^* = (2, 1, 0) - 1 \cdot (1, 1, 1) = (1, 0, -1).$$

Para calcular el tercer vector ortogonal calculamos los coeficientes de proyección:

$$\mu_{3,1} = \frac{\langle b_3, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \frac{3 \cdot 1 + 1 \cdot 1 + 2 \cdot 1}{1^2 + 1^2 + 1^2} = 2, \quad \mu_{3,2} = \frac{\langle b_3, b_2^* \rangle}{\langle b_2^*, b_2^* \rangle} = \frac{3 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1)}{1^2 + 0^2 + (-1)^2} = \frac{1}{2}.$$

Ahora calculamos b_3^*:

$$b_3^* = b_3 - \mu_{3,1} \cdot b_1^* - \mu_{3,2} \cdot b_2^* = (3, 2, 1) - 2 \cdot (1, 1, 1) - \frac{1}{2} \cdot (1, 0, -1) = (0.5, -1, 0.5).$$

Los vectores ortogonales son:

$$B' = \{(1, 1, 1), (1, 0, -1), (0.5, -1, 0.5)\}.$$

Podemos verificar que son ortogonales calculando los productos internos:

- $\langle b_1^*, b_2^* \rangle = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot (-1) = 0.$
- $\langle b_1^*, b_3^* \rangle = 1 \cdot 0.5 + 1 \cdot (-1) + 1 \cdot 0.5 = 0.$
- $\langle b_2^*, b_3^* \rangle = 1 \cdot 0.5 + 0 \cdot (-1) + (-1) \cdot 0.5 = 0.$

■

4.2.4. Reducción de bases de retículos con LLL

Ahora vamos a definir formalmente qué significa que una base esté reducida en el sentido de Lenstra–Lenstra–Lovász (LLL) y presentamos el algoritmo que produce este tipo de bases. Recordemos que, dada una base

$$B = \{b_1, \ldots, b_n\}$$

tenemos su familia de vectores de Gram–Schmidt

$$B^* = \{b_1^*, \ldots, b_n^*\}$$

y los coeficientes de Gram–Schmidt

$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}, \quad 1 \leq j < i \leq n,$$

de forma que

$$b_i = b_i^* + \sum_{j=1}^{i-1} \mu_{i,j} \, b_j^*.$$

Definición 4.35 (Base reducida por tamaño)**.** Una base $B = \{b_1, \ldots, b_n\}$ se dice reducida por tamaño si

$$|\mu_{i,j}| \leq \frac{1}{2}, \quad \text{para todo } 1 \leq j < i \leq n.$$

Es decir, cada vector b_i ha sido ajustado mediante combinaciones lineales con los vectores anteriores para que sus proyecciones sobre b_j^* sean pequeñas.

Definición 4.36 (Base LLL-reducida)**.** Sea $B = \{b_1, \ldots, b_n\}$ una base de un retículo y $B^* = \{b_1^*, \ldots, b_n^*\}$ sus vectores de Gram–Schmidt, con coeficientes $\mu_{i,j}$.

Fijamos un parámetro δ con

$$\frac{1}{4} < \delta < 1,$$

típicamente $\delta = \frac{3}{4}$.

Decimos que B es una base LLL-reducida (con parámetro δ) si se cumplen las dos condiciones:

1. B es reducida por tamaño, es decir

$$|\mu_{i,j}| \leq \frac{1}{2} \quad \text{para todo } 1 \leq j < i \leq n;$$

2. (Condición de Lovász) para todo $k = 2, \ldots, n$ se cumple

$$\delta \, \|b_{k-1}^*\|^2 \leq \|b_k^*\|^2 + \mu_{k,k-1}^2 \, \|b_{k-1}^*\|^2.$$

En el caso habitual $\delta = \frac{3}{4}$, la condición de Lovász puede escribirse de forma equivalente como

$$\|b_k^*\|^2 \geq \left(\frac{3}{4} - \mu_{k,k-1}^2\right) \|b_{k-1}^*\|^2.$$

La condición de Lovász impone que las normas de los vectores b_k^* no decrezcan demasiado rápido al aumentar k. En particular, como

$$|\mu_{k,k-1}| \leq \frac{1}{2},$$

se tiene

$$\|b_k^*\|^2 \geq \left(\frac{3}{4} - \frac{1}{4}\right) \|b_{k-1}^*\|^2 = \frac{1}{2}\|b_{k-1}^*\|^2.$$

Es decir,

$$\|b_k^*\|^2 \geq \frac{1}{2} \|b_{k-1}^*\|^2.$$

Antes de definir qué es una base reducida para LLL, necesitamos ver qué relación hay entre los mínimos sucesivos y los vectores de Gram–Schmidt.

Teorema 4.37. *Sea L un retículo con base $B = \{b_1, \ldots, b_n\}$ y vectores de Gram–Schmidt $B^* = \{b_1^*, \ldots, b_n^*\}$. Entonces*

$$\lambda_1(L) \geq \min_{1 \leq i \leq n} \|b_i^*\|.$$

Demostración. Sea $v \in L$ un vector no nulo. Podemos escribirlo como

$$v = z_1 b_1 + \cdots + z_n b_n, \quad z_i \in \mathbb{Z}.$$

Sea k el índice máximo tal que $z_k \neq 0$. Usando la descomposición en la base de Gram–Schmidt,

$$b_k = b_k^* + \sum_{j=1}^{k-1} \mu_{k,j}\, b_j^*,$$

se obtiene que la componente de v en la dirección de b_k^* es

$$z_k\, b_k^*.$$

Por tanto, la proyección de v sobre la recta generada por b_k^* tiene norma

$$\|z_k\, b_k^*\| = |z_k|\, \|b_k^*\| \geq \|b_k^*\|,$$

ya que z_k es un entero no nulo. Como la norma de v es, al menos, la norma de cualquiera de sus componentes ortogonales, se cumple

$$\|v\| \geq \|b_k^*\| \geq \min_{1 \leq i \leq n} \|b_i^*\|.$$

Esto vale para cualquier $v \in L$ no nulo, así que

$$\lambda_1(L) = \min_{v \in L \setminus \{0\}} \|v\| \geq \min_{1 \leq i \leq n} \|b_i^*\|.$$

□

Podemos usar este teorema, junto con la condición de Lovász, para acotar la longitud del primer vector de una base LLL-reducida, como hace el paper original en [17].

Teorema 4.38. *Sea L un retículo de rango n y sea $B = \{b_1, \ldots, b_n\}$ una base LLL-reducida de L con parámetro $\delta = \frac{3}{4}$. Entonces*

$$\|b_1\| \leq 2^{\frac{n-1}{2}} \lambda_1(L).$$

Demostración. Por la observación anterior, para cada $i = 2, \ldots, n$ se cumple

$$\|b_i^*\|^2 \geq \frac{1}{2} \|b_{i-1}^*\|^2.$$

Invirtiendo la desigualdad obtenemos

$$\|b_{i-1}^*\|^2 \leq 2 \|b_i^*\|^2.$$

Aplicando esto de forma iterada,

$$\|b_1^*\|^2 \leq 2 \|b_2^*\|^2 \leq 2^2 \|b_3^*\|^2 \leq \cdots \leq 2^{n-1} \|b_n^*\|^2.$$

En particular, para cualquier i,

$$\|b_1^*\|^2 \leq 2^{i-1} \|b_i^*\|^2,$$

de donde

$$\|b_1^*\|^2 \leq \max_{1 \leq i \leq n} 2^{i-1} \|b_i^*\|^2 \leq 2^{n-1} \max_{1 \leq i \leq n} \|b_i^*\|^2.$$

Como $\|b_1\| = \|b_1^*\|$ y

$$\lambda_1(L) \geq \min_{1 \leq i \leq n} \|b_i^*\|,$$

se tiene

$$\lambda_1(L)^2 \geq \min_{1 \leq i \leq n} \|b_i^*\|^2 \geq \frac{1}{2^{n-1}} \|b_1^*\|^2.$$

Reordenando,

$$\|b_1\|^2 = \|b_1^*\|^2 \leq 2^{n-1} \lambda_1(L)^2,$$

lo que implica

$$\|b_1\| \leq 2^{\frac{n-1}{2}} \lambda_1(L).$$

□

Este resultado muestra que el primer vector de una base LLL-reducida es una aproximación al vector más corto del retículo.

El algoritmo LLL combina dos operaciones básicas sobre la base:

- reducción de tamaño, que garantiza $|\mu_{i,j}| \leq \frac{1}{2}$;
- intercambios (swap) de vectores, que fuerzan el cumplimiento de la condición de Lovász.

Reducción de tamaño

Dada una base $B = \{b_1, \ldots, b_n\}$, la reducción de tamaño ajusta cada vector b_i restando múltiplos enteros de los vectores anteriores, con el objetivo de que las proyecciones sobre

los b_j^* sean pequeñas.

Algorithm 4 Reducción de tamaño

Require: Base $B = \{b_1, \ldots, b_n\} \subset \mathbb{R}^m$
Ensure: Base B reducida por tamaño
1: Calcula los vectores de Gram–Schmidt $b_1^*, \ldots, b_n^*$ y los coeficientes $\mu_{i,j}$.
2: **for** $i = 2$ **to** n **do**
3: **for** $j = i - 1$ **downto** 1 **do**
4: $\mu \leftarrow \mu_{i,j}$
5: $k \leftarrow \text{round}(\mu)$ ▷ redondeo al entero más cercano
6: **if** $k \neq 0$ **then**
7: $b_i \leftarrow b_i - k\, b_j$
8: **end if**
9: **end for**
10: **end for**
11: **return** B

Tras ejecutar este procedimiento, la base resultante B verifica $|\mu_{i,j}| \leq \frac{1}{2}$ para todos los índices, es decir, es reducida por tamaño.

Intercambio de vectores

La segunda operación comprueba la condición de Lovász para índices consecutivos y, cuando esta falla, intercambia los vectores correspondientes de la base.

Algorithm 5 Operación de intercambio

Require: Base reducida por tamaño $B = \{b_1, \ldots, b_n\}$, parámetro $\delta \in (1/4, 1)$
Ensure: Base B tras aplicar un intercambio
1: Calcula los vectores $b_1^*, \ldots, b_n^*$ y coeficientes $\mu_{i,j}$
2: **for** $k = 2$ **to** n **do**
3: **if** $\delta \, \|b_{k-1}^*\|^2 > \|b_k^*\|^2 + \mu_{k,k-1}^2 \|b_{k-1}^*\|^2$ **then**
4: intercambia b_{k-1} y b_k
5: **break**
6: **end if**
7: **end for**
8: **return** B

El intercambio hace que la base se acerque a cumplir la condición de Lovász para todos los índices. Sin embargo, después de un intercambio la base deja de estar reducida por tamaño, por lo que es necesario volver a aplicar la reducción de tamaño.

Algoritmo LLL

El algoritmo LLL repite alternadamente la reducción de tamaño y los posibles intercambios de vectores hasta que la base cumple simultáneamente las dos condiciones de la definición de base LLL-reducida.

Algorithm 6 Algoritmo LLL

Require: Base inicial $B = \{b_1, \ldots, b_n\} \subset \mathbb{Z}^m$, parámetro $\delta \in (1/4, 1)$
Ensure: Base LLL-reducida B del mismo retículo

```
Calcula B* y los coeficientes μ_{i,j}
k ← 2
while k ≤ n do
    for j = k − 1 downto 1 do
        k_j ← round(μ_{k,j})
        if k_j ≠ 0 then
            b_k ← b_k − k_j b_j
            actualiza b*_k y μ_{k,·}
        end if
    end for
    if δ ‖b*_{k−1}‖² > ‖b*_k‖² + μ²_{k,k−1} ‖b*_{k−1}‖² then
        intercambia b_{k−1} y b_k
        recalcula b*_{k−1}, b*_k y los coeficientes correspondientes
        k ← max(2, k − 1)
    else
        k ← k + 1
    end if
end while
return B
```

Ejemplo 4.39. Para el algoritmo LLL, usaremos un ejemplo sencillo con una base de retículo en $\mathbb{Z}^2$:

- $b_1 = (1, 1)$
- $b_2 = (1, 0)$

Usaremos $\delta = 0.75$ para la condición LLL.

El primer paso es hacer la ortogonalización de Gram-Schmidt:

- $b_1^* = b_1 = (1, 1)$.
- $\mu_{2,1} = \dfrac{\langle b_2, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \dfrac{1 \cdot 1 + 0 \cdot 1}{1^2 + 1^2} = \dfrac{1}{2}$.
- $b_2^* = b_2 - \mu_{2,1} \cdot b_1^* = (1, 0) - \dfrac{1}{2} \cdot (1, 1) = (0.5, -0.5)$.

Comprobamos la condición de reducción de tamaño: como $|\mu_{2,1}| = 1/2$, que es igual al límite $1/2$, no neceistamos reducir el tamaño.

Verificamos si $\|b_2^*\|^2 \geq \delta \cdot \|b_1^*\|^2$:

- $\|b_1^*\|^2 = 1^2 + 1^2 = 2$.
- $\|b_2^*\| = 0.5^2 + (-0.5)^2 = 0.5$.
- $\delta \cdot \|b_1^*\|^2 = 0.75 \cdot 2 = 1.5$.

Como $0.5 < 1.5$, la condición no se cumple. Por lo tanto, debemos intercambiar b_1 y b_2:

- $b_1 = (1, 0)$
- $b_2 = (0, 1)$

Al repetir el proceso de Gram-Schmidt nos queda que: $B' = \{(1, 0), (0, 1)\}$. Y nos queda verificar la condición LLL:

- $\|b_1^*\|^2 = 1^2 + 0^2 = 1$
- $\|b_2^*\|^2 = 0^2 + 1^2 = 1$
- $\delta \cdot \|b_1^*\|^2 = 0.75 \cdot 1 = 0.75$.

Como $\|b_2^*\|^2 = 1 > 0.75 = \delta \cdot \|b_1^*\|^2$, la condición LLL se cumple y la base reducida LLL es:

$$B' = \{(1, 0), (0, 1)\},$$

que es la base canónica de $\mathbb{Z}^2$, es decir, la base más reducida posible.

■

Este ataque siempre termina y su tiempo de ejecución es polinómico en n:

- La reducción de tamaño no modifica el retículo generado, ni el producto

$$\prod_{i=1}^{n} \|b_i^*\|^2,$$

  que está relacionado con el volumen del retículo.
- Cada operación de intercambio hace disminuir este producto en un factor acotado por una constante menor que 1 (dependiente de δ), de manera que no se pueden realizar intercambios indefinidamente.
- El número total de posibles intercambios está acotado por una función polinómica en n y en el tamaño de las entradas de los vectores de la base.

Por lo tanto, el número de iteraciones del algoritmo es polinómico y cada iteración realiza operaciones aritméticas también polinómicas en n. Por tanto, LLL es un algoritmo rápido para tamaños de entrada pequeños.

4.3. Ataques por fuerza bruta

.

Además del ataque por tríos y de los ataques basados en densidad al SSP, existe un ataque más sencillo que explota el hecho de que los parámetros n (longitud del mensaje) y t (peso del mensaje) son públicos. Recordemos que el texto plano es un vector

$$e = (e_1, \ldots, e_n) \in \{0, \ldots, z - 1\}^n,$$

con exactamente t posiciones no nulas. La idea del atacante es intentar adivinar qué coordenadas son no nulas, y, una vez fijadas esas posiciones, reducir el problema a un SSP de dimensión mucho menor, sobre el que el algoritmo LLL es efectivo.

4.3.0.1. Idea general del ataque

Si el atacante conociera el conjunto de índices

$$S \subset \{1, \ldots, n\}, \quad |S| = t$$

que contienen las entradas no nulas de e, podría restringir el problema únicamente a esas coordenadas. Denotando por t_S el vector formado por las componentes $(t_i)_{i \in S}$, y por e_S el vector de coeficientes desconocidos $(e_i)_{i \in S}$, la primera componente del cifrado verifica

$$c_1 \;=\; \sum_{i=1}^{n} e_i t_i \;=\; \sum_{i \in S} e_i t_i,$$

ya que el resto de entradas son cero. De la misma forma,

$$c_2 \;=\; \sum_{i=1}^{n} e_i \;=\; \sum_{i \in S} e_i.$$

Fijado S, el atacante se enfrenta a un SSP de dimensión t con objetivo (c_1, c_2) y pesos conocidos $(t_S, 1)$. Cuando n y t son pequeños, este problema reducido es vulnerable a un ataque por reducción de retículos mediante LLL.

El ataque por fuerza bruta consiste en recorrer todas las posibilidades de S de tamaño t:

$$\text{número de subconjuntos posibles} \;=\; \binom{n}{t},$$

y, para cada uno, intentar resolver el SSP reducido. Si n es pequeño, la densidad del de la instancia del SSP es baja y el algoritmo LLL puuede salir exitoso.

La solución a este ataque es aumentar el valor del tamaño del mensaje n. De esta forma, el número de combinaciones necesarias para convertir el WSSP al SSP aumenta de forma exponencial y, en caso de que se consiga, la densidad de la instancia del problema sería grande, lo que haría su ataque más costoso.

5. Implementación del criptosistema GLN

5.1. Arquitectura general

La implementación del criptosistema GLN se ha desarrollado íntegramente en C++17, siguiendo una arquitectura modular y extensible. Cada módulo es una parte independiente del sistema que cumple una función concreta, con una interfaz bien definida y que puede cambiarse sin afectar al resto del sistema. Además, el sistema es extensible porque permite añadir nuevas funcionales (nuevos ataques, variantes de cifrado. etc.) sin romper la implementación actual.

Cada módulo se corresponde con un archivo principal del directorio `src/` y una cabecera asociada en `include/gln/`, de forma que el código puede compilarse de forma incremental y reutilizarse fácilmente.

La arquitectura utiliza un patrón en capas:

- Capa matemática (`bigint`, `nt_utils`, `random`): implementa operaciones aritméticas usando enteros grandes, utilidades de teoría de números (generación de primos, inversos modulares, potencias, etc) y generación de enteros aleatorios grandes.
- Capa de parámetros y datos (`params`, `keygen`): define las estructuras `Params`, `PublicKey`, `PrivateKey`, y genera las claves a partir de los parámetros públicos.
- Capa de operaciones criptográficas (`encrypt`, `decrypt`): implementa el algoritmo de cifrado y descifrado.
- Capa de experimentación (`attack`, `experiments`): contiene el ataque por tríos, además del código de benchmarking y evaluación del rendimiento.

5.2. Descripción de módulos

5.2.1. La clase `BigInt`

El módulo `bigint` define el tipo básico de dato entero de precisión arbitraria que se utiliza en todo el criptosistema GLN. Su objetivo es ofrecer una interfaz para operar con enteros muy grandes, así trabajar de forma limpia sin detalles técnicos de representación y eficiencia.

Para ello, la clase `BigInt` encapsula una implementación interna basada en `Boost.Multiprecision` y expone solo las operaciones necesarias para criptografía de enteros en módulos grandes. Esta separación mejora la mantenibilidad del código y permite cambiar la implementación subyacente sin afectar al resto de capas del sistema.

5.2.1.1. Implementación interna con `boost::multiprecision::cpp_int`

La representación numérica se apoya en `boost::multiprecision::cpp_int`, un tipo de la biblioteca Boost que permite trabajar con enteros de tamaño no acotado. A diferencia de los

tipos nativos del compilador, `cpp_int` crece dinámicamente para añadir más bits cuando es necesario, lo que es de gran utilidad en criptografía con módulos g y claves de cientos de miles de bits. Se elige Boost por:

- Ser portable, evitando dependencias externas como `GMP` y se integra bien con C++.
- Es seguro en el manejo de desbordamientos y operaciones de alto nivel.
- Ofrece un rendimiento razonable adecuado para aritmética.

En esta clase se encapsula el único atributo `cpp_int` dentro de una estructura privada y solo expone la interfaz pública necesaria para el resto de módulos.

5.2.1.2. Interfaz pública: constructores, operadores y utilidades

La clase `BigInt` proporciona una interfaz completa para los usos del sistema:

- Constructores: desde datos primitivos `long long`, desde cadenas decimales `string`, y desde buffers o secuencias de bytes.
- Operadores básicos: suma, resta, producto, división, módulo y comparaciones. Están definidos con la semántica habitual en C++, por lo que `BigInt` puede utilizarse de forma natural en expresiones aritméticas.
- Funciones utilitarias: `abs(x)` para el valor absoluto; `sgn(x)` para el signo; `sqrt(x)` para la raíz cuadrada entera; y la conversión a texto `to_string(x)`.

5.2.1.3. Ocultación de detalles con el patrón Plmpl

Para desacoplar la interfaz pública de `BigInt` de su representación interna, el módulo emplea el patrón Pointer to Implementation (Plmpl). La clase pública declara solo un puntero a una estructura `Impl`, que se encarga de la gestión de `cpp_int` y de toda su lógica. Este enforque aporta beneficios como:

- Esconde los detalles de la implementación de la interfaz
- Los cambios en la implementación no fuerzan a recompilar los módulos que la usan, lo que reduce el tiempo de compilación.
- Evita incluir implementaciones específicas, como los headers de Boost.

En Código 5.1 puede verse cómo se separa la interfaz de `bigint.hpp` y la implementación de `bigint.cpp`, que guarda la estructura `Impl` conteniendo el `boost::multiprecision::cpp_int`.

```
// ----------------------------------------------------
// interfaz (bigint.hpp)

class BigInt
{
public:
    // miembros publicos (constructores, operadores, etc.)

private:
```

```
    struct Impl;
    std::unique_ptr<Impl> p_;
};

// ------------------------------------------------------
// implementacion (bigint.cpp)

struct BigInt::Impl
{
    boost::multiprecision::cpp_int v;

    Impl() : v(0) {}
    explicit Impl(long long x) : v(x) {}
    explicit Impl(const std::string& dec) : v(0)
    {
        bool neg = false;
        std::size_t i = 0;
        if (!dec.empty() && dec[0] == '-') { neg = true; i = 1; }
        for (; i < dec.size(); ++i) {
            char c = dec[i];
            if (c >= '0' && c <= '9') { v *= 10; v += (c - '0'); }
        }
        if (neg) v = -v;
    }
};
```

Código 5.1: Ejemplo del patrón PImpl aplicado a la clase `BigInt`

5.2.2. El módulo `nt_utils`

Este módulo se encarga de implementar algoritmos útiles como el test de Miller-Rabin que comprueba si un entero es primo (útil en la generación de claves) o como el Algoritmo de Euclides Extendido Truncado (útil en la descifrado).

Como funciones básicas de este módulo se implementan la suma, resta, producto e inverso modulares. Esta última operación requiere de la implementación del Algoritmo de Extendido de Euclides, tal y como se define en la Subsección 3.4.1.

El algoritmo de descifrado utiliza el TrEEA implementado en este módulo tal y como se definió en el pseudocódigo referenciado. Además, también hace uso de la función gcd, que devuelve el gran común divisor entre dos variables de tipo `BigInt` usando el Algoritmo de Euclides.

Algorithm 7 Máximo común divisor por Euclides

Require: $a, b \in \mathbb{Z}$
Ensure: $\gcd(a, b)$
1: **while** $b \neq 0$ **do**
2: $r \leftarrow a \bmod b$
3: $a \leftarrow b$
4: $b \leftarrow r$
5: **end while**
6: **return** $|a|$

El algoritmo de generacion de claves necesita muchos primos: una familia de n primos distintos p_i para la clave privada. Como los enteros son muy grandes, no es viable precomputarlos, así que hace falta un test rápido que descarte compuestos y devuelva una primalidad probable. En este módulo se implementa el test de Miller-Rabin.

5.2.2.1. Test de primalidad de Miller-Rabin

La idea principal de Miller-Rabin no es demostrar que n es primo, sino que, tras varias rondas, puede afirmar que n es probablemente primo con una probabilidad de error que decrece exponencialmente con el número de bases probadas.

Primero veamos un recordatorio de la base matemática ([19]).

Teorema 5.1 (Pequeño teorema de Fermat)**.** *Sea p primo y $\gcd(a, p) = 1$, entonces:*

$$a^{p-1} \equiv 1 \pmod{n}.$$

Y, en general, este teorema no se cumple para números compuestos.

Como consecuencia se desarrolla el test de Fermat: elegir una base a y comprobar $a^{n-1} \equiv 1 \pmod{n}$. Si falla, n es compuesto. Si pasa, n quizá es primo. Decimos que a es testigo si detecta que n es compuesto, y mentirosa si, siendo n compuesto, la comprobación pasa.

El test de Fermat falla con los números de Carmichael: son enteros compuestos n para los que $a^{n-1} \equiv 1 \pmod{n}$ se cumple para toda base a coprima con n. El primero es $n = 561 = 3 \times 11 \times 17$.

Teorema 5.2 (Raíces cuadradas de 1 en módulo primo)**.** *Si p es primo impar, las únicas soluciones de $x^2 \equiv 1 \pmod{p}$ son $x \equiv -1, +1$.*

Esta propiedad, junto con el hecho de que $\mathbb{Z}/n\mathbb{Z}$ no tiene divisores de cero, es la clave que refuerza el test de Miller-Rabin [21].

Para cualquier impar $n > 2$ se escribe $n - 1 = d \cdot 2^s$, con d impar. De forma que para una base aleatoria a, la congruencia del Pequeño teorema de Fermat queda:

$$a^{d \cdot 2^s} \equiv 1 \pmod{n},$$

cuando n es primo y $\gcd(a, n) = 1$. A partir de aquí, el desarrollo algebraico encadena

factorizaciones de la forma $X^2 - 1 = (X-1)(X+1)$. Se cumple que:

$$\begin{aligned}
a^{d2^s} \equiv 1 \pmod{n} &\iff \left(a^{d2^{s-1}}\right)^2 - 1 \equiv 0 \pmod{n} \\
&\iff \left(a^{d2^{s-1}} - 1\right)\left(a^{d2^{s-1}} + 1\right) \equiv 0 \pmod{n} \\
&\iff \left(\left(a^{d2^{s-2}}\right)^2 - 1\right)\left(a^{d2^{s-1}} + 1\right) \equiv 0 \pmod{n} \\
&\iff \left(a^{d2^{s-2}} - 1\right)\left(a^{d2^{s-2}} + 1\right)\left(a^{d2^{s-1}} + 1\right) \equiv 0 \pmod{n} \\
&\iff \cdots \\
&\iff \left(a^{d} - 1\right)\left(a^{d} + 1\right)\left(a^{d2} + 1\right) \cdots \left(a^{d2^{s-2}} + 1\right)\left(a^{d2^{s-1}} + 1\right) \equiv 0 \pmod{n}.
\end{aligned}$$

La interpretación es la siguiente: si n es primo, entonces n debe dividir a alguno de los factores. Es decir, para algún $i \in \{0, \ldots, s-1\}$ debe cumplirse:

$$a^{d2^i} \equiv -1 \pmod{n},$$

o bien ya desde el inicio $a^d \equiv 1 \pmod{n}$. Si ninguno de esos casos aparece, el patrón que caracteriza el caso primo se rompe y concluimos que n es compuesto con la base a como testigo.

Para n impar compuesto, como máximo una cuarta parte de las bases en $\{2, \ldots, n-2\}$ son mentirosas. Tras k bases independientes, la probabilidad de que un compuesto pase todas las rondas es $\leq 4^{-k}$. En la práctica, unas pocas bases son suficientes.

El pseudocódigo de la implementación del test de primalidad de Miller-Rabin es el siguiente:

Algorithm 8 Exponenciación modular rápida PowMod

Require: $a, e, m \in \mathbb{Z}$ con $m > 0$
Ensure: $r \equiv a^e \pmod{m}$

```
1: r ← 1
2: a ← a mod m
3: while e > 0 do
4:     if e mod 2 = 1 then
5:         r ← MulMod(r, a, m)
6:     end if
7:     e ← ⌊e/2⌋
8:     a ← MulMod(a, a, m)
9: end while
10: return r
```

Algorithm 9 Test de primalidad probable de Miller–Rabin

Require: $n \in \mathbb{Z}$ impar, $n \geq 3$; *rounds* $\in \mathbb{N}$
Ensure: **true** si n es probablemente primo, **false** si es compuesto

if $n \leq 1$ **then**
  return false
end if
if $n = 2$ **or** $n = 3$ **then**
  return true
end if
if $n \bmod 2 = 0$ **then**
  return false
end if
$d \leftarrow n - 1, s \leftarrow 0$
while $d \bmod 2 = 0$ **do**
  $d \leftarrow d/2$
  $s \leftarrow s + 1$
end while
$\mathcal{B} \leftarrow \langle 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 \rangle$
use $\leftarrow \min(\textit{rounds}, |\mathcal{B}|)$
for $i \leftarrow 1$ **to** *use* **do**
  $a \leftarrow \mathcal{B}[i]$
  if $a \geq n - 2$ **then**
    continue
  end if
  $x \leftarrow \textsc{PowMod}(a, d, n)$
  if $x = 1$ **or** $x = n - 1$ **then**
    continue ▷ esta base no contradice primalidad
  end if
  pasa $\leftarrow$ **false**
  for $r \leftarrow 1$ **to** $s - 1$ **do**
    $x \leftarrow \textsc{MulMod}(x, x, n)$
    if $x = n - 1$ **then**
      pasa $\leftarrow$ **true**
      break
    end if
  end for
  if not *pasa* **then**
    return false ▷ se detecta compuesto
  end if
end for
return true ▷ probablemente primo

5.2.3. El módulo `Params`

Este módulo encapsula los parámetros de alto nivel del esquema y fija, a partir de entradas sencillas (n, t, z, β), los tamaños mínimos coherentes con la implementación.

```
35 // ------------------------------------------------------
36 // interfaz (params.hpp)
37 struct Params {
38   int n, t, z;
39   int b, ell, beta;
40   int bits_g;
41 };
42
43
44 Params choose_params(int n, int t, int z, int beta);
```

Código 5.2: Estructura Params

Los parámetros n, t, z son el tamaño del mensaje, el peso del mensaje y el tamaño del alfabeto usado, respectivamente. Por otro lado, la variable β define la ventana de bits para los p_i, ya que los primos se toman con longitudes en $[b+1, b+\beta]$.

Los valores de b y ell son el número de bits necesarios para representar z y t, respectivamente. Se calculan como:

$$b = \lceil \log_2 z \rceil, \quad ell = \lceil \log_2 t \rceil.$$

Finalmente, el valor clave calculado con la función `choose_params` es el tamaño en bits del módulo g, ya que condiciona la corrección del esquema.

5.2.3.1. Cálculo de los bits del módulo g

Para definir un rango seguro de bits de g cabe recordar la corrección de GLN (el Teorema 3.18). Aquí se definen las cotas inferiores para el módulo g

$$g \geq 4\lambda_{max}^2, \quad g > 4\omega_{max}^2,$$

donde $\lambda_{max} = \prod_{j=1}^{t} p_{k_j}$, el producto de los t primos más grandes de la clave privada. Y $\omega_{max} = \sum_{j=1}^{t} e_j \frac{\lambda_{max}}{p_{k_j}}$, donde e_{k_j} son las componentes del texto plano.

Para acotar inferiormente los bits de g debemos acotar superiormente los bits de λ_{max} y ω_{max}, que dependen principalmente de los primos de la clave privada. Sabemos que la clave privada esta formada por n primos $p_1, \ldots, p_n$ y cada uno tiene entre $b+1$ bits (para asegurar que $p_i > z-1$) y $b+\beta$ bits.

Para asegurar que existe un número suficiente de primos en el rango $[b+1, b+\beta]$, podemos utilizar el Teorema de los Números Primos.

Por lo tanto,

$$\lambda_{max} \leq 2^{t(b+\beta)}.$$

Para acotar ω_{max}, podemos ver que dividir λ_{max} por uno de los t primos más pequeños, elimina en el peor caso, un factor de orden $2^{b+\beta}$. Por tanto,

$$\frac{\lambda_{max}}{p_{k_j}} \leq \frac{2^{t(b+\beta)}}{2^{b+\beta}} = 2^{t(b+\beta)-b-\beta}.$$

Como $e_j < 2^b$, tenemos

$$e_j \cdot \frac{\lambda_{max}}{p_{k_j}} \leq 2^b \cdot 2^{t(b+\beta)-b-\beta} = 2^{t(b+\beta)-\beta}.$$

La suma de t cantidades de tamaño anterior añade, en bits, a los sumo $\log_2(t) \leq ell$. Nos queda entonces,

$$\omega_{max} \leq t \cdot 2^{t(b+\beta)-\beta} \leq 2^{t(b+\beta)+ell-\beta}.$$

Para la condición de g tomamos logaritmos base 2 y usamos las cotas de 5.2.3.1 y 5.2.3.1 para obtener:

$$bits\_g \geq \max\{2t(b+\beta)+2, 2(t(b+\beta)+ell-\beta)+2\}.$$

Al implementarlo, se añade un pequeño margen constante y se usa que:

$$bits\_g = \max\{2t(b+\beta)+3, 2(t(b+\beta)+ell-\beta)+3\}.$$

5.2.4. El módulo `random`

El módulo `random` proporciona la capa de generación de números aleatorios para todo el criptosistema GLN. En criptografía, la generación de valores aleatorios de alta calidad es esencial para la creación de claves y de pruebas. Para ello, este módulo encapsula el uso del generador estándar de C++ y define una interfaz para producir valores uniformemente distribuidos, bytes aleatorios y números grandes `BigInt` de tamaño arbitrario.

Internamente, la clase `Random` utiliza el motor `std::mt19937_64`, una implementación del algoritmo Mersenne Twister de 64 bits. Este motor ofrece una buena combinación entre velocidad, uniformidad estadística y facilidad de uso. Aunque no es un generador criptográficamente seguro, resulta suficiente para la experimentación, simulaciones y validación funcional del criptosistema.

La clase dispone de dos constructores: uno por defecto, que inicializa el motor usando `std::random_device`, que toma la entropía del hardware; y un constructor con semilla explícita, que permite fijar una semilla concreta. de est amenra, la clase puede funcionar tanto en contextos de producción (semillas impredecibles) como en entornos de test (semillas fijas).

5.2.4.1. Generación de bytes aleatorios

La función `random_bytes(uint8_t* dst, int n)` rellena un búfer de tamaño `n` con bytes aleatorios generados a partir del motor `eng`. Cada byte se obtiene aplicando una máscara de 8 bits (`eng() 0xFF`) al valor entero de 64 bits producido por el generador. Este método es simple y eficiente, y constituye la base para las demás funciones que generan enteros grandes o distribuciones uniformes sobre intervalos.

Esta función es especialmente útil cuando se requiere material aleatorio bruto, por ejemplo, para inicializar claves, semillado de otros generadores o creación de números grandes con una longitud específica de bits.

5.2.4.2. Generación de números grandes aleatorios

La función `random_bits(int bits)` extiende la capacidad de generación de enteros hacia valores de tamaño arbitrario mediante el uso del tipo BigInt. Su propósito es obtener un

número uniforme en el intervalo $[0, 2^{bits} - 1]$, con la misma probabilidad para cada posible valor de esa longitud.

El proceso consiste en:

1. Calcular el número de bytes necesarios: $bytes = \lceil bits/8 \rceil$.
2. Rellenar un búfer de esa longitud con bytes aleatorios usando `random_bytes`.
3. Si el número total de bits generados excede la longitud solicitada, se limpian los bits más altos sobrantes del primer byte (`buf[0] &= 0xFF » extra`), de modo que el resultado final no supere el rango definido.
4. Construir un objeto BigInt a partir del búfer de bytes resultante.

De esta manera, `random_bits` permite generar valores grandes de tamaño controlado, que sirven como base para otros procedimientos como la creación de números con un número exacto de bits.

5.2.4.3. La función `random_of_bitlen`

Esta función tiene como objetivo muestrear un entero con longitud de bits exacta (uniforme en $[2^{bits-1}, 2^{bits} - 1]$) y forzarlo impar. Para ello, primero crea un valor uniforme entre $[0, 2^{bits} - 1]$. A continuación, mapea la mitad inferior ($[0, 2^{bits-1} - 1]$) sobre $[2^{bits-1}, 2^{bits} - 1]$, sumando 2^{bits-1} a cada valor del primer rango. Para forzarlo impar, se añade 1.

5.2.4.4. La función `random_prime_bits`

Esta función genera un impar aleatorio de bits (pasado como parámetro) y busca el siguiente primo mayor o igual a éste. Para ello, genera un valor aleatorio usando la función `random_of_bitlen` y va iterando cada dos valores y comprobando si es primo con el test de primalidad de Miller-Rabin implementado en `nt_utils` (Subsubsección 5.2.2.1).

5.2.5. El módulo keygen

Este módulo se encarga de definir las estructuras que guardan las claves (pública y privada) y de generar sus valores. Se definen de la siguiente forma:

```
45 // ------------------------------------------------------
46 // interfaz (keygen.hpp)
47 struct PublicKey { vector<BigInt> t; BigInt g; };
48 struct PrivateKey { vector<BigInt> p; BigInt g, u; };
49 struct KeyPair { PublicKey pk; PrivateKey sk; };
50
51 KeyPair keygen(const Params& prm, Random& rng);
```

Código 5.3: Estructuras de Claves

Para la clave privada se necesitan t primos con bits entre $[b + 1, b + \beta]$. Por lo tanto, una estrategia es alternar entre tamaños en este rango y llamar a la función `random_prime_bits`, que recibe como argumento el número de bits deseado y devuelve un primo que lo cumpla.

A continuación, se elige un valor g aleatorio con los bits indicados en la estructura de `Params`. Para ello, se utiliza la función `random_of_bitlen`. Se llama esta función hasta que el valor devuelto sea coprimo con todos los valores de p_{k_j} generados anteriormente.

A continuación, se generan los $h_i = p_{k_j}^{-1} \pmod{g}$ usando las operaciones implementadas en la clase `BigInt`. También se genera el valor u aleatorio menor que g y diferente de cada h_i.

Los valores generados (p_{k_j}, g, u) se guardan en una variable de tipo `PrivateKey`.

Finalmente, para la clave pública se generan los t_i que se guardarán en una variable de tipo `PublicKey`.

La función `keygen` devuelve un `KeyPair` con las dos variables creadas anteriormente.

5.2.6. El módulo encrypt

Este módulo se encarga de encriptar un mensaje cuando recibe la clave pública y devuelve un texto cifrado. Para ello, define la siguiente estructura.

```
52 // ------------------------------------------------------
53 // interfaz (encrypt.hpp)
54 struct Ciphertext {
55     BigInt c1;
56     BigInt c2;
57 };
58 Ciphertext encrypt(vector<int> e, PublicKey pk);
```

Código 5.4: Estructuras de Texto Cifrado

El método `encrypt`, calcula c_1 y c_2 tal y como se detalla en la Subsección 3.3.2.

5.2.7. El módulo decrypt

Este módulo tiene una única función:

```
59 // ------------------------------------------------------
60 // interfaz (decrypt.hpp)
61 vector<int> decrypt(Ciphertext ct, PrivateKey sk, Params prm);
```

Código 5.5: Descifrado

Se encarga de aplicar todo el algoritmo de descifrado detallado en la Subsección 3.3.3 y devolver un vector de enteros, que es el texto plano descifrado.

5.2.8. El módulo attack

El módulo ataca al criptosistema usando la clave pública y genera un reporte con los resultados.

```
// ------------------------------------------------------
// interfaz (attack.hpp)
struct AttackReport {
  BigInt g_guess;
  bool   found_exact = false;
  int combos_tested = 0;
};

AttackReport attack_triples(PublicKey pk, int b, int beta);
```

Código 5.6: Estrutura de ataque

El reporte incluye el valor del mejor g encontrado, un booleano que indica si es exactamente el valor buscado y cuántos combos se han testeado.

Este módulo implementa un ataque concreto: el ataque por tríos. La idea es muy simple: iterar por todas las posibles primos en el rango $[2^{b+1}, 2^{b+\beta}]$ y por todos los valores de la clave pública t_i tres a tres, tal y como se explica en la Sección 4.1.

5.3. Entorno experimental

Hardware

- CPU: `AMD Ryzen 7 7735HS with Radeon Graphics`
- Núcleos físicos: `8`
- Hilos lógicos: `16`
- Hyperthreading: activado con `2` threads per core.
- Memoria RAM: `30 GiB`

Software y compilación

- Sistema operativo: `Ubuntu 24.04.3 LTS`
- Compilador C++: `g++ 13.3.0`
- Boost: `1.74`
- El código se encuentra en `https://github.com/carmenazorinm/GLN`.

5.4. Métricas

Una configuración es una asignación de valores a los parámetros (n, t, z, β) necesarios para generar un experimento que incluya la generación de las claves, la generación de un texto plano, y ejecutar los algoritmos de cifrado y descifrado.

Para medir el rendimiento del criptosistema se han utilizado una serie de métricas que miden tanto la eficiencia en tiempo, como el tamaño de las claves, las longitudes del mensaje y si el proceso fue correcto.

A continuación se detallan las métricas principales usadas.

1. `keygen_s`, `enc_s`, `dec_s`

Son los tiempos (en segundos) para la generación de claves, cifrado y descifrado.

Estas operaciones son las más básicas de cualquier criptosistema: miden la eficiencia computacional y permiten comparar el rendimiento entre distintas configuraciones.

- `keygen` mide el el tiempo de preparar las claves.
- `encrypt` mide el tiempo de cifrar un mensaje.
- `decrypt` mide el tiempo de descifrar.

2. `pubkeys_bits`

Es el tamaño en bits de la clave pública (la suma de los bits de cada t_i con $i = 1, \cdots, n$). Cada experimento devuelve `pubkey_bits`.

El tamaño de la clave pública determina el espacio de almacenamiento necesario, el ancho de banda para transmitir claves y la comparabilidad frente a otros criptosistemas.

4. `n_primes`

Es la aproximación del número de primos posibles generables dentro del rango dado, es decir, en $[2^b, 2^{b+\beta})$. Se obtiene usando la función de densidad $\pi(x)$ dada por el Teorema de los Números Primos (Teorema 4.1). Este número mide cuántos primos distintos se pueden usar para generar caves distintas bajo una misma configuración.

Es importante porque determina la entropía total del espacio de claves privadas: a mayor número de primos posibles, más difícil es realizar ataques combinatorios como el ataque por tríos.

5. `sec_triples`

Es la estimación del nivel de seguridad teórico en bits frente a un ataque por tríos. Los bits de seguridad se calculan como:

$$\text{sec\_triples} = 3\log_2(\text{cant\_prim}) - \log_2(6),$$

donde cant_prim es el número de primos disponibles en $[2^b, 2^{b+\beta})$, que se aproxima con el Teorema de los Números Primos (Teorema 4.1).

Para entender la fórmula, cabe recordar que el ataque por tríos al criptosistema GLN consiste en probar combinaciones de 3 de los primos posibles entre el rango mencionado anteriormente. El número de combinaciones posibles es

$$N_{triples} = \binom{\text{cant\_prim}}{3} = \frac{\text{cant\_prim}(\text{cant\_prim}-1)(\text{cant\_prim}-2)}{6}.$$

El nivel de seguridad frente a este ataque se mide como el $\log_2$ del número de operaciones totales. Por tanto:

$$\text{sec\_triples} = \log_2 \binom{\text{cant\_prim}}{3}.$$

Desarrollando la combinación:

$$\binom{\text{cant\_prim}}{3} = \frac{\text{cant\_prim}^3}{6} \cdot \left(1 - \frac{3}{\text{cant\_prim}} + \frac{2}{\text{cant\_prim}^2}\right).$$

Para n grande, eso se aproxima por $n^3/6$, y nos queda:

$$\text{sec\_triples} \approx \log_2(\text{cant\_prim}^3/6) = 3\log_2(\text{cant\_prim}) - \log_2(6).$$

6. `sec_brute_force`

Es la estimación del nivel de seguridad teórico en bits frente a un ataque por fuerza bruta. Los bits de seguridad frente a este ataque se calculan como:

$$\text{sec\_brute\_force} = \sum_{i=1}^{t} (\log_2(n-t+i) - \log_2(i)) \tag{5.1}$$

donde n es el tamaño del mensaje y t es la cantidad de componentes no nulas en él.

Para entender la fórmula, cabe recordar que el ataque por fuerza bruta a GLN consiste en probar todas las posibles combinaciones de t posiciones en un vector de n elementos, esto es

$$\binom{n}{t} = \frac{n!}{t!(n-t)!} = \frac{(n-t+1)(n-t+2)\cdots(n-t+t)}{1\cdot 2\cdots t}.$$

Luego, el nivel de seguridad se calcula como el $\log_2$ de esta cantidad de operaciones, lo que resulta en la (5.1).

7. `density`

La densidad de la instancia se calcula como

$$\text{density} = \frac{n}{\log_2(g)},$$

lo que permite analizar el ataque LLL, cuyo éxito depende básicamente en esta medida.

5.5. Experimentación de rendimiento

Para la medición del rendimiento general del sistema se ha llevado a utilizado el módulo `experiments`, cuyo objetivo es cuantificar el coste temporal de las tres fases del esquema: generación de claves, cifrado y descifrado bajo distintas configuraciones de parámetros.

El ejecutable `main`, imprime los resultados en un formato legible: JSON o CSV (según indicado) para el análisis de los resultados. Y en cada experimento, el ejecutable verifica la corrección funcional comprobando que `decrypt(encrypt(e)) = e`.

5.5.1. Compilación

La implementación del criptosistema se encuentra en el repositorio de GitHub, ver Párrafo 5.3. Para usarlo, se debe clonar el repositorio localmente y seguir las instrucciones del archivo `README.md`.

La compilación se realiza usando el archivo `CMakeLists.txt`, que permite compilar el programa y los tests independientemente del entorno operativo en el que se encuentre el usuario.

Este archivo se encarga de:

1. Manejar las dependencias con la librería externa `Boost`,

2. definir los archivos que deben ser compilados,
3. crear los ejecutables especificados (`main` y los tests).

5.5.2. Configuración por línea de comandos

El binario generado tras la compilación acepta argumentos que fijan la configuración experimental:

- `-n, -t, -z, -beta`: definen los parámetros de cada configuración experimental.
- `-seed`: semilla para reproducibilidad.
- `-csv, -json`: selecciona el formato de salida.
- `-quiet`: reduce el ruido en consola, útil cuando se redirige la salida a un fichero.

Esto permite ejecutar barridos sistemáticos de configuraciones con scripts, por ejemplo en Python.

5.5.3. Salidas del programa

El programa imprime todas las métricas medidas para cada configuración.

Salida general La salida general del programa al ejecutar el comando:

```
./main -n 10 -t 3 -z 1024 -beta 1 -seed 123
```

```
Parameters: n=10 t=3 z=1024 beta=1 g  71   bits seed=123
KeyGen completed in 0.002854 s
Plaintext (sparse) generated (showing non-zero entries): [2:404, 4:568,
    9:956]
Encryption completed in 0.000006 s
Ciphertext: c1 = 946181556283540302817300  c2 = 1928
Decryption completed in 0.000200 s
Recovered plaintext non-zero entries: [2:404, 4:568, 9:956]
[OK] decrypt(encrypt(e)) == e
SUMMARY: n=10 t=3 z=1024 g_bits=22 c1_bits=24 c2_bits=4 pubkey_bits=210
    keygen_s=0.002854 enc_s=0.000006 dec_s=0.000200 ok=1 n_primes
    =120.871613 sec_triples=18 sec_brute_force=6 density=0.140845
```

Código 5.7: Salida por defecto

Salida CSV La salida CSV incluye todas las métricas necesarias para analizar los resultados de seguridad bajo ataques combinatorios y de rendimiento del criptosistema. Estas medidas incluyen:

- Paramétros de la configuración n, t, z y semilla,
- bits del módulo grande g,
- bits del cifrado (c_1, c_2),

- bits de la clave pública (t_i),
- tiempo en segundos de cada fase (generación de claves, cifrado y descifrado),
- una flag $(0/1)$ indicando si el descifrado ha sido exitoso,
- la cantidad de primos disponibles en el rango $[2^b, 2^{b+\beta})$,
- la seguridad en bits frente al ataque por tríos, y
- la seguridad en bits frente al ataque por fuerza bruta.

La salida de ejecutar el programa con el comando:

```
./main -n 10 -t 3 -z 1024 -beta 1 -seed 123 -csv -quiet
```

```
CSV
    ,10,3,1024,1,123,22,24,4,210,0.001990,0.000003,0.000097,1,120.871613,18,6
```

Código 5.8: Salida CSV

Salida JSON El formato JSON es útil en el caso de atacar usando el algoritmo LLL. La idea es crear un mensaje en formato JSON con la clave pública, el texto cifrado, el texto plano y la densidad de la instancia. Es útil en scripts que leen esta salida e intentan atacar el sistema usando algoritmos ya implementados, como LLL en Sage. Con el texto plano se puede comprobar que el ataque ha sido exitoso.

La salida de ejecutar el programa con el comando:

```
./main -n 10 -t 3 -z 1024 -beta 1 -seed 123 -json -quiet
```

```
{"h": [115484977857227926279,...,289809394181788142114],

"ciphertext": { "c1": "946181556283540302817300", "c2": 1928},

"plaintext": [0,0,404,0,568,0,0,0,0,956],

"d": 0.140845}
```

Código 5.9: Salida JSON

5.5.4. Flujo de ejecución

1. Parámetros: se leen los argumentos y se construye `Params`.
2. Generación de claves: se mide `keygen(prm, rng)`.
3. Texto plano: se genera con la función `random_plaintext(rng,prm)` mostrando únicamente entradas no nulas.
4. Cifrado: se mide `encrypt(e, pk, prm)` y se imprime el criptograma.
5. Descifrado: se mide `decrypt(ct, sk, prm)` y se validan las entradas recuperadas.
6. Reporte: se imprime la salida en CSV o JSON según flags.

6. Experimentación del criptosistema

6.1. Análisis de seguridad frente a ataques combinatorios

La experimentación se ha llevado a cabo llamando al programa `experiments` desde un script de Python, automatizando la ejecución masiva y reproducible del binario sobre muchas configuraciones de parámetros, recogiendo los resultados CSV. Además, se paralelizan las ejecuciones y se repite cada configuración 10 veces.

Los experimentos cubren combinaciones representativas de los parámetros del esquema:

- longitudes de mensaje $n \in 50, 100, 150, 200, 150, 300, 350, 400$,
- pesos $t \in 0.2n, 0.4n, 0.6n, 0.8n$,
- tamaños de alfabeto $z \in 2^{10}, \ldots, 2^{40}$ y
- β calculado según $\beta = 2 + \lceil \log_2(n) \rceil$, para asegurar un número suficiente de primos.

6.1.1. Gráfico de pares entre parámetros

En la Figura 6.1 se evidencian los parámetros usados para realizar los experimentos. Vemos que:

- El tamaño del texto n va desde 50 hasta 400 y se ejecuta cada valor el mismo número de veces.
- El tamaño de t depende del valor de n, ya que se han ejecutado los experimentos según el peso del mensaje. Por ello la gráfica n-t muestra que hay cuatro valores distintos para cada cada valor de n.
- El tamaño del alfabeto z se ha iterado desde 2^{10} hasta 2^{40} y se han ejecutado cada valor el mismo número de veces.
- El peso del mensaje se ha ejecutado con valores desde 0.2 hasta 0.8, el mismo número de veces para cada valor.

Cada posible combinación de n, z y el peso se ha ejecutado 10 veces.

6.1.2. Bits de seguridad para ataque por tríos

En la Figura 6.2 se muestra:

- En el eje X, el tamaño del alfabeto en bits ($\log_2(z)$). Cuanto más a la derecha, mayor es el alfabeto.
- En el eje Y, el tamaño del mensaje (n). Cuanto más abajo, más grande es su valor.

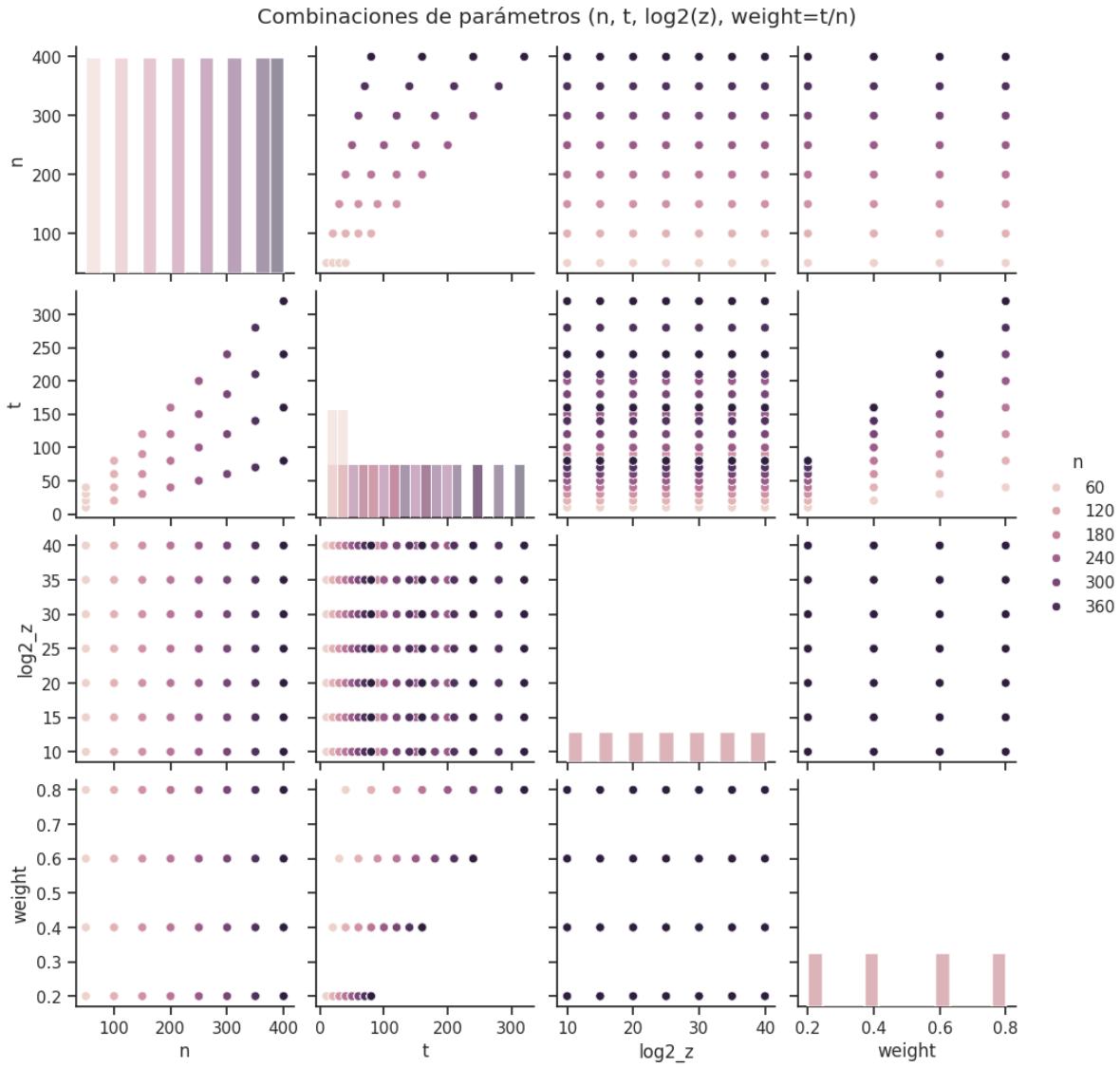


Figura 6.1.: Gráfico de pares para n, t, $log_2(z)$ y peso

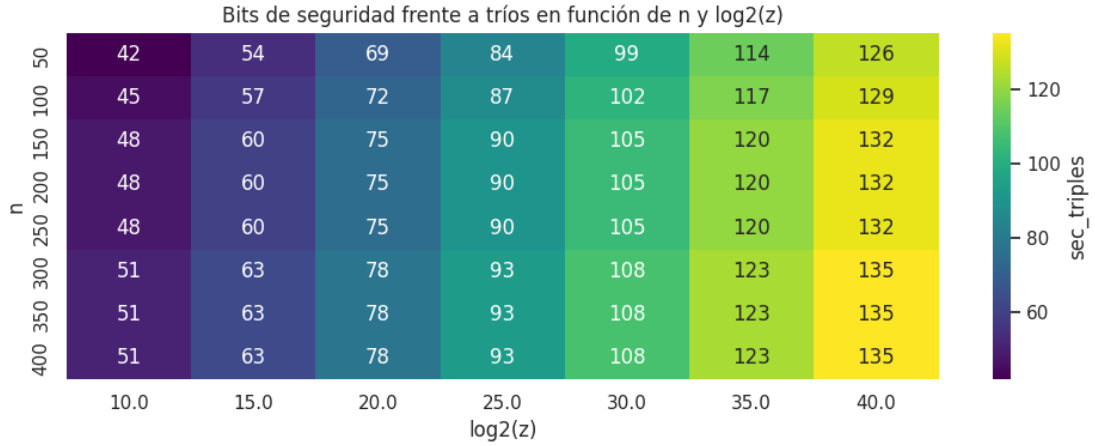


Figura 6.2.: Heatmap de bits de seguridad para ataque por tríos

- Los bits de seguridad frente al ataque por tríos se representa con los colores. Un color más claro (amarillo) indica más seguridad que un color oscuro (morado).

Podemos ver que hay una clara dependencia de los bits de seguridad por tríos en función del tamaño del alfabeto. Esto se muestra porque cada columna es de diferente color, es decir, el eje X es el que afecta en mayor medida a los colores.

Tiene sentido, puesto que la seguridad frente al ataque por tríos depende del número de primos disponibles entre $[2^b, 2^{b+\beta})$, que si recordamos la fórmula del Teorema de los Números Primos, sabemos que depende principalmente de $b = \log_2(z)$.

Sin embargo, n también tiene un pequeño efecto (vemos que el color es más claro para mayores valores de n) porque el valor de β se calcula en función de n para asegurar que haya suficientes primos disponibles.

Frente a estos resultados, tenemos que recordar la Tabla 2.1, que nos indica qué bits de seguridad son los estandarizados. Buscamos más de 128 bits de seguridad, para el legítimo uso de este criptosistema en el futuro. Para este límite mínimo, necesitamos un alfabeto de al menos 40 bits y un tamaño de mensaje de al menos 100 elementos.

El valor de t (o el peso del mensaje) no se tiene en cuenta porque no afecta a la cantidad de primos y, por tanto, no afecta a la seguridad frente al ataque por tríos.

6.1.3. Bits de seguridad para ataque por fuerza bruta

En la Figura 6.3 se muestra:

- En el eje X, el peso del mensaje ($w = t/n$). Cuanto más a la derecha, mayor es el peso.
- En el eje Y, el tamaño del mensaje (n). Cuanto más abajo, más grande es su valor.
- Los bits de seguridad frente al ataque por fuerza bruta se representa con los colores. Un color más claro (amarillo) indica más seguridad que un color oscuro (morado).

En este caso la seguridad frente al ataque por fuerza bruta depende principalmente del tamaño del mensaje n. Esto se comprueba observando que los colores dependen principalmente de la fila. Recordemos que este ataque depende del número de combinaciones de t

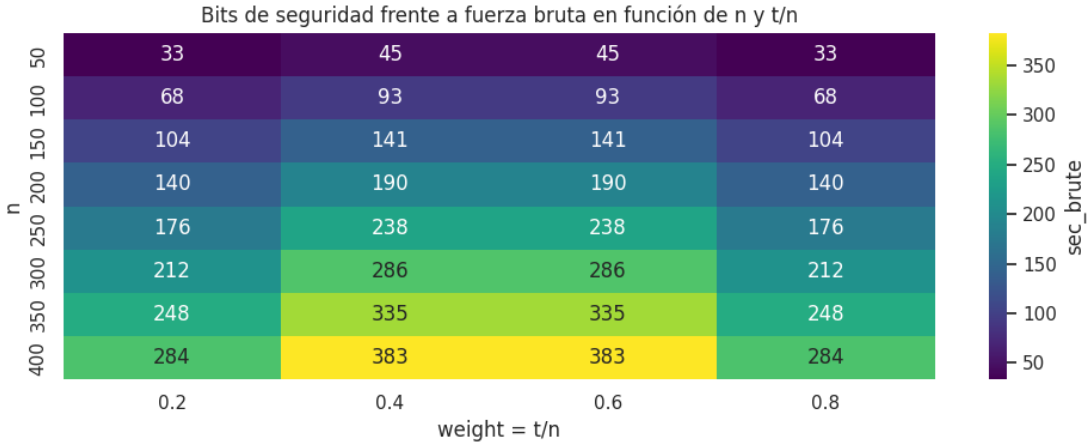


Figura 6.3.: Heatmap de bits de seguridad para ataque por fuerza bruta

elementos en un mensaje de tamaño n, que se calcula con el coeficiente binomial

$$\binom{n}{t}.$$

Vemos que valores pequeños de n tienen muy poca seguridad frente al ataque por fuerza bruta. Sin embargo, los bits de seguridad frente a este ataque crecen de forma exponencial con respecto al tamaño de n. Esto quiere decir que, para valores de $n = 150$, ya se obtiene unos bits de seguridad suficientemente buenos para algunos valores del peso.

El peso también afecta a la seguridad frente al ataque por fuerza bruta ya que, para un valor fijo de n, los coeficientes binomiales forman una parábola. Para valores extremos de t, el coeficiente binomial es menor; mientras que cuanto más cerca del cenrto se encuentre t, mayor coeficiente se obtiene. Esto se traduce como que la seguridad frente al ataque por tríos es más seguro cuanto más cerca se encuentre el peso de 0.5.

Por lo tanto, para valores de $n = 150$, podemos usar pesos centrales (entre 0.4 y 0.6) para ser suficientemente seguro según la tabla de NIST. Si nos vamos a mayores valores $n \geq 200$, la seguridad ya es suficiente para cualquier peso del mensaje.

En este caso, no se mide la seguridad de este ataque dependiendo del tamaño del alfabeto z, ya que no afecta en absoluto al coeficiente binomial.

6.2. Análisis de rendimiento

Con los mismos experimentos de la sección anterior, se han medido los tiempos de generación de claves, cifrado y descifrado de mensajes, y el tamaño de la clave pública.

Cada figura usada para analizar los tiempos de cada fase muestra cuatro gráficos correspondientes a cada posible peso del mensaje:

- En el eje X, el tamaño del alfabeto en bits ($\log_2(z)$). Cuanto más a la derecha, mayor es el alfabeto.

- En el eje Y, el tiempo de generación de claves. Cuanto más arriba, más tiempo se ha requerido.
- Cada color representa un valor de n. Cuanto más oscuro es el valor, más grande es el mensaje.

6.2.1. Tiempo de generación de claves

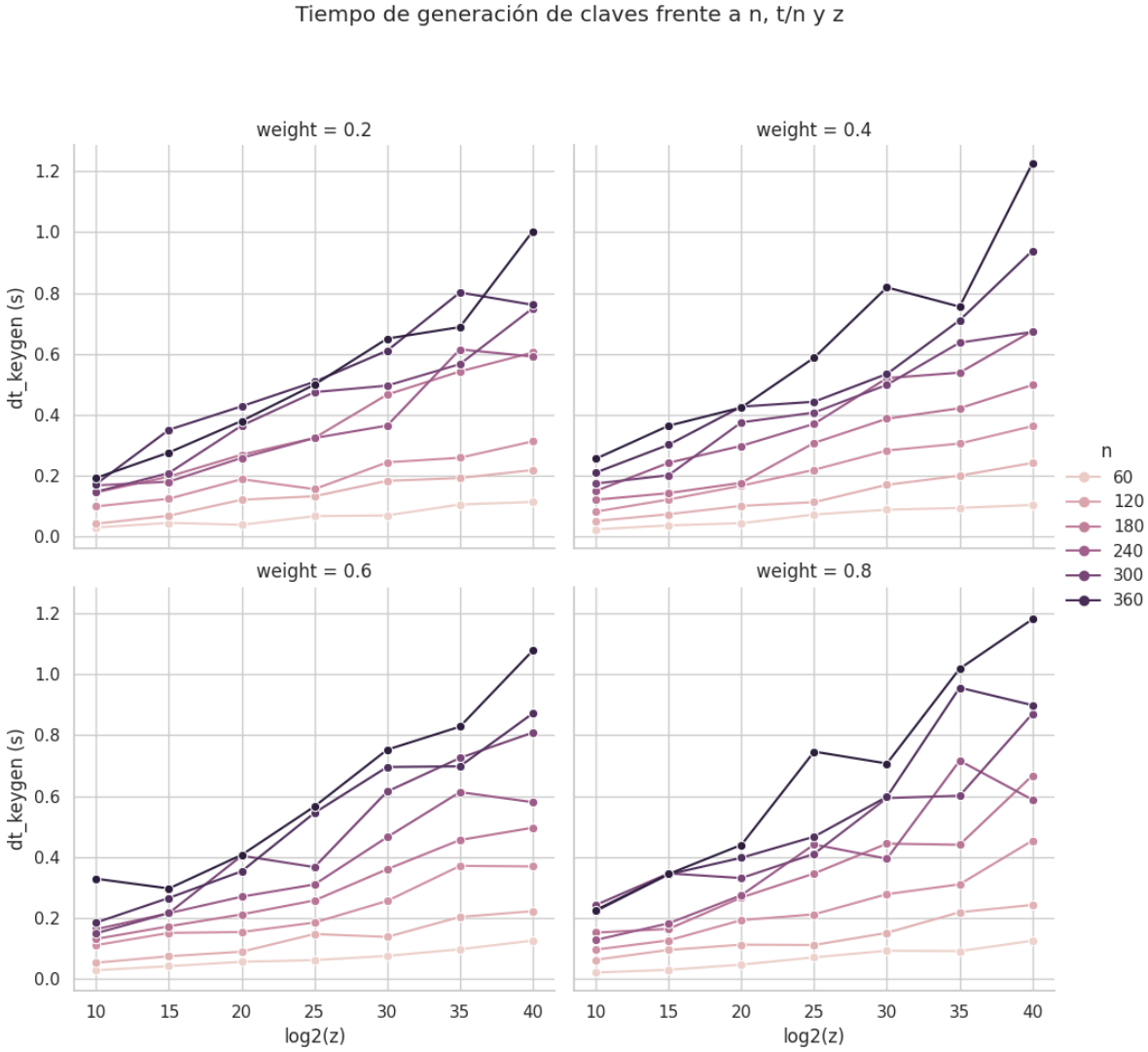


Figura 6.4.: Tiempo de generación de claves

En la Figura 6.4 se observa que los parámetros que más afectan a la generación de claves son el tamaño del alfabeto z y el tamaño del mensaje n. Esto se ve porque para cada punto de cada gráfico:

- Cuanto más a la derecha se encuentra el punto, tiende a estar más arriba: a mayor alfabeto, más tiempo requerido.
- Cuanto más oscuro es el punto, tiende a encontrarse más arriba: a mayor tamaño del mensaje, más tiempo requerido.
- Los cuatro gráficos no son demasiado diferentes entre ellos, lo que indica que el peso del mensaje no afecta en gran medida a la generación de claves.

Recordemos que la generación de claves incluye:

- Generar n primos aleatorios entre b y $b + \beta$ bits.
- Calcular n inversos modulares.
- Construir la lista pública t_i.

Por lo que el tiempo depende principalmente de n y de b, como se refleja en el gráfico. Además, podemos ver que los tiempos no son demasiado grandes para la gran mayoría de configuraciones, donde los valores no son extremos.

6.2.2. Tiempo de cifrado

En la Figura 6.5 se observa que los parámetros que más afectan al tiempo de cifrado son el tamaño del mensaje n y su peso. Esto se evidencia porque para cada punto de los gráficos:

- Cuanto más oscuro es el punto, tiende a encontrarse más arriba: a mayor tamaño del mensaje, más tiempo de cifrado.
- Cuanto mayor peso tiene el gráfico en el que se encuentra el punto, tiende a estar más arriba: a mayor peso, más tiempo de cifrado.
- Cada recta que forman los puntos no tiene demasiada inclinación: el tamaño del alfabeto no afecta demasiado al tiempo de cifrado.

Recordemos que durante el cifrado del mensaje solamente se deben generar dos valores:

$$c_1 = \sum_{i=1}^{n} e_i t_i, \quad c_2 = \sum_{i=1}^{n} e_i,$$

donde solamente t sumandos son no nulos (e_i es el mensaje con t posiciones no nulas). Por lo tanto, tiene sentido que dependa en mayor medida de la cantidad de sumandos (n), pero sobretodo en la cantidad de posiciones no nulas (el peso), tal y como indica el gráfico.

Sin embargo, esta fase no genera ningún tipo de problemas, ya que vemos que los tiempos son prácticamente inmediatos para cualquier configuración probada.

6.2.3. Tiempo de descifrado

La Figura 6.6 observamos que todos los parámetros afectan en gran medida al descifrado. Esto quiere decir que para cada punto de cada gráfico:

- Cuanto más a la derecha se encuentra, tiende a estar más arriba: a mayor alfabeto, más tiempo de descifrado.

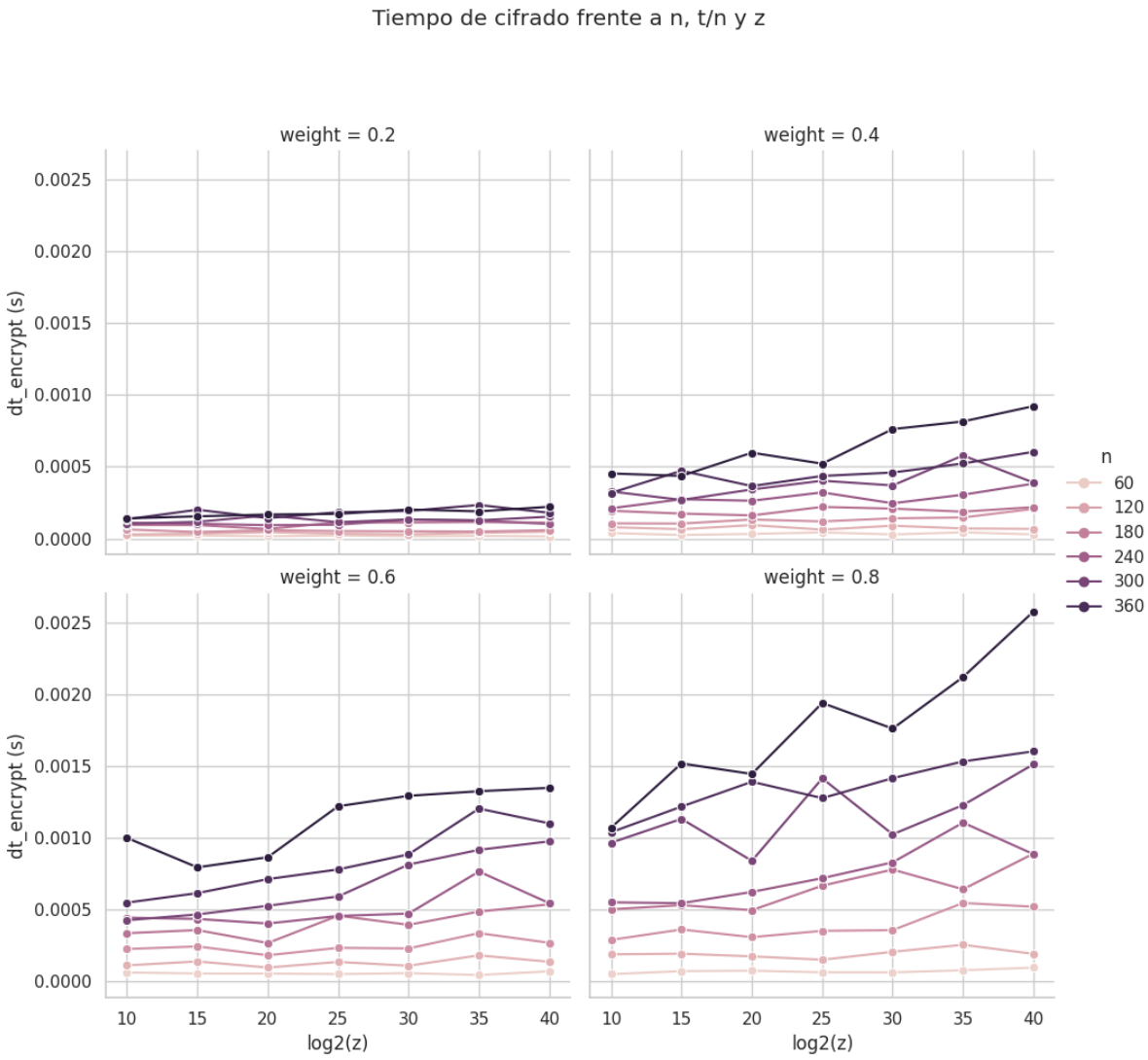


Figura 6.5.: Tiempo de cifrado

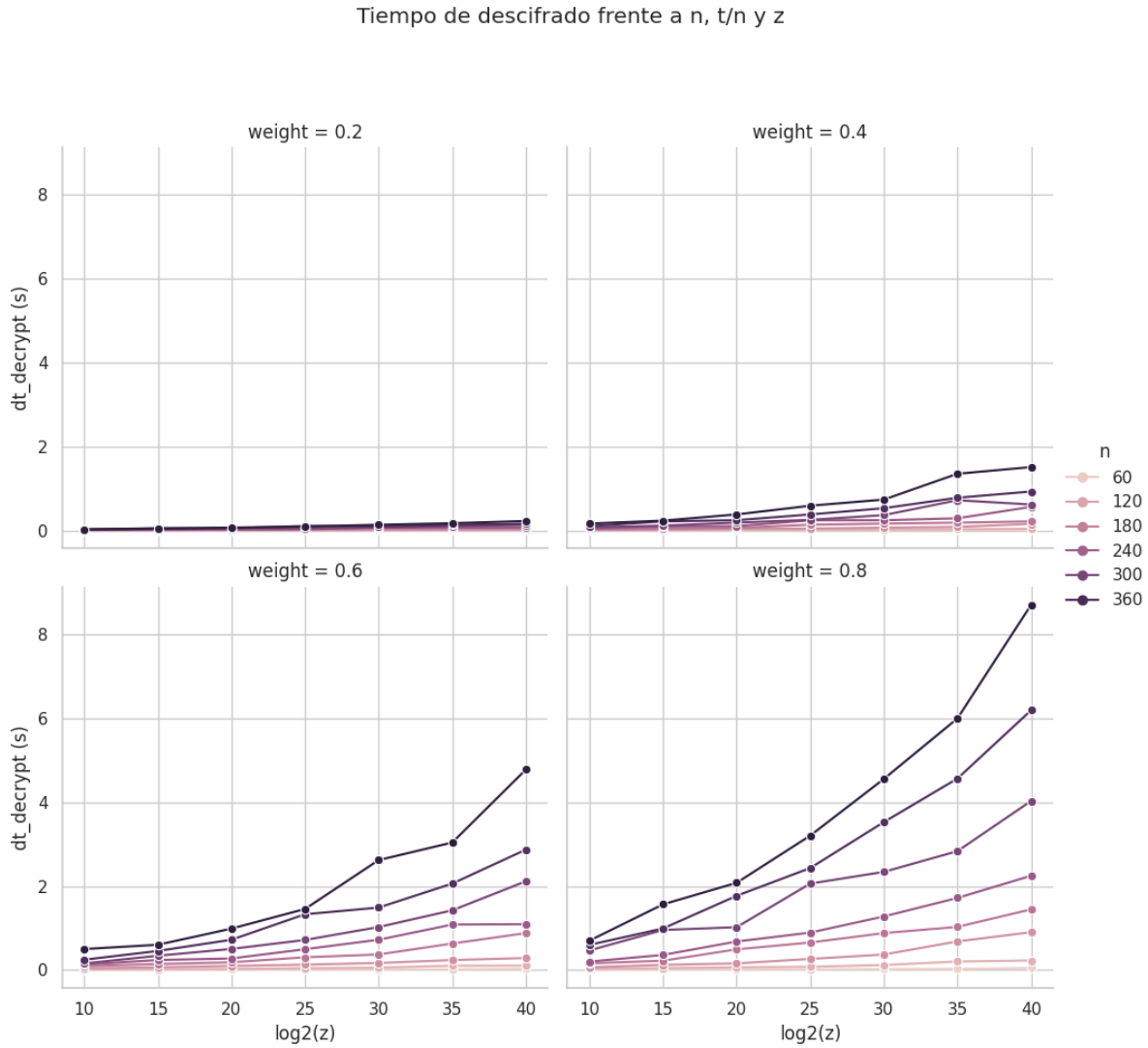


Figura 6.6.: Tiempo de descifrado

- Cuanto más oscuro es el punto, tiende a estar más arriba: a mayor tamaño del mensaje, más tiempo de descifrado.
- Cuanto más peso tiene el gráfico en el que se encuentra el punto, tiende a estar más arriba: a mayor peso, más tiempo de descifrado.

En esta fase se juega con:

- El tamaño de la clave privada para calcular cada $\gcd(p_i, g)$, donde la clave privada tiene $n + 2$ componentes. Por tanto, n afecta directamente a la cantidad de veces que se calcula el gcd.
- El tamaño de cada elemento de la clave privada también afecta al coste de cada operación. Lo que explica que z también afecta directamente al tiempo de descifrado.
- Una vez encontrados (λ, ω), se deben resolver t congruencias que nos dan cada componente no nula del mensaje original. Por lo tanto, el peso también afecta directamente al tiempo de descifrado.

Vemos que esta es la fase más costosa con diferencia. Por tanto, debemos mantener todos los valores de los parámetros lo más pequeños posibles, para evitar que el tiempo crezca de manera incontrolable.

6.2.4. Tamaño de la clave pública

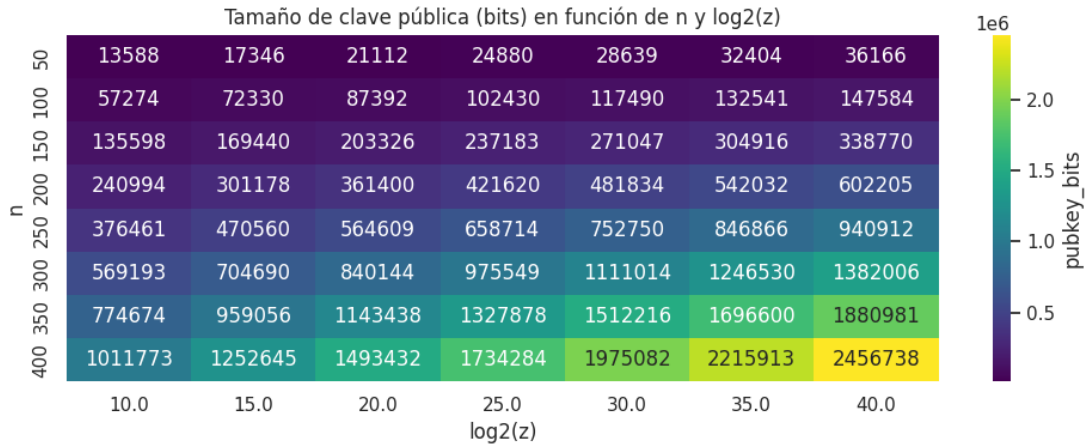


Figura 6.7.: Tamaño de la clave pública

En la Figura 6.7 se muestra:

- En el eje X, el tamaño del alfabeto en bits ($\log_2(z)$). Cuanto más a la derecha, mayor es el alfabeto.
- En el eje Y, el tamaño del mensaje (n). Cuanto más abajo, más grande es su valor.
- El tamaño de la clave pública en bits se representa con los colores. Un color más claro (amarillo) indica mayor tamaño que un color oscuro (morado).

Vemos que, en este caso, el tamaño de la clave pública depende tanto de n como de z. Esto se evidencia porque no hay una clara distinción de los colores ni por columnas ni por filas. De hecho, vemos que el color oscuro comienza arriba a la izquierda (con n pequeño y z pequeño) y se va aclarando conforme nos acercamos a la esquina inferior derecha (mayor n y mayor z).

Recordemos que la clave pública es un vector de n enteros, cuyo tamaño debe ser mayor que z. Por lo tanto, aumentar el tamaño de n aumenta el número de elementos en el vector y aumentar el tamaño de z, implica que el tamaño de cada uno de los elementos aumenta. Por lo tanto, el mapa de colores tiene mucha concordancia con la teoría.

En general, se busca tener el menor tamaño de clave pública posible, porque es ventajoso a la hora de almacenar la clave, transmitirla o realizar operaciones de cómputo con ella.

Por lo tanto, buscamos obtener tamaños de mensaje n y tamaños de alfabeto z lo más pequeños posibles, sin olvidarnos de la seguridad.

En este caso, no se mide el tamaño de la clave pública dependiendo del peso (o de t), porque apenas tiene efecto. Tiene efecto a la hora de calcular los bits del parámetro g, en la que afecta de forma similar que z. Por lo que podemos asegurar que para conseguir menores tamaños de la clave pública, buscamos menores tamaños de t (o del peso).

6.3. Análisis seguridad frente al ataque por baja densidad

En la sección anterior hemos visto una serie de configuraciones de parámetros que son seguras frente al ataque por tríos y al ataque por fuerza bruta. Sin embargo, no son los únicos ataques a tener en cuenta, sino que hay un ataque mucho más robusto frente a los esquemas basados en el SSP: el ataque por baja densidad.

Este tipo de ataque se ha explicado de forma detallada en la Subsección 4.2.1. Para experimentar frente a este tipo de ataques, se puede utilizar la librería Sage de Python [4]. Esta librería tiene la función `algorithm='NTL:LLL'`, que permite reducir los vectores de la base de un retículo dado usando los parámetros $\delta = 0.99$ y $\eta = 0.501$. Cabe recordar que para el ataque por reducción de retículos LLL, hay que transformar la instancia del SSP al SVP. Para hacerlo se ha probado con dos matrices diferentes: de una y de dos restricciones.

La matriz de una restricción utiliza

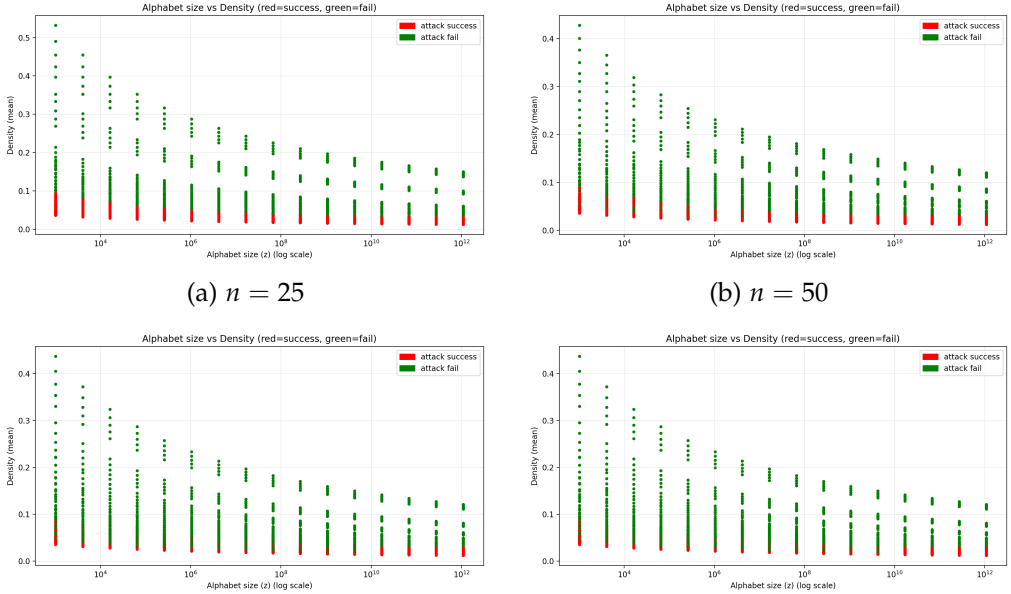
$$M = \begin{bmatrix} I_n & 0 \\ t & -c_1 \end{bmatrix},$$

donde I_n es la matriz identidad de tamaño n, t es el vector de la clave pública y c_1 es la primera componente del mensaje cifrado. La matriz de dos restricciones es

$$M = \begin{bmatrix} I_n & 0 \\ t & -c_1 \\ 1 & -c_2 \end{bmatrix},$$

donde c_2 es la segunda componente del mensaje cifrado.

Como se ha explicado en Párrafo 5.5.3, para atacar al esquema solamente se necesitan la clave pública y el texto cifrado, que es lo que vería el atacante del criptosistema. Sin embargo, para comprobar que el mensaje se ha descifrado correctamente, también se imprime el mensaje en texto plano.



(a) $n = 25$ (b) $n = 50$

(c) $n = 75$ (d) $n = 100$

Figura 6.8.: Ataques exitosos por d y z.

De esta forma, se han podido analizar los resultados obtenidos tras varios intentos con distintas configuraciones del criptosistema. Para atacarlo se han usado configuraciones computacionalmente asumibles para un ordenador convencional: $n = 25, 50, 75, 100$.

Además, para cada n, se han probado configuraciones con pesos $t = 0.1n, 0.15n, \cdots, 0.95n$ y $b = 10, \cdots, 40$. Para los valores de β, se han usado valores desde 1 hasta 5. Y para cada configuración, se han usado 10 semillas distintas.

De esta forma, queda con que por cada n se han probado $18 \times 16 \times 5 \times 10 = 14400$ configuraciones distintas.

Por cada intento de ataque se ha guardado el éxito del ataque (existoso o no), con qué método (una o dos restricciones) y su densidad. De estos resultados podemos obtener una gráfica que nos muestra la salida del ataque en función de los parámetros de la instancia y de su densidad.

6.3.1. Tamaño del alfabeto vs densidad

En la Figura 6.8 se ven cuatro subgráficas, una por dimensión: $n = 25$, $n = 50$, $n = 75$ y $n = 100$. En cada una:

- El eje X (logarítimico) representa el tamaño del alfabeto.
- El eje Y muestra la densidad de la instancia d.
- Un punto rojo indica que el ataque tiene éxito, mientras que un punto verde indica que el ataque ha fallado.

En las cuatro subgráficas se ve el mismo patrón:

- Los puntos rojos se concentran siempre en la franja de menor densidad (parte baja del eje Y).
- Para densidades más altas, solo hay puntos verdes, es decir, el ataque siempre falla.
- Además, incluso en densidad baja, los puntos rojos solamente aparecen para alfabetos pequeños (primeras columnas del eje X). A medida que el alfabeto crece, los puntos rojos cada veces aparecen menos.

Esto es exactamente lo que se espera a partir de la teoría: una densidad baja favorece que el vector solución sea geométricamente único, y LLL puede encontrarlo. Además, cuando z crece, todos los vectores tienen norma mucho más grande y encontrar el vector con norma pequeña se dificulta, de modo que ni siquiera con baja densidad se consigue ejecutar el ataque con éxito.

La banda donde hay éxitos es bastante fina. Para cada n, los éxitos se concentran en un rango de densidad muy reducido, aproximadamente por debajo de $d \approx 0.1$. Por encima de esa banda, incluso para alfabetos pequeños, el ataque ya no funciona.

Comparado con los umbrales clásicos de Lagarias-Odlyzko y Coster ($d < 0.64$ [16] y $d < 0.94$ [8]), estas instancias son mucho más resistentes. Esto se debe a que el esquema GLN no es un BSSP limpio, sino una versión modular ponderada, con un alfabeto no binario.

Comparando los cuatro paneles, para $n = 25$ hay algo más de puntos rojos que para $n = 100$. A medida que crece n, la franja donde el ataque tiene éxito tiende a encogerse y el número de éxitos disminuye. Es decir, aumentar n también ayuda contra el ataque, pero mucho menos que aumentar la densidad d o z.

6.3.2. Densidad en función del peso

En la Figura 6.9 se muestra claramente cómo varía la densidad del problema en función del peso relativo a la clave pública $w = t/n$.

Vemos que la densidad disminuye rápidamente al aumentar el peso w. Cuando w se encuentra entre 0.1 y 0.2, la densidad es relativamente alta (entre 0.15 y 0.5). Y a partir de $w \geq 0.4$, la densidad cae por debajo de 0.1 de manera consistente.

Esto es coherente: cuanto mayor es el peso t respecto al tamaño del vector, más grandes son los elementos de la mochila y, por tanto, más bits ocupan. Esto hace que el denominador $\log_2(g)$ crezca, reduciendo la densidad.

6.3.3. Ataques exitosos por peso de la instancia

La Figura 6.10 muestra, para cada par $(n, w = t/n)$, si el ataque por reducción de retículos consiguió recuperar el mensaje (rojo) en alguno de sus intentos o falló en todos los intentos (verde).

Existe una zona segura de solo puntos verdes cuando $w \leq 0.35$, esto quiere decir que el ataque no tiene éxito en ninguna configuración probada con esta condición. Es decir, para pesos pequeños el esquema es robusto frente a LLL en todo el barrido de z, β y semillas.

Encontramos una zona vulnerable de solo puntos rojos a partir de $w \geq 0.4$, donde aparecen únicamente puntos rojos para todos los n. Esto indica que para cada tupla (n, t), existe al menos una configuración en la que el ataque por retículos consigue romper el esquema.

En la Figura 6.9 anterior se veía que al aumentar el peso, la densidad disminuye. Lo que se comprueba claramente en este gráfico.

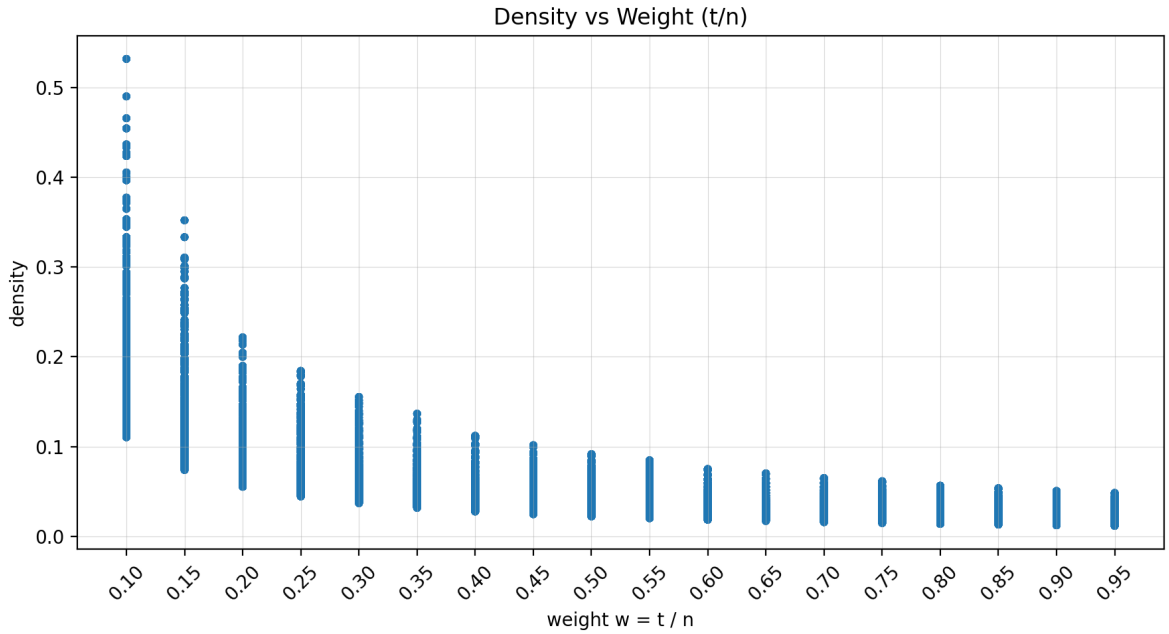


Figura 6.9.: Densidad en función del peso.

6.3.4. Éxito del ataque en función del alfabeto y el peso

La Figura 6.11 muestra, para cada par (b, w), si el ataque usando LLL tiene éxito o falla. La frontera es clarísima de nuevo: para todas las columnas de b, cuando $w \leq 0.35$ todos los ataques fallan y cuando $w \geq 0.4$ hay algún ataque que ha tenido éxito.

Por lo que hay una conclusión clara, el peso relativo manda totalmente. El ataque rompe el esquema cuando el peso del secreto es suficientemente grande independientemente del tamaño del alfabeto z. Esto coincide perfectametne con la teoría del SSP: el 80-90 % del comportamiento está determinado por la densidad.

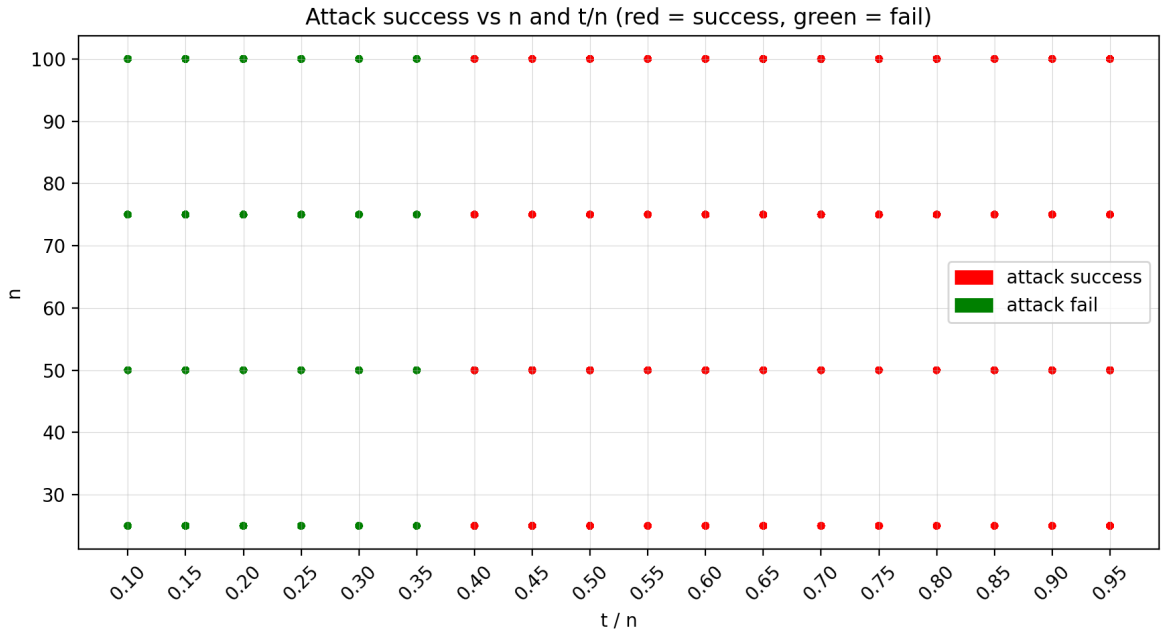


Figura 6.10.: Ataques exitosos por peso.

6.4. Conclusión de parámetros

El conjunto de experimentos realizados permite obtener una visión clara y completa del comportamiento del criptosistema GLN en términos de seguridad y eficiencia. Los análisis contemplan tres tipos de ataques (por tríos, por fuerza bruta y por baja densidad) y los parámetros relevantes del sistema (n,t,z). A partir de los resultados pueden obtenerse un conjunto de conclusiones sobre qué configuraciones son seguras y cuáles no son recomendables.

Seguridad frente al ataque por tríos Los resultados confirman que la seguridad combinatorio (bits de seguridad por tríos) depende casi exclusivamente del tamaño del alfabeto z. El parámetro n tiene un efecto secundario a través de β, y el peso t no influye en absoluto.

Para obtener más de 128 bits de seguridad, el mínimo recomendado por el NIST para cifrar datos a partir de 2031, se necesita:

- $\log_2(z) \geq 40$
- $n \geq 100$

Esto sitúa el esquema en la franje de seguridad adecuada frente a ataques combinatorios clásicos.

Seguridad frente al ataque por fuerza bruta El ataque por fuerza bruta consiste en enumerar todas las posibles combinaciones de t posiciones no nulas. Si coste es aproximadamente $\binom{n}{t}$.

Por tanto,

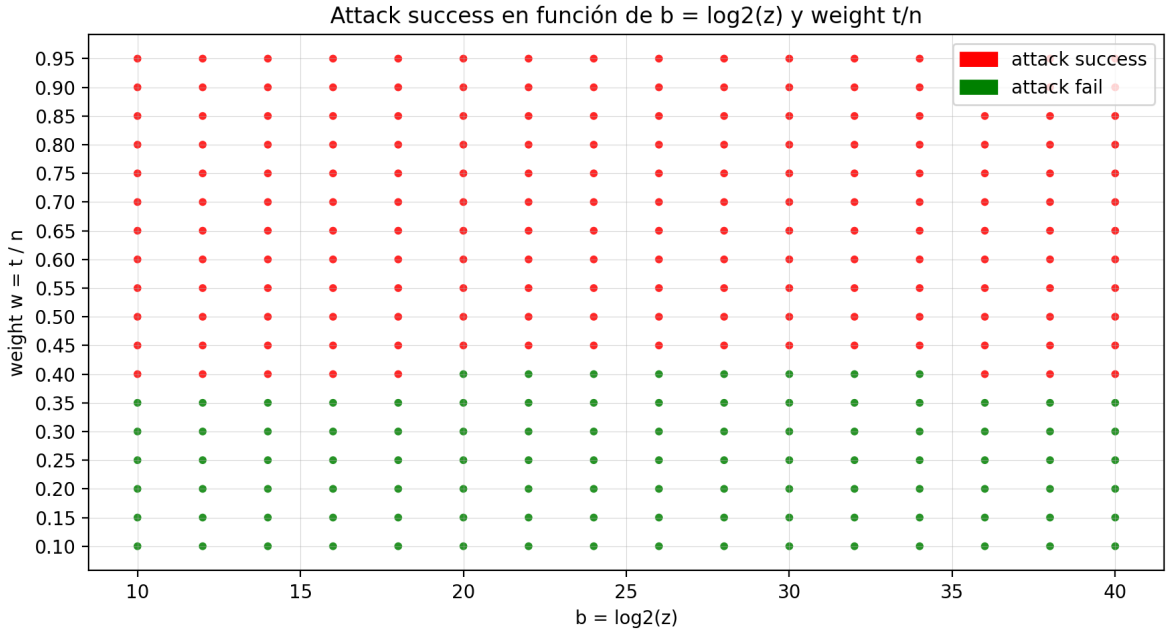


Figura 6.11.: Ataques exitosos en función de b y el peso.

- La seguridad crece con n de forma exponencial, mientras que z no influye.
- Para pesos centrales ($t \approx 0.4$-0.6), la seguridad es máxima.
- Para $n < 100$, la seguridad por fuerza bruta está muy por debajo de los 112 bits, y el sistema debe considerarse inseguro.
- A partir de $n \approx 150$, incluso con pesos moderados, se obtiene más de 128 bits, cumpliendo los estándares modernos.

En consecuencia, los parámetros pequeños aunque eficientes computacionalmente, no deben usaese en escenarios reales.

Seguridad frente al ataque por reducción de retículos El ataque LLL es mucho más sutil y está guiado por la densidad:

$$d = \frac{n}{\log_2(g)}.$$

Los experimentos muestran un patrón claro:

- Para pesos $w \leq 0.35$, el ataque falla sistemáticamente en todos los casos probados.
- Para pesos $w \geq 0.4$, aparece algún ataque exitoso.

Esto confirma la teoría del SSP: la densidad (y por tanto el peso) es el factor decisivo. Para garantizar seguridad frente a LLL es necesario imponer

$$\frac{t}{n} \leq 0.35.$$

Impacto en el rendimiento Los tiempos experimentales permiten evaluar la viabilidad práctica del esquema:

- La generación de claves crece con n y con $\log_2(z)$, pero sigue siendo razonable para todos los parámetros probados.
- El tiempo de cifrado es extremadamente rápido y apenas depende de z; depende sobretodo del peso t.
- El tiempo de descifrado es crítico: crece rápidamente con n, t y z.

Tamaño de la clave pública El tamaño de la clave pública depende de n y de $b = \log_2(z)$. El crecimiento es muy pronunciado para parámetros grandes, y esta es la principal limitación práctica del esquema.

Valores altos de seguridad combinatoria implican claves públicas muy grandes (de cientos de kilobits).

6.4.1. Análisis final

Integrando los tres ataques y el coste computacional, las configuraciones recomendables deben cumplir simultáneamente:

- Seguridad por tríos: $b = \log_2(z) \geq 40$ y $n \geq 100$.
- Seguridad por fuerza bruta: $n \geq 200$ o $n \geq 150 \wedge t \approx 0.5n$.
- Seguridad frente a LLL: $t \leq 0.35$.
- Eficiencia razonable: $n \leq 200 \wedge b \leq 40$.

Un rango equilibrado y razonable para el uso práctico de GLN sería:

$$\begin{gathered} 150 \leq n \leq 200 \\ 0.25 \leq w \leq 0.35 \\ \log_2(z) \approx 40 \end{gathered}$$

Estas configuraciones satisfacen simultáneamente todos los requisitos para un criptosistema seguro frente a los tres ataques vistos y tiempos manejables en hardware convencional.

7. Conclusión

El principal objetivo de este Trabajo de Fin de Grado era evaluar la viabilidad práctica y la seguridad del criptosistema GLN, un esquema de clave pública basado en variantes del SSP y candidato a ofrecer seguridad frente a adversarios cuánticos.

El trabajo comienza estableciendo los fundamentos necesarios de teoría de la computación y complejidad algorítmica. Este enfoque permite entender por qué problemas como la suma de subconjuntos o el Shortest Vector Problem en retículos constituyen la base matemática de muchos criptosistemas postcuánticos. Asimismo, se revisa la evolución de la criptografía de clave pública, los límites de los esquemas clásicos ante la posible computación cuántica y la motivación del diseño de nuevas familias criptográficas resistentes tanto a Shor como a Grover.

En este contexto se presenta el criptosistema GLN, cuyo diseño combina ideas de esquemas de tipo mochila y técnicas algebraicas inspiradas en el sistema de McEliece. El análisis teórico del esquema incluye la demostración de su corrección, así como el estudio de los parámetros necesarios para garantizar que el descifrado produce un mensaje válido y único. Esta revisión cubre así el primer objetivo del proyecto: comprender en profundidad los fundamentos matemáticos necesarios para estudiar el GLN.

A continuación, se ha llevado a cabo la implementación completa del criptosistema en C++, cumpliendo el segundo objetivo del TFG. Esta implementación ha requerido desarrollar una aritmética de enteros grandes adaptada al tamaño de los parámetros, así como algoritmos auxiliares como Miller–Rabin, operaciones modulares eficientes, generación controlada de primos y de parámetros. Los módulos de generación de claves, cifrado, descifrado y ataques se han diseñado de forma independiente, teniendo en cuenta la claridad, la extensibilidad y la reproducibilidad de los experimentos.

El tercer y cuarto objetivo se han alcanzado mediante un procedimiento experimental automatizado con Python. Los resultados permiten analizar:

- La influencia de los parámetros estructurales (n, t, z) en los tiempos de ejecución y en el tamaño de la clave pública.
- La seguridad efectiva frente a ataques combinatorios (ataque por tríos y fuerza bruta).
- La resistencia frente al ataque de baja densidad basado en LLL, uno de los ataques más poderosos contra esquemas tipo mochila.

Los experimentos confirman las predicciones teóricas. Frente al ataque por tríos, la seguridad crece fundamentalmente con el tamaño del alfabeto z y, en menor medida, con n. Frente a la fuerza bruta, la seguridad depende casi exclusivamente de n y del peso relativo t/n, mostrando el crecimiento exponencial esperado del coeficiente binomial. Por su parte, los ataques basados en retículos muestran que la densidad es el parámetro crítico: pesos pequeños (aproximadamente $t/n \leq 0.35$) producen densidades suficientemente altas como para resistir de manera consistente instancias atacadas muchas veces.

Al combinar estos tres análisis (tríos, fuerza bruta y LLL) se obtiene una caracterización clara del espacio de parámetros seguros. Existen rangos amplios donde el GLN alcanza más

de 128 bits de seguridad clásica, requisito mínimo del NIST para algoritmos válidos más allá de 2031, y además es resistente a ataques cuánticos conocidos al basarse en un problema NP-duro sin algoritmos cuánticos eficientes conocidos. Curiosamente, los resultados evidencian que el principal obstáculo práctico no es la seguridad, sino el tamaño de la clave pública, que crece linealmente con n y con $\log_2(z)$ y puede alcanzar valores de cientos de kilobits.

Este aspecto sitúa al GLN ante un dilema común en la criptografía postcuántica: mejorar la seguridad implica aumentar la longitud de las claves y los tiempos de descifrado. Este comportamiento no es exclusivo del GLN; de hecho, criptosistemas estandarizados como McEliece presentan claves públicas de tamaño incluso mayor. El GLN es una propuesta adecuada entre los esquemas postcuánticos basados en problemas combinatorios y de retículos, donde el coste de almacenamiento es el precio a pagar por resistir ataques cuánticos estructurales.

A pesar de esta limitación, los experimentos muestran que el GLN puede operar de forma eficiente para tamaños moderados de n (en torno a 150–200 valores), ofreciendo seguridad combinatoria elevada y resistiendo LLL siempre que el peso sea lo suficientemente pequeño. Además, el coste de cifrado es extremadamente bajo y el descifrado, aunque el componente más costoso, se mantiene dentro de rangos razonables para configuraciones seguras.

En conjunto, los resultados del TFG permiten concluir que:

- El criptosistema GLN es un esquema teóricamente sólido, cuya construcción se apoya en problemas duros tanto en escenarios clásicos como postcuánticos.
- Es posible implementarlo de forma modular, eficiente y reproducible.
- Su resistencia empírica frente a ataques conocidos es elevada, permitiendo identificar zonas claras de parámetros seguros.
- Su principal limitación práctica es el tamaño de la clave pública, una característica compartida con otros esquemas postcuánticos basados en combinatoria.
- El GLN constituye una alternativa viable en el contexto postcuántico, especialmente en escenarios donde el tamaño de clave no es crítico y donde se requiere simplicidad algebraica y ausencia de estructuras lineales que puedan explotarse con algoritmos cuánticos.

A. Mecanismo de encapsulación de claves (KEM)

,

Como hemos explicado en el Capítulo 3, GLN es un sistema de cifrado de clave pública (PKE). Esto significa que, en principio, podría usarse para cifrar directamente cualquier mensaje m. Sin embargo, en la práctica esto sería ineficaz y poco seguro. Por un lado, los cifrados de clave pública son muy lentos y pesados para mensajes largos; por otro lado, los PKE suelen ser seguros solo frente a ataques Chosen Plaintext Attack (CPA), pero no frente a ataques más fuertes como Chosen Ciphertext Attack (CCA2), que son el estándar real de seguridad.

La solución al problema es usar un mecanismo de encapsulación de claves (KEM), que separa el problema en dos capas:

- Encapsular una clave corta con GLN (KEM - Key Encapsulation Mechanism): el emisor usa GLN para cifrar un valor pequeño aleatorio e. Este valor, combinado con el mensaje cifrado, se pasa por una función hash para generar una clave de sesión k.
- Usar esa clave de sesión con un cifrado simétrico (DEM - Data Encapsulation Mechanism): con k, que ahora se usa como clave simétrica, se cifra el mensaje real usando un algoritmo rápido y probado (AES, por ejemplo).

Así, obtenemos el esquema estándar KEM-DEM. Para construir un KEM a partir de un PKE, como el introducido en el apartado anterior, se definen tres algoritmos: `KeyGen'`, `Encap` y `Decap`, y dos conjuntos finitos no vacíos: `SKeys` y `CipherKeys`. El primero de ellos contiene las posibles claves de sesión que produce el KEM tras aplicar la función hash, mientras que `CipherKeys` contiene las posibles claves simétricas que se derivan tras encapsular.

En el PKE de GLN teníamos un conjunto de mensajes en claro, `Plaintexts`, formado por todos los vectores $e = (e_1, \ldots, e_n)$ con $0 < e_j < z$. Para construir el KEM a partir del PKE, estos mensajes en claro no se usan como mensajes normales, sino como semillas para generar claves. Es decir, los mensajes que antes eran `Plaintexts` ahora son tratados como posibles `CipherKeys` (valores intermedios que, tras aplicar un hash, darán lugar a las verdaderas claves de sesión `SKeys`). Es por ello que:

$$Plaintexts \subseteq CipherKeys.$$

Ejemplo A.1. Supongamos que Alice quiere mandarle un mensaje m a Bob. Es decir, lo que busca es acordar con Bob una clave simétrica corta k que servirá para cifrar el mensaje m de forma segura, usando un algoritmo de clave privada.

Alice elige un elemento $e \in Plaintexts$ y calcula el criptograma $(c_0, c_1) = Encrypt(e)$, que será lo que envíe a Bob. A continuación, aplica una función hash

$$k = H(1, e, (c_0, c_1)) \in SKeys.$$

Ahora Alice puede utilizar k para cifrar el mensaje real m usando un algoritmo simétrico eficiente y seguro, como AES

$$C_{sym} = Enc_k^{AES}(m).$$

Tenemos entonces que Alice envía a Bob: el criptograma corto (c_0, c_1) (donde se encapsula la clave), y el criptograma simétrico C_{sym} (el mensaje cifrado de verdad).

Al recibirlo, Bob ejecuta `Decap` con su clave privada y recupera la misma clave k. Con esta clave descifra el mensaje simétrico

$$m = Dec_l^{AES}(C_{sym}).$$

■

El KEM derivado se llama `SimpleKEM(Π)` y se describe como una tupla

$$(HashInputs, PubKeys, PrivKeys', SKeys, Ciphertexts, KeyGen', Encap, Decap).$$

El espacio de entrada de la función hash será $HashInputs = \{0,1\} \times CipherKeys \times CipherTexts$, donde el primer bit indica si la decapsulación fue existosa o fallida.

La clave privada se amplía: ya no es solo $(p_1, \ldots, p_m, g, u)$, sino que incluye también la clave pública y un valor aleatorio $r \in CipherKeys$ que se usará en caso de error de decapsulación. Formalmente,

$$PrivKeys' = PrivKeys \times PubKeys \times CipherKeys$$

El algoritmo `KeyGen'` trata de ejecutar el `KeyGen` del PKE y obtener la clave pública $(t_1, \ldots, t_n)$ y la clave privada $(p_1, \ldots, p_n, g, u)$. Además, se elige al azar un $r \in CipherKeys$. El resultado será:

- Clave pública: $(t_1, \ldots, t_n)$.
- Clave privada extendida: $(p_1, \ldots, p_n, g, u, t_1, \ldots, t_n, r)$.

Para la encapsulación (`Encap`) se utiliza como entrada la clave pública y una función hash:

$$H : HashInputs \to SKeys.$$

A continuación, se elige aleatoriamente un $e \in PlainTexts$ y se cifra con el PKE

$$(c_0, c_1) = Encrypt(e).$$

La salida es un par

$$((c_0, c_1), H(1, e, (c_0, c_1))).$$

Solo el cifrado (c_0, c_1) se transmite y la clave simétrica encapsulada es la salida H.

Para la decapsulación (`Decap`) se utiliza como entrada la función hash H, el cifrado (c_0, c_1) y la clave privada extendida, y se descifra $e = Decrypt((c_0, c_1))$. Si $e \in PlainTexts$ y al volver al cifrarlo se recupera (c_0, c_1), entonces la decapsulación es correcta y la salida es:

$$H(1, e, (c_0, c_1)).$$

En caso contrario, ha habido un fallo de descifrado y se devuelve

$$H(0, r, (c_0, c_1)).$$

Glosario

$\mathbb{R}$ Conjunto de números reales.

$\mathbb{C}$ Conjunto de números complejos.

$\mathbb{Z}$ Conjunto de números enteros.

NP Tiempo polinómico no determinista.

SAT Satisfiability Problem.

SSP Subset Sum Problem.

AES Advanced Encryption Standard.

RSA Rivest-Shamir-Adleman.

ECC Elliptic Curve Cryptography.

PQC Criptografía post-cuántica.

NTRU Number Theorists 'R' Us.

PKE Public Key Encryption.

KEM Key Encapsulation Mechanism.

DEM Data Encapsulation Mechanism.

EEA Extended Euclidean Algorithm.

TrEEA Truncated Extended Euclidean Algorithm.

GLN José Gómez-Torrecillas, F. J. Lobillo, Gabriel Navarro

WMSSP Weighted Modular Subset Sum Problem.

MSSP Modular Subset Sum Problem.

BSSP Binary Subset Sum Problem.

NIST National Institute of Standards and Technology.

LLL Lenstra-Lenstra-Lóvasz.

SVP Shortests Vector Problem.

GCD Greatest COmmon Divisor.

OOP Object-Oriented Programming.

UML Unified Modeling Language.

Plmpl Pointer to Implementation.

RNG Random Number Generator.

API Application Programming Interface.

CPU Central Processing Unit.

RAM Random Access Memory.

CSV Comma-Separated Values.

JSON JavaScript Object Notation.

GMP GNU Multiple Precision Arithmetic Library.

LTS Long Term Support.

Bibliografía

[1] C. 101. V1: Introduction to lattices (lattice basis reduction), 09 2025.

[2] M. Albrecht, A. Davidson, A. Deo, B. Curtis, D. Stehlé, E. Postlethwaite, E. Kirshanova, F. Virdia, F. Göpfert, G. Herold, J. Schanck, L. Ducas, M. Stevens, P. Kirchner, P.-A. Fouque, R. Player, S. Scott, S. Bai, T. Wunderer, V. Gheorghiu, and W. Wen. Algorithms for lattice problems: In practice lattice boot camp @ simons based on joint work with, 2020.

[3] M. Albrecht and L. Ducas. Sage for lattice-based cryptography, 11 2024.

[4] M. R. Albrecht and L. Ducas. Sage for lattice-based cryptography. Oxford Lattice School lecture notes. Apuntes del Oxford Lattice School.

[5] N. . E. Barker. Recommendation for key management, part 1: General. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf, 2016. Sitio oficial del NIST, actualizado a mayo de 2020.

[6] D. J. Bernstein, J. Buchmann, and E. Dahmen. *Post-Quantum Cryptography*. Springer, Dordrecht, 2008.

[7] C++. Plmpl. https://en.cppreference.com/w/cpp/language/pimpl.html. Sitio oficial de C++.

[8] M. Coster, A. Joux, B. Lamacchia, A. Odlyzko, C. Schnorr, and J. Stern. Improved low-density subset sum algorithms. volume 2, 04 1999.

[9] P. Crescenzi and V. Kann. The np-completeness of subset sum. Lecture notes, University of Florence and KTH, Oct. 2011. October 2011.

[10] N. . CSRC. Post-quantum cryptography: Selected algorithms. https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms, 2025. Sitio oficial del NIST, actualizado a septiembre de 2025.

[11] M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, complexity, and languages (2nd ed.): fundamentals of theoretical computer science*. Academic Press Professional, Inc., USA, 1994.

[12] S. Galbraith. Mathematics of public key cryptography. version 2.0, 10 2018.

[13] J. Gómez-Torrecillas, F. J. Lobillo, and G. Navarro. A knapsack mceliece-based public key cryptosystem. Manuscrito, Department of Algebra, University of Granada, 2024.

[14] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer, New York, NY, 2 edition, 2014.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. Pearson Educación S.A., Madrid, 2007.

[16] J. C. Lagarias and A. M. Odlyzko. Solving low density subset sum problems. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 1–10, 1983.

[17] L. A. L. L. Lenstra, H.W. jr. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[18] J. Liu, J. Bi, and S. Xu. An improved attack on the basic merkle–hellman knapsack cryptosystem. *IEEE Access*, 7:59388–59393, 2019.

[19] B. Lynn. Primality tests. https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html. Sitio oficial de Standford.

[20] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5):525–530, 1978.

[21] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[22] Y. Murakami and R. Sakai. Security of knapsack cryptosystem using subset-sum decision problem against alternative-solution attack. In *2018 International Symposium on Information Theory and Its Applications (ISITA)*, pages 614–617, 2018.

[23] National Institute of Standards and Technology. Advanced encryption standard (aes). Federal Information Processing Standards Publication 197-upd1, 2001. Updated May 9, 2023.

[24] A. M. Odlyzko. The rise and fall of knapsack cryptosystems. 1998.

[25] C. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 08 1994.

[26] X. Schoefield and É. Tardos. Subset sum is np-complete. Lecture notes for CS 4820: Introduction to Algorithms, Cornell University, Oct. 2018. Lecture notes, October 29, 2018.

[27] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

[28] M. Stamp. Lattice reduction attack on the knapsack.

[29] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

[30] J. van der Pol. The bkz algorithm. *Cryptology ePrint Archive*, 2009. Report 2009/009.

[31] J. von zur Gathen. *CryptoSchool*. Springer, Berlin, Heidelberg, 1 edition, 2015.

[32] E. W. Weisstein. Prime number theorem. From MathWorld – A Wolfram Resource. Consultado en 2025.