Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Matemáticas

Criptografía de curva elíptica: Teoría, Seguridad e Implementación

Autor: Francisco Cardeñoso Perez
Tutor: Jose Enrique Marcos Naveira
Año: 2025

Resumen

El objetivo de este trabajo es estudiar la criptografía de curva elíptica (ECC) desde tres enfoques complementarios: teórico, criptográfico y práctico. En el plano teórico, se presentan los fundamentos de la criptografía y se introducen las curvas elípticas junto con sus propiedades algebraicas esenciales. Se presta especial atención a su comportamiento sobre cuerpos finitos, incluyendo el endomorfismo de Frobenius, el teorema de Hasse y el problema del logaritmo discreto en curvas elípticas (ECDLP), base de su seguridad. Desde el punto de vista criptográfico, se analizan los principales algoritmos conocidos para resolver el ECDLP, evaluando su complejidad y efectividad. Finalmente, en el nivel práctico, se implementa un sistema completo de intercambio de claves utilizando ECC. A través de pruebas con claves de tamaño reducido, se ilustra empíricamente el crecimiento exponencial de la complejidad asociado al tamaño de la clave, lo que evidencia la solidez de la curva elíptica como herramienta criptográfica.

Palabras clave

Criptografía, curvas elípticas, ECC, ECDLP, cuerpos finitos, endomorfismo de Frobenius, teorema de Hasse, algoritmos subexponenciales, intercambio de claves, firma electrónica, complejidad computacional, SageMath

Abstract

This work explores elliptic curve cryptography (ECC) from three interconnected angles: theory, security, and implementation. On the theoretical side, it introduces the foundations of cryptography and develops the algebraic structure of elliptic curves, with a focus on their behavior over finite fields. Key concepts such as the Frobenius endomorphism, Hasse's theorem, and the elliptic curve discrete logarithm problem (ECDLP) are examined in depth. The security section analyzes the most effective known attacks on the ECDLP, which forms the basis of ECC's cryptographic strength. Finally, the practical component involves implementing a complete key exchange system using ECC. By testing small key sizes, the exponential growth in complexity as key length increases is demonstrated, highlighting the robustness of elliptic curves in secure communication protocols.

Keywords

Cryptography, elliptic curves, ECC, ECDLP, finite fields, Frobenius endomorphism, Hasse's theorem, subexponential algorithms, key exchange, digital signature, computational complexity, SageMath

Agradecimientos

En primer lugar agradecer a Jose Enrique Marcos Naveira por su paciencia y disponibilidad cada vez que le entregaba versiones incompletas del trabajo. Siempre se tomaba el tiempo de imprimirlas y revisarlas. Sus comentarios, consejos y contribuciones han contribuido enormemente a la forma final del escrito.

También agradecer al que fue mi tutor de empresa durante las prácticas Alejandro Alfonso Fernandez, que me introdujo al mundo de la firma electrónica y los servicios de confianza. Durante las prácticas aprendí enormemente sobre la implementación en software de muchos servicios y las dificultades que aparecen al tratar de implementar cualquier solución técnicamente.

Objetivos

En un principio este trabajo comienza dentro de un marco de prácticas de empresa en DigitelTS. Esta empresa es un proveedor de servicios de confianza que elabora soluciones de firma electrónica. Este contexto promueve que los objetivos del trabajo sean tanto académicos como profesionales. Se ha tratado de balancear la importancia dada tanto al rigor matemático en la exposición de las estructuras tratadas como la implementación práctica de estos objetos y estructuras en el ámbito profesional.

De este modo la estructura propuesta del trabajo adquiere naturalmente un comienzo más teórico centrado en la introducción de conceptos e ideas y su definición explícita, seguido de un progresivo acercamiento a las aplicaciones que es visible desde los propios títulos de los capítulos. El primer capítulo se denomina introducción a la criptografía, que describe los principios fundamentales de la criptografía. Le sigue el capítulo más extenso y formal del trabajo que trata sobre la curvas elípticas. En él se introducen definiciones y propiedades asociadas a ellas. El tercer capítulo se acerca ya mucho a la parte más aplicada y trata sobre los protocolos basados en curvas elípticas. Se describen los más importantes y usados en el mundo actual. El capítulo cuatro, es uno más centrado en la seguridad de la criptografía con curvas elípticas, ECC en adelante. Se describen los dos mejores algorítmos actuales que resuelven el problema en el que se basa la seguridad de la ECC, el problema del logarítmo discreto para curvas elípticas (ECDLP). Finalmente se implementan todos los conceptos descritos a nivel teórico mediante el framework más común utilizado en criptografía a nivel académico SageMath. Para más información se incluye un apéndice con una introducción a la teoría de la complejidad, que es la herramienta de la que disponen las matemáticas para estimar el coste que implica ejecutar un algorítmo.

El primer capítulo tiene como objetivo introducir al lector en los fundamentos teóricos de la criptografía, abordando tanto sus principios como las diferencias fundamentales entre la criptografía simétrica y la asimétrica. Este capítulo sirve como base conceptual para comprender los capítulos posteriores, donde se profundiza en herramientas criptográficas más avanzadas.

El segundo capítulo está dedicado íntegramente al estudio de las curvas elípticas. Su objetivo es presentar de manera rigurosa la definición matemática de estas curvas, así como sus propiedades algebraicas más relevantes. Se abordan conceptos como el espacio proyectivo, las reglas de adición de puntos, el invariante j, los puntos de torsión, y su comportamiento sobre cuerpos finitos. Este capítulo constituye el núcleo teórico del trabajo, y sienta las bases necesarias para comprender tanto los protocolos como los ataques que se analizan más adelante.

En el tercer capítulo se estudian algunos de los protocolos criptográficos más importantes que se basan en curvas elípticas. El objetivo de este capítulo es mostrar cómo las propiedades descritas en el capítulo anterior permiten construir sistemas criptográficos concretos, como ECDSA para la firma digital y ECDH para el intercambio de claves. También se revisan las curvas más comunes utilizadas en implementaciones reales.

El cuarto capítulo tiene como objetivo analizar los algoritmos conocidos para resolver el problema del logaritmo discreto en curvas elípticas, base de la seguridad de la criptografía de curva elíptica. Se presentan los algoritmos Baby-Step Giant-Step y Pollard's Rho, describiendo tanto sus fundamentos teóricos como sus implicaciones prácticas. Este capítulo aporta una perspectiva crítica sobre las limitaciones y fortalezas de la ECC.

El quinto capítulo aborda la implementación práctica de los conceptos desarrollados a lo largo del trabajo. Se realiza un análisis detallado del proceso de generación de curvas, la implementación de protocolos como ECDH, y se construyen experimentos concretos para resolver instancias del ECDLP utilizando herramientas computacionales como SageMath. El objetivo de este capítulo es establecer un puente entre la teoría matemática y su aplicación real en un entorno profesional.

Finalmente, el apéndice sobre teoría de la complejidad tiene como objetivo proporcionar al lector los fundamentos necesarios para entender cómo se estima el coste computacional de un algoritmo. Se explican nociones básicas sobre clases de complejidad, tiempos de ejecución, y se discute brevemente el problema de la factibilidad algorítmica en el contexto de la criptografía.

Contenidos

Capítulo 1

Introducción a la criptografía

1.1. Introducción histórica

La criptografía, el arte de proteger la información mediante técnicas de ocultamiento, tiene sus raíces en la antigüedad. Ya en tiempos de los romanos, se empleaban sistemas simples de cifrado para proteger mensajes militares o políticos. Uno de los métodos más conocidos es el cifrado de César, una técnica de sustitución monoalfabética en la que cada letra del mensaje original se reemplaza por otra situada tres posiciones más adelante en el alfabeto. Matemáticamente, este cifrado puede expresarse mediante la fórmula $C = P + 3 \mod 26$, donde P representa la posición de la letra en el alfabeto y C la letra cifrada resultante[19]. A pesar de su simplicidad, este sistema cumple la función básica de ocultar la información, pero es fácilmente quebrantable mediante análisis de frecuencia, ya que cada letra del texto plano se transforma siempre en la misma letra del texto cifrado.

En el siglo XVI, el criptógrafo francés Blaise de Vigenère propuso una mejora significativa a este esquema mediante el uso de una palabra clave. En lugar de aplicar un único desplazamiento a todo el mensaje, Vigenère propuso agrupar el texto en bloques de k letras y aplicar a cada bloque una traslación diferente, definida por los caracteres sucesivos de la palabra clave. Esta técnica, que puede interpretarse como una suma vectorial en $(\mathbb{Z}/26\mathbb{Z})^k$, aumenta la complejidad del cifrado y reduce la efectividad del análisis de frecuencia. No obstante, si se conoce la longitud del período o si se dispone de un volumen suficientemente grande de texto cifrado, este método también puede ser vulnerado, especialmente si se analiza la frecuencia de aparición de letras en posiciones con la misma separación.

Un paso más allá se dio en 1931, cuando Lester Hill propuso un cifrado basado en álgebra lineal. En su sistema, cada bloque de k letras del texto plano es representado como un vector en $(\mathbb{Z}/N\mathbb{Z})^k$, y se transforma mediante una matriz invertible con coeficientes en $\mathbb{Z}/N\mathbb{Z}$. Esta operación lineal produce el vector cifrado, y la seguridad del sistema reside en la dificultad de deducir la matriz a partir del texto cifrado. El cifrado de Hill resiste el análisis de frecuencia, pero sigue siendo vulnerable si se conocen algunos pares de texto plano y texto cifrado, ya que en ese caso es posible reconstruir la matriz mediante técnicas de álgebra lineal modular.

Hasta la década de 1970, la criptografía permaneció firmemente anclada en técnicas basadas en álgebra y teoría de números elementales. Si bien se hicieron algunos avances teóricos, como el uso de secuencias de registros de desplazamiento para generar cifras pseudoseguras, la criptografía no había incorporado aún herramientas matemáticas modernas. Una excepción destacada es el teorema de Claude Shannon en 1949, que establece que la única manera de lograr una seguridad perfecta es mediante el uso de un cifrado de uso

único o *one-time pad*. Este sistema, conceptualmente una versión extrema del cifrado de Vigenère con una clave tan larga como el mensaje, garantiza que el texto cifrado no contiene ninguna información estadística sobre el mensaje original. Sin embargo, su uso práctico es limitado, ya que requiere el intercambio seguro de claves de gran tamaño.

El cambio de paradigma comenzó a gestarse en los años 60. En 1968, el investigador R. M. Needham introdujo la idea de utilizar funciones unidireccionales para proteger contraseñas en sistemas informáticos. Su propuesta consistía en cifrar la contraseña al momento de ser creada o modificada, almacenar únicamente su versión cifrada, y al momento de autenticar al usuario, volver a cifrar la entrada y compararla con la versión almacenada. De esta forma, incluso si un atacante lograba acceder a la base de datos de contraseñas cifradas, no podría usarlas directamente ni revertir fácilmente el proceso de cifrado para obtener la contraseña original. Este enfoque introdujo por primera vez el concepto práctico de una función fácil de evaluar pero difícil de invertir, una idea que sería central en la criptografía moderna.

Pocos años después, en 1974, J. F. Purdy publicó una descripción detallada de una función unidireccional concreta basada en aritmética modular. En su esquema, las contraseñas y sus representaciones cifradas eran tratadas como elementos de $\mathbb{Z}_p$, con p un primo grande, y se aplicaba un polinomio de grado alto con coeficientes grandes para transformarlas. La evaluación del polinomio era eficiente, pero invertir la función —esto es, recuperar la entrada original a partir del resultado— resultaba computacionalmente inviable. Este tipo de funciones no sólo permitía verificar identidades sin almacenar información sensible, sino que establecía las bases para una nueva clase de criptografía: aquella basada en problemas computacionalmente difíciles.

Esta idea —la existencia de funciones que son fáciles de calcular pero difíciles de invertir— condujo directamente a la aparición de la criptografía de clave pública. A diferencia de los sistemas anteriores, donde emisor y receptor debían compartir una clave secreta común, los nuevos esquemas permitirían que cada usuario tuviera una clave pública, accesible a cualquiera, y una clave privada, mantenida en secreto. La seguridad ya no dependía de mantener oculta la técnica, sino de la dificultad inherente de invertir una operación matemática concreta sin conocer cierta información. Este enfoque revolucionó la criptografía y permitió, por primera vez, aplicaciones como el intercambio seguro de claves, las firmas digitales y la autenticación descentralizada, estableciendo los cimientos de los sistemas criptográficos modernos que sustentan la seguridad digital actual.

1.2. Fundamentos matemáticos

La criptografía moderna, y en especial la de clave pública, se basa en problemas matemáticos difíciles de resolver sin información adicional. Para comprender estos sistemas y su seguridad, necesitamos herramientas de la aritmética modular, teoría de números y álgebra de grupos. Esta sección presenta los conceptos esenciales que sustentan la construcción y análisis de los esquemas criptográficos actuales.

1.2.1. Aritmética modular y teoría de números

La noción de congruencia es central en la aritmética modular y, por extensión, en muchos sistemas criptográficos modernos.

Definición 1.2.1 (Relación de congruencia)**.** Sean $a, b \in \mathbb{Z}$ y $m \in \mathbb{N}$, con $m \geq 1$. Decimos que a es

congruente con b módulo m si m divide a su diferencia, es decir, si existe un entero $k \in \mathbb{Z}$ tal que:

$$a - b = k \cdot m$$

Esta relación se denota:

$$a \equiv b \mod m$$

Esta notación indica que a y b dejan el mismo resto al dividirse por m. Más formalmente, si escribimos la división euclídea de a y b por m, se tiene:

$$a = p \cdot m + r, \quad b = q \cdot m + r, \quad \text{con } 0 \leq r < m$$

Entonces $a \equiv b \mod m$ si y solo si comparten el mismo residuo r.

La relación de congruencia módulo m es una relación de equivalencia, y además es compatible con las operaciones de suma, resta y multiplicación. Es decir, define una *congruencia* en el sentido algebraico, lo que permite construir el anillo cociente $\mathbb{Z}/m\mathbb{Z}$, cuyos elementos son las clases de equivalencia módulo m.

Proposición 1.2.2 (Propiedades de la congruencia)**.** Sean $a, b, c, d \in \mathbb{Z}$ y $m \in \mathbb{N}$. Si $a \equiv b \mod m$ y $c \equiv d \mod m$, entonces:

- $a + c \equiv b + d \mod m$
- $a - c \equiv b - d \mod m$
- $a \cdot c \equiv b \cdot d \mod m$

Estas propiedades permiten manipular congruencias como si se tratara de igualdades ordinarias, lo que resulta esencial en la construcción de algoritmos criptográficos como RSA o Diffie–Hellman. Para resolver ecuaciones congruenciales del tipo $ax \equiv 1 \mod m$, necesitamos una herramienta fundamental de la teoría de números:

Teorema 1.2.3 (Teorema de Bézout)**.** *Sean $a, b \in \mathbb{Z}$, no ambos nulos. Entonces existen enteros $x, y \in \mathbb{Z}$ tales que:*

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Este resultado garantiza que, si $\gcd(a, m) = 1$, entonces a tiene inverso módulo m.

Proposición 1.2.4 (Inverso multiplicativo módulo m)**.** Sea $a \in \mathbb{Z}$ y $m \in \mathbb{N}$. El entero a admite un inverso multiplicativo módulo m, es decir, existe $a^{-1} \in \mathbb{Z}$ tal que

$$a \cdot a^{-1} \equiv 1 \mod m$$

si y solo si $\gcd(a, m) = 1$.

Demostración. Supongamos primero que $\gcd(a, m) = 1$. Por el teorema de Bézout, existen $x, y \in \mathbb{Z}$ tales que:

$$a \cdot x + m \cdot y = 1$$

Tomando módulo m, se tiene:

$$a \cdot x \equiv 1 \mod m$$

Luego, x es un inverso de a módulo m.

Recíprocamente, si existe x tal que $a \cdot x \equiv 1 \mod m$, entonces existe $k \in \mathbb{Z}$ tal que $ax - km = 1$. Esto implica que $\gcd(a,m) = 1$ por el teorema de Bézout. □

1.2.2. Estructuras algebraicas

La aritmética modular permite construir estructuras algebraicas finitas que son esenciales para entender y diseñar sistemas criptográficos modernos. A continuación, presentamos dos de las más relevantes: los anillos y los grupos definidos módulo n.

Definición 1.2.5 (Anillo $\mathbb{Z}_n$)**.** Para un entero $n \geq 2$, se define el conjunto $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ como el conjunto de clases de equivalencia de los enteros módulo n. Cada elemento de $\mathbb{Z}_n$ es una clase de la forma $[a]_n = \{b \in \mathbb{Z} : b \equiv a \mod n\}$.

Las operaciones de suma y producto están definidas como:

$$[a]_n + [b]_n = [a+b]_n, \quad [a]_n \cdot [b]_n = [a \cdot b]_n$$

Con estas operaciones, $\mathbb{Z}_n$ forma un anillo conmutativo con unidad.

El anillo $\mathbb{Z}_n$ puede contener elementos no invertibles. Sin embargo, los enteros que son coprimos con n sí tienen inverso multiplicativo módulo n. El conjunto de todas las clases invertibles forma una estructura adicional de gran importancia.

Definición 1.2.6 (Grupo multiplicativo $\mathbb{Z}_n^*$)**.** Se define $\mathbb{Z}_n^*$ como el conjunto de clases de equivalencia $[a]_n \in \mathbb{Z}_n$ tales que $\gcd(a,n) = 1$. Bajo el producto módulo n, $\mathbb{Z}_n^*$ forma un grupo abeliano.

Este grupo es finito, y su orden está dado por la función totiente de Euler $\varphi(n)$. Cuando n es primo, todos los elementos no nulos son invertibles, por lo que:

$$\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}, \quad \text{con } |\mathbb{Z}_p^*| = p - 1$$

Dentro de $\mathbb{Z}_n^*$ con n primo, los grupos cíclicos juegan un papel especial en criptografía y en particular en la criptografía de curva elíptica.

Teorema 1.2.7. *Para todo número primo p, el grupo multiplicativo $\mathbb{Z}_p^*$ es cíclico. Es decir, existe un elemento $g \in \mathbb{Z}_p^*$ tal que:*

$$\mathbb{Z}_p^* = \{g^k \mod p \mid 0 \leq k < p-1\}$$

Sabemos que todo grupo cíclico finito de orden n tiene, para cada divisor $d \mid n$, al menos un elemento de orden d; en particular, existen elementos de orden máximo n. Este hecho se deduce utilizando el teorema de Cauchy sobre un grupo cíclico, el cual establece que:

Si G es un grupo finito de orden n y $p \mid n$ con p primo, entonces existe $g \in G$ tal que $\text{ord}(g) = p$.

Un resultado clásico de la teoría de números establece que, si p es un número primo y $d \mid (p-1)$, entonces el número de elementos de $\mathbb{Z}_p^*$ que tienen orden exactamente d es igual a $\varphi(d)$, donde φ denota la función totiente de Euler. Sumando sobre todos los divisores d de n, se tiene:

$$\sum_{d|n} \varphi(d) = n$$

Esto implica que hay exactamente $\varphi(n)$ elementos de orden n, es decir, generadores del grupo.

Ejemplo 1.2.8 (Distribución de órdenes en $\mathbb{Z}_7^*$)**.** Como 7 es primo, el grupo $\mathbb{Z}_7^* = \{1,2,3,4,5,6\}$ tiene 6 elementos. Sus divisores son $d = 1,2,3,6$, y:

$$\varphi(1) = 1, \quad \varphi(2) = 1, \quad \varphi(3) = 2, \quad \varphi(6) = 2, \quad \Rightarrow \sum_{d|6} \varphi(d) = 6$$

Busquemos los elementos de cada orden:

1. **Orden 1:** 1, ya que $1^1 \equiv 1 \mod 7$.
2. **Orden 2:** 6, ya que $6^2 \equiv 1 \mod 7$, pero $6^1 \not\equiv 1$.
3. **Orden 3:** 2 y 4, pues:

$$2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 1; \quad \text{y} \quad 4^1 = 4, \quad 4^2 = 2, \quad 4^3 = 1$$

4. **Orden 6:** 3 y 5, ya que ambos generan todo el grupo:

$$\langle 3 \rangle = \{3,2,6,4,5,1\}, \quad \langle 5 \rangle = \{5,4,6,2,3,1\}$$

Esto ilustra que en un grupo cíclico de orden n, hay exactamente $\varphi(d)$ elementos de orden d para cada divisor $d \mid n$. En particular, los de orden máximo n son los generadores del grupo.

A estos generadores se les conoce como raíces primitivas módulo p, y su existencia permite construir sistemas como el protocolo de intercambio de claves de Diffie–Hellman y el criptosistema ElGamal. En ambos casos, la seguridad está basada en la dificultad del problema del logaritmo discreto en $\mathbb{Z}_p^*$.

1.2.3. Funciones unidireccionales

Una función unidireccional es, de manera informal, una función que es fácil de evaluar en una dirección, pero difícil de invertir sin información adicional. Estas funciones son fundamentales en criptografía, ya que permiten definir operaciones seguras que solo pueden ser deshechas con una clave secreta.

Formalmente, una función $f : X \to Y$ es *unidireccional* si:

- Existe un algoritmo eficiente (en tiempo polinómico) que calcula $f(x)$ para toda $x \in X$.
- No existe un algoritmo eficiente que, dado $y \in Y$, encuentre un $x \in X$ tal que $f(x) = y$, salvo con probabilidad despreciable.

Un ejemplo clásico es la **exponenciación modular**. Dados un entero a, un exponente e y un módulo n, calcular

$$f(a) = a^e \mod n$$

es una operación rápida mediante técnicas como exponenciación binaria. Sin embargo, el problema inverso —encontrar a dado $f(a)$, e y n— se conoce como problema del logaritmo discreto y es computacionalmente difícil en general, ver anexo A sobre teoría de la complejidad.

Una propiedad clave de la exponenciación modular es que, si se conoce información adicional sobre la estructura del grupo, puede llegar a invertirse. Esta relación está formalizada por resultados clásicos como el **pequeño teorema de Fermat** que afirma que si p es primo y $a \in \mathbb{Z}_p^*$, entonces $a^{p-1} \equiv 1 \mod p$.

Este resultado es un caso particular del siguiente teorema más general:

Teorema 1.2.9 (Teorema de Euler)**.** *Sea $a \in \mathbb{Z}_n^*$ y $n \in \mathbb{N}$. Entonces se cumple:*

$$a^{\varphi(n)} \equiv 1 \mod n$$

donde $\varphi(n)$ es la función totiente de Euler, que cuenta la cantidad de enteros menores que n y coprimos con él.

Este teorema garantiza que si se conoce $\varphi(n)$, es posible calcular el inverso de la función $a \mapsto a^e \mod n$, ya que se puede hallar un exponente d tal que $ed \equiv 1 \mod \varphi(n)$. Sin embargo, calcular $\varphi(n)$ requiere conocer la factorización de n, que es precisamente el problema difícil sobre el que se basan esquemas como RSA.

El pequeño teorema de Fermat se deduce directamente del teorema de Euler: si $n = p$ es un número primo, entonces todos los enteros en $\mathbb{Z}_p^*$ son coprimos con p, por lo que $\varphi(p) = p - 1$. Al aplicar el teorema de Euler en este caso, se obtiene:

$$a^{p-1} \equiv 1 \mod p,$$

lo que coincide con la afirmación del teorema de Fermat.

Así, la exponenciación modular ofrece una transformación eficiente de evaluar, pero difícil de invertir sin información oculta —una característica esencial de las funciones unidireccionales en criptografía.

1.2.4. Números primos y pruebas de primalidad

Los números primos son los bloques fundamentales de la aritmética y, por tanto, de muchos sistemas criptográficos. En el contexto de la criptografía moderna, es necesario generar grandes números primos de forma eficiente para construir claves seguras, especialmente en algoritmos como RSA.

Un número primo se define como aquel entero mayor que 1 que sólo es divisible por 1 y por sí mismo. Por ejemplo, 2, 3, 5, 7, 11 son primos, mientras que 4, 6 y 9 no lo son.

La verificación de si un número dado es primo no es tan trivial como parece cuando se trata de enteros muy grandes, como los que se usan en criptografía (del orden de 1024 o 2048 bits). Esto ha motivado el desarrollo de algoritmos llamados *pruebas de primalidad*, que permiten distinguir primos de compuestos de forma rápida y, en muchos casos, probabilística.

Una de las primeras ideas proviene del pequeño teorema de Fermat, caso particular del Teorema 1.2.9:

Definición 1.2.10 (Test de primalidad de Fermat)**.** Sea $n \in \mathbb{N}$, impar y mayor que 2. Elegimos un entero $a \in \{2, \ldots, n-2\}$. Si:

$$a^{n-1} \not\equiv 1 \mod n,$$

entonces n es compuesto. En caso contrario, se dice que *n pasa el test de Fermat* para la base a.

Este test es eficiente, pero presenta limitaciones importantes. Existen enteros compuestos que satisfacen la congruencia anterior para algunas bases a, llamados *pseudoprimos de Fermat*. Peor aún, hay enteros compuestos que satisfacen la congruencia para todas las bases coprimas, conocidos como *números de Carmichael*. Por tanto, el test de Fermat no es fiable por sí solo.

Para resolver estas debilidades se han desarrollado pruebas probabilísticas más robustas. Una de las más utilizadas en la práctica es el test de Miller–Rabin, que mejora el de Fermat introduciendo condiciones adicionales sobre la factorización de $n-1$.

Teorema 1.2.11 (Test de Miller–Rabin (enunciado informal))**.** *Sea $n > 2$ un número impar, y sea $n-1 = 2^s \cdot d$ con d impar. Sea $a \in \{2, \ldots, n-2\}$. Entonces n falla el test de Miller–Rabin para la base a si se cumple:*

$$a^d \not\equiv 1 \mod n \quad y \quad a^{2^r \cdot d} \not\equiv -1 \mod n \quad \textit{para todo } 0 \leq r < s$$

Si se cumple esta doble condición, entonces n es compuesto.

Este test es probabilístico: si un número pasa el test para varias bases a elegidas aleatoriamente, la probabilidad de que sea compuesto disminuye exponencialmente con el número de pruebas. En la práctica, se considera que pasar el test para 20 bases independientes da una seguridad criptográficamente aceptable.

En resumen, las pruebas de primalidad basadas en exponenciación modular como las de Fermat y Miller–Rabin son herramientas fundamentales en la generación de claves seguras. Mientras el test de Fermat es simple y didáctico, Miller–Rabin ofrece una mejora significativa sin perder eficiencia y se utiliza ampliamente en la práctica.

1.3. Principios fundamentales de la criptografía

Imagina que Ana y Juan están organizando una reunión secreta para un proyecto confidencial de trabajo. Para ello, necesitan asegurarse de que sus comunicaciones sean seguras y que ningún intruso pueda interferir o manipular sus mensajes. A través de esta historia, veremos cómo los principios fundamentales de la criptografía garantizan la seguridad de su intercambio.

1. Confidencialidad: Ana decide enviar un mensaje a Juan con la dirección del lugar donde se reunirán. Para evitar que un atacante (por ejemplo, Carlos, que está espiando la comunicación) descubra esta información, Ana cifra el mensaje usando una clave secreta compartida. De este modo, incluso si Carlos intercepta el mensaje, no podrá leer su contenido sin la clave. Este proceso garantiza que la información se mantenga confidencial.

2. Integridad: Juan recibe el mensaje y quiere asegurarse de que no ha sido alterado durante la transmisión. Para ello, Ana incluye un *hash* (resumen criptográfico) del mensaje original. Al recibir el mensaje, Juan calcula el *hash* nuevamente y lo compara con el que envió Ana. Si los *hashes* coinciden, Juan puede estar seguro de que el mensaje no ha sido modificado, protegiendo así la integridad de la comunicación.

3. Autenticidad: Carlos podría intentar hacerse pasar por Ana y enviar un mensaje falso a Juan para confundirlo. Para evitarlo, Ana utiliza su clave privada para firmar digitalmente el mensaje. Juan verifica la firma utilizando la clave pública de Ana, asegurándose de que el mensaje realmente proviene de ella y no de un impostor. Este paso garantiza la autenticidad de la información.

4. No repudio: Más tarde, si Ana intentara negar que envió el mensaje, la firma digital sería una prueba irrefutable de que fue ella quien lo envió. Esto se debe a que solo Ana posee su clave privada, lo que asegura que no puede repudiar haber enviado el mensaje. De este modo, se protege el principio de no repudio.

Gracias al uso de estos principios fundamentales —confidencialidad, integridad, autenticidad y no repudio—[17], Ana y Juan logran coordinar su reunión de manera segura, incluso en presencia de un atacante como Carlos. Este ejemplo muestra cómo la criptografía es esencial para proteger la comunicación en el mundo real.

1.4. Criptografía de clave pública

La criptografía de clave pública, también conocida como criptografía asimétrica, es un marco criptográfico que utiliza un par de claves distintas: una clave pública y una clave privada. Este enfoque permite resolver limitaciones de los sistemas simétricos, como la necesidad de compartir una clave secreta de forma segura. La clave pública se utiliza para cifrar mensajes o verificar firmas digitales, mientras que la clave privada, mantenida en secreto, permite descifrar los mensajes o generar firmas digitales.

La seguridad de los sistemas de clave pública se basa en problemas matemáticos que son fáciles de calcular en una dirección, pero extremadamente difíciles de resolver en la dirección inversa sin información adicional, como la clave privada. Entre los problemas matemáticos más relevantes en este ámbito se encuentran la factorización de enteros y el problema del logaritmo discreto.

1.4.1. Factorización de enteros

Este problema consiste en encontrar los factores primos de un número compuesto grande. Mientras que multiplicar dos números primos es una operación computacionalmente simple, el proceso inverso, es decir, determinar los factores primos de un número grande, es extraordinariamente difícil para los algoritmos actuales cuando los números involucrados son suficientemente grandes. Este principio es la base del algoritmo RSA, uno de los sistemas de clave pública más utilizados, en el que la seguridad depende de la dificultad de factorizar un número compuesto que es el producto de dos primos grandes.

El algoritmo RSA utiliza esta propiedad matemática para implementar un esquema de criptografía de clave pública. Su funcionamiento se basa en los siguientes pasos[24]:

1. **Generación de claves**: Se eligen dos números primos grandes p y q, y se calcula su producto $n = p \cdot q$, conocido como el módulo. A partir de n, se calcula $\phi(n) = (p-1)(q-1)$, la función indicatriz de Euler. Luego, se selecciona un exponente público e, que sea coprimo con $\phi(n)$, y se calcula el exponente privado d, el cual es el inverso modular de e respecto a $\phi(n)$ ($d \cdot e \equiv 1$ mód $\phi(n)$).

2. **Cifrado**: Para enviar un mensaje cifrado, el remitente utiliza la clave pública del receptor, compuesta por (e,n), y cifra el mensaje m (convertido en un número menor que n) mediante:

$$c = m^e \mod n$$

donde c es el texto cifrado que se transmite.

3. **Descifrado**: El receptor, conociendo la clave privada d, descifra el texto cifrado c utilizando:

$$m = c^d \mod n$$

recuperando así el mensaje original m.

La seguridad de RSA radica en que, dado el valor de n, resulta extremadamente difícil factorizarlo para obtener p y q, y por ende calcular $\phi(n)$ y derivar la clave privada d.

1.4.2. Comparación de la criptografia de curvas elípticas con RSA

A pesar de la robustez de RSA, la criptografía de curvas elípticas (ECC, por sus siglas en inglés) ha ganado relevancia debido a su eficiencia y menor tamaño de clave. ECC se basa en un problema matemático diferente: el problema del logaritmo discreto en el contexto de las curvas elípticas sobre cuerpos finitos.

El problema del logaritmo discreto (DLP) El problema del logaritmo discreto (DLP) surge en el contexto de grupos finitos. Dados un generador g de un grupo finito G, un número primo p y un elemento $h \in G$, el problema consiste en encontrar un entero x tal que[24]:

$$g^x \equiv h \mod p \tag{1.1}$$

aunque calcular $g^x \mod p$ es computacionalmente eficiente, el proceso inverso para encontrar x es extremadamente difícil sin información adicional.

El problema del logaritmo discreto en curvas elípticas (ECDLP) El ECDLP es una extensión del DLP al contexto de las curvas elípticas sobre cuerpos finitos. La estructura matemática de las curvas elípticas permite un nivel de seguridad equivalente al DLP tradicional, pero con tamaños de clave mucho menores. Por ejemplo, una clave de 256 bits en ECC equivale en seguridad a una clave de 3072 bits en RSA.

Ventajas de la ECC ECC ofrece varias ventajas sobre RSA, especialmente en términos de eficiencia y tamaño de claves:

- **Mayor eficiencia**: Los algoritmos de ECC requieren menos operaciones matemáticas que RSA, lo que se traduce en un menor consumo de energía y tiempos de cálculo más rápidos. Esto es especialmente importante para dispositivos con recursos limitados, como sensores, dispositivos IoT y teléfonos móviles.

- **Seguridad frente a avances futuros**: Aunque tanto RSA como ECC podrían ser vulnerables a la computación cuántica mediante el algoritmo de Shor, ECC ofrece mejores perspectivas de seguridad a largo plazo debido a su menor tamaño de clave y mayor resistencia a algunos ataques.
- **Aplicaciones flexibles**: ECC soporta múltiples funcionalidades criptográficas, como el intercambio de claves (ECDH), firmas digitales (ECDSA) y cifrado, todo dentro del mismo marco matemático.

En resumen, aunque RSA sigue siendo ampliamente utilizado y ofrece robustez probada, ECC se ha convertido en una alternativa más eficiente y adaptable a las necesidades modernas de la criptografía. Con su combinación de claves más pequeñas, mayor eficiencia y flexibilidad, ECC es ideal para aplicaciones en un entorno tecnológico en evolución.

Capítulo 2

Curvas elípticas

2.1. Definición

Una **curva elíptica** E sobre un cuerpo $\mathbb{F}$ es una curva que está dada por una ecuación de la forma:

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6, \quad a_i \in \mathbb{F}. \tag{2.1}$$

Denotamos por $E(\mathbb{F})$ al conjunto de puntos $(x,y) \in \mathbb{F}^2$ que satisfacen esta ecuación, junto con un "punto en el infinito"denotado por O.

Para que la curva $(2{,}1)$ sea una curva elíptica, debe ser **suave**. Esto significa que no hay ningún punto en $E(\overline{\mathbb{F}})$ (recordemos que $\overline{\mathbb{F}}$ denota el cierre algebraico de $\mathbb{F}$) donde se anulen simultáneamente las derivadas parciales. En otras palabras, las dos ecuaciones:

$$\frac{\partial}{\partial X}\left(Y^2 + a_1XY + a_3Y - X^3 - a_2X^2 - a_4X - a_6\right) = 0 \tag{2.2}$$

$$\frac{\partial}{\partial Y}\left(Y^2 + a_1XY + a_3Y - X^3 - a_2X^2 - a_4X - a_6\right) = 0 \tag{2.3}$$

no pueden ser satisfechas simultáneamente por ningún $(x,y) \in E(\overline{\mathbb{F}})$.

Si $\mathbb{F}$ no es de característica 2, entonces, sin pérdida de generalidad, podemos suponer que $a_1 = a_3 = 0$ (se puede hacer una transformación lineal que conduce a esos parámetros). En el caso importante de característica 2, tenemos el llamado caso supersingular con $Y^2 + a_3Y$ en el lado izquierdo de la ecuación $(2{,}1)$, y el caso "no supersingularçon $Y^2 + a_1XY$ en el lado izquierdo; en este último caso, sin pérdida de generalidad podemos suponer que $a_1 = 1$ (mediante una transformación lineal como antes).

Si la característica de $\mathbb{F}$ no es ni 2 ni 3, entonces, después de simplificar el lado izquierdo de $(2{,}2)$ mediante un cambio lineal de variables (es decir, $X \to X - \frac{a_2}{3}$), también podemos eliminar el término X^2. Es decir, sin pérdida de generalidad podemos suponer que nuestra curva elíptica está dada por una ecuación de la forma[19]:

$$Y^2 = X^3 + aX + b, \quad a,b \in \mathbb{F}, \quad \text{char}(\mathbb{F}) \neq 2,3. \tag{2.4}$$

Para que la curva dada en $(2{,}4)$ sea suave, debe cumplirse que su **discriminante** no se anule [29]. En esta forma simplificada, el discriminante está dado por:

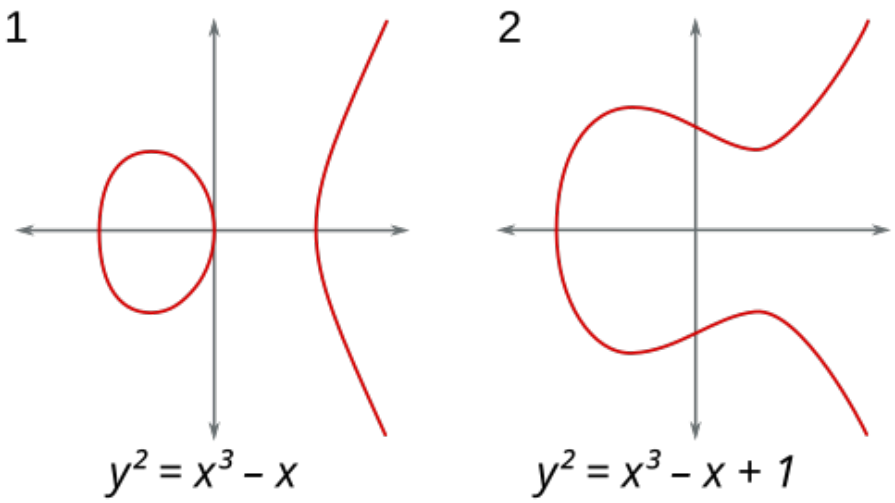


Figura 2.1: Gráficos de las curvas $y^2 = x^3 - x$ y $y^2 = x^3 - x + 1$

$$\Delta = -16(4a^3 + 27b^2), \tag{2.5}$$

y la condición de no singularidad es simplemente:

$$\Delta \neq 0.$$

Esto garantiza que la curva no tiene puntos singulares (es decir, que no hay puntos donde ambas derivadas parciales se anulan simultáneamente), y por lo tanto que E define una curva elíptica en sentido estricto.

2.2. Espacio proyectivo y el punto en el infinito

El espacio proyectivo es una herramienta fundamental para el estudio geométrico de las curvas elípticas. Nos permite extender el plano afín $\mathbb{A}^2$ añadiendo los llamados *puntos en el infinito*, y proporciona un marco riguroso para afirmaciones intuitivas como "las rectas paralelas se encuentran en el infinito". También es esencial para tratar el punto en el infinito de una curva elíptica como un punto más, sin necesidad de hacer excepciones.

Sea $\mathbb{F}$ un cuerpo. El plano proyectivo $\mathbb{P}^2_{\mathbb{F}}$ se define como el conjunto de clases de equivalencia de ternas $(x,y,z) \in \mathbb{F}^3 \setminus \{(0,0,0)\}$ bajo la relación:

$$(x_1,y_1,z_1) \sim (x_2,y_2,z_2) \iff \exists \lambda \neq 0 \in \mathbb{F} \text{ tal que } (x_1,y_1,z_1) = (\lambda x_2, \lambda y_2, \lambda z_2).$$

A una clase de equivalencia se le denota $(x : y : z)$, y se interpreta como un punto proyectivo. Si $z \neq 0$, entonces podemos dividir por z y escribir $(x : y : z) = (x/z : y/z : 1)$, lo cual nos permite identificar estos puntos con los del plano afín $\mathbb{A}^2$, mediante la inclusión:

$$(x,y) \in \mathbb{A}^2 \mapsto (x : y : 1) \in \mathbb{P}^2.$$

Los puntos con $z = 0$ no pueden representarse de esta forma y se denominan *puntos en el infinito*. Así, el espacio proyectivo $\mathbb{P}^2$ puede verse como una compactificación del plano afín: añade una línea en el infinito donde se encuentran las direcciones del plano.

Para trabajar con ecuaciones en coordenadas proyectivas, se requiere que los polinomios sean *homogéneos*. Un polinomio $F(x,y,z)$ es homogéneo de grado n si todos sus monomios tienen grado total n, es decir, si $F(\lambda x, \lambda y, \lambda z) = \lambda^n F(x,y,z)$ para todo $\lambda \in \mathbb{F}$. Esto garantiza que el conjunto de ceros de F en $\mathbb{P}^2$ esté bien definido, es decir, que no dependa del representante elegido para cada clase de equivalencia.

Dado un polinomio afín $f(x,y)$, se puede obtener su versión homogénea $F(x,y,z)$ multiplicando cada término por potencias adecuadas de z para igualar el grado. Por ejemplo:

$$f(x,y) = y^2 - x^3 - Ax - B \quad \rightsquigarrow \quad F(x,y,z) = y^2z - x^3 - Axz^2 - Bz^3.$$

La forma homogénea de una curva elíptica es útil para estudiar el punto en el infinito. Consideremos la curva elíptica afín:

$$E : y^2 = x^3 + Ax + B,$$

cuya versión proyectiva es:

$$E : y^2z = x^3 + Axz^2 + Bz^3.$$

Los puntos afines de E se corresponden con los puntos de la forma $(x : y : 1)$. Para encontrar los puntos en el infinito, basta con imponer $z = 0$ en la ecuación homogénea. Esto da:

$$y^2 \cdot 0 = x^3 \Rightarrow x = 0.$$

Como el punto $(0 : 0 : 0)$ no está permitido en el espacio proyectivo, y debe ser distinto de cero, y se puede dividir por y para obtener:

$$(0 : y : 0) = (0 : 1 : 0).$$

Así, el **único punto en el infinito** de la curva elíptica es $(0 : 1 : 0)$.

Este punto tiene una interpretación geométrica importante: todas las rectas verticales, que no se intersecan entre sí en el plano afín, se encuentran en el punto $(0 : 1 : 0)$ en el espacio proyectivo. Esto puede comprobarse fácilmente: las rectas $x = c_1$ y $x = c_2$ tienen representación homogénea independiente de y y su intersección ocurre cuando $z = 0$, $x = 0$, es decir, en $(0 : y : 0) \sim (0 : 1 : 0)$.

De manera similar, dos rectas paralelas no verticales, por ejemplo $y = mx + b_1$ y $y = mx + b_2$, tienen forma homogénea $y = mx + b_1 z$, $y = mx + b_2 z$, y su intersección ocurre en:

$$z = 0, \quad y = mx \Rightarrow (x : mx : 0) \sim (1 : m : 0).$$

Es decir, se intersectan en el punto en el infinito que representa la dirección común de las dos rectas.

En el caso de las curvas elípticas, este único punto en el infinito, $(0:1:0)$, juega un papel crucial: es el *elemento neutro* del grupo definido sobre la curva, y permite que la operación de suma esté definida globalmente sin necesidad de casos especiales.

Aunque en muchas aplicaciones criptográficas se trabaja en coordenadas afines, el uso de coordenadas proyectivas no solo facilita una formulación más elegante, sino que también permite optimizaciones computacionales (por ejemplo, evitando divisiones al representar puntos como tríos $(X:Y:Z)$ en lugar de (x,y)).

2.3. Reglas de adición de puntos

En esta sección presentamos las reglas para sumar puntos en una curva elíptica, siguiendo las directrices geométricas dadas en Koblitz[19]. Vamos a trabajar a partir de ahora con curvas en la forma reducida cuya característica no es ni 2 ni 3.

Considere una curva elíptica E definida sobre los números reales mediante la ecuación 2.4. Sea P y Q dos puntos en E. Definimos el negativo de un punto P y la suma $P+Q$ empleando las reglas siguientes:

1. **Elemento neutro:** Si P es el punto en el infinito O, entonces definimos $-P = O$. Para cualquier punto Q, la suma $O+Q$ se define como Q. De este modo, el punto en el infinito O actúa como el elemento neutro aditivo (el "cero" del grupo). A partir de aquí, supondremos que ni P ni Q son el punto en el infinito.

2. **Inversos aditivos:** El negativo de un punto $P = (x,y)$ es $-P = (x,-y)$. Dado que la curva es simétrica respecto del eje x, si (x,y) pertenece a la curva, también $(x,-y)$ lo hace. Si $Q = -P$, entonces definimos $P+Q = O$.

3. **Suma de puntos distintos:** Si P y Q tienen diferentes coordenadas x, entonces la recta $\ell = PQ$ intersecta la curva en exactamente un tercer punto R.

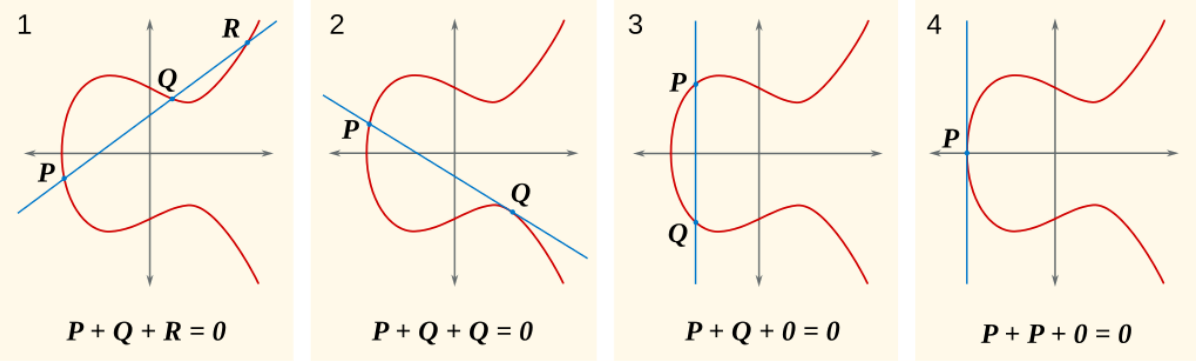
   - Si ℓ es secante (corta la curva en puntos distintos), este tercer punto R es diferente de P y Q.
   - Si ℓ es tangente en P (pero no en Q), entonces $R = P$.
   - Si ℓ es tangente en Q, entonces $R = Q$.

   Una vez encontrado R, definimos $P+Q = -R$, es decir, el simétrico de R respecto del eje x.

4. **Suma de un punto consigo mismo:** Si $P = Q$, entonces consideramos la recta tangente a la curva en P. Esta recta toca la curva en P y en otro punto R. Definimos $2P = -R$. Si la tangente es "doblemente tangente" en P, es decir, si P es un punto de inflexión, entonces $R = P$.

En resumen, las reglas anteriores se pueden condensar en la idea de que *la suma de los tres puntos donde una recta interseca la curva es igual a cero*. Si la recta pasa por el punto en el infinito O, tenemos $P+Q+O = O$, lo que implica $Q = -P$. En caso contrario, si la recta corta en tres puntos finitos P, Q y R, se cumple $P+Q+R = O$.

A continuación, mostraremos por qué la recta ℓ que pasa por P y Q intersecta a la curva exactamente en un tercer punto. Al mismo tiempo, derivaremos fórmulas explícitas para las coordenadas de este tercer punto, y por ende, para las coordenadas de $P+Q$.



Sea $P=(x_1,y_1)$, $Q=(x_2,y_2)$ y $P+Q=(x_3,y_3)$. Queremos expresar x_3 y y_3 en términos de x_1, y_1, x_2, y_2.

2.3.1. Intersección de curvas en el plano proyectivo: el teorema de Bézout

Una de las herramientas fundamentales de la geometría algebraica es el siguiente resultado clásico, que nos permite contar con precisión el número de puntos de intersección entre dos curvas:

Teorema 2.3.1 (Teorema de Bézout)**.** *Sean X y Y dos curvas proyectivas planas definidas sobre un cuerpo $\mathbb{F}$, dadas por polinomios homogéneos sin factores comunes (es decir, no comparten ningún componente irreducible). Entonces, el número total de puntos de intersección entre X y Y en $\mathbb{P}^2(\overline{F})$, contados con multiplicidad, es igual al producto de sus grados:*

$$\#(X\cap Y)=\deg(X)\cdot\deg(Y).$$

En particular, si una curva de grado tres (una cúbica) es intersectada por una recta (grado uno), el número total de puntos de intersección, contando multiplicidades y puntos en el infinito, es exactamente tres. Este hecho, aplicado a curvas elípticas, permite definir una operación geométrica de suma entre puntos sobre la curva: dados dos puntos P y Q, la recta que los une corta a la curva en un tercer punto R. El punto $P+Q$ se define entonces como la reflexión de R respecto al eje x.

A continuación, mostraremos por qué la recta ℓ que pasa por P y Q intersecta a la curva exactamente en un tercer punto. Al mismo tiempo, derivaremos fórmulas explícitas para las coordenadas de este tercer punto, y por ende, para las coordenadas de $P+Q$.

Sea $P=(x_1,y_1)$, $Q=(x_2,y_2)$ y $P+Q=(x_3,y_3)$. Queremos expresar x_3 y y_3 en términos de x_1, y_1, x_2, y_2.

Caso $P=Q$

En el caso 3), donde $P\neq Q$ y la línea ℓ no es vertical, la ecuación de la recta que pasa por P y Q puede escribirse como:

$$y=\alpha x+\beta,$$

donde

$$\alpha=\frac{y_2-y_1}{x_2-x_1}\quad\text{y}\quad\beta=y_1-\alpha x_1.$$

Un punto $(x,\alpha x+\beta)$ pertenece a la curva si y solo si $(\alpha x+\beta)^2=x^3+ax+b$. Por tanto, las intersecciones se determinan resolviendo la ecuación cúbica:

$$x^3-(\alpha x+\beta)^2+ax+b=0.$$

Sabemos que x_1 y x_2 son raíces de este polinomio (corresponden a los puntos P y Q). La suma de las raíces de una ecuación cúbica monómica es igual al negativo del coeficiente del término x^2. A partir de ello, se deduce que la tercera raíz es:

$$x_3 = \alpha^2 - x_1 - x_2.$$

De esta forma, se obtiene:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2,$$

y

$$y_3 = -y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_1 - x_3).$$

Estas fórmulas permiten calcular fácilmente el punto $P + Q = (x_3, y_3)$.

Caso $P = Q$

Cuando $P = Q$, la pendiente α se obtiene a partir de la tangente a la curva en P:

$$\alpha = \frac{3x_1^2 + a}{2y_1}.$$

Con este valor, las coordenadas de $2P$ resultan:

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1,$$

$$y_3 = -y_1 + \frac{3x_1^2 + a}{2y_1}(x_1 - x_3).$$

De este modo, hemos establecido las fórmulas explícitas para la suma de puntos en una curva elíptica, tanto para el caso $P \neq Q$ como para $P = Q$.

Teorema 2.3.2. *La suma de puntos en una curva elíptica E satisface las siguientes propiedades:*

1. ***Conmutatividad:*** *Para todos los puntos $P_1, P_2 \in E$, se tiene que $P_1 + P_2 = P_2 + P_1$.*
2. ***Existencia del neutro:*** *Para todo punto $P \in E$, se cumple que $P + O = P$, donde O es el punto en el infinito.*
3. ***Existencia de inversos:*** *Para todo punto $P \in E$, existe un punto $-P \in E$ tal que $P + (-P) = O$. Este punto $-P$ es la reflexión de P respecto al eje x.*
4. ***Asociatividad:*** *Para todos los puntos $P_1, P_2, P_3 \in E$, se cumple que $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$.*

Demostración. La conmutatividad es inmediata, ya sea a partir de las fórmulas explícitas o por el hecho geométrico de que la recta que pasa por P_1 y P_2 es la misma que la que pasa por P_2 y P_1.

La existencia del elemento neutro también se cumple por definición: el punto en el infinito O actúa como neutro respecto a la suma de puntos.

Para demostrar la existencia de inversos, basta observar que si $P = (x, y)$, entonces $-P = (x, -y)$ (en coordenadas afines). La recta vertical que pasa por ambos puntos intersecta la curva en el punto en el infinito, por lo tanto $P + (-P) = O$.

La asociatividad es, con diferencia, la propiedad más sutil y no evidente de la operaci´on de suma en curvas elípticas. A nivel algebraico, la asociatividad puede comprobarse mediante un cálculo explícito utilizando las fórmulas de suma de puntos. Sin embargo, hay varios casos a considerar: si $P_1 = P_2$, si $P_3 = -(P_1 + P_2)$, etc., lo que hace que la demostración directa sea tediosa y propensa a errores. Existe una demostración alternativa basada en una interpretación geométrica más profunda, que se puede consultar en el libro [29]. □

Ejemplo 2.3.3. A continuación, ilustramos la operación de suma de puntos en curvas elípticas concretas, aplicando las fórmulas deducidas anteriormente.

- En la curva
$$y^2 = \frac{x(x+1)(2x+1)}{6},$$
se tiene:
$$(0,0) + (1,1) = \left(\frac{1}{2}, -\frac{1}{2}\right), \quad \left(\frac{1}{2}, -\frac{1}{2}\right) + (1,1) = (24, -70).$$

- En la curva
$$y^2 = x^3 - 25x,$$
al duplicar el punto $(-4, 6)$ se obtiene:
$$2(-4,6) = (-4,6) + (-4,6) = \left(\frac{1681}{144}, -\frac{62279}{1728}\right).$$

- En la misma curva, también se cumple:
$$(0,0) + (-5,0) = (5,0), \quad 2(0,0) = 2(-5,0) = 2(5,0) = O.$$
Esto ocurre porque los puntos $(0,0)$, $(-5,0)$ y $(5,0)$ tienen orden dos: son iguales a sus inversos aditivos.

2.4. El invariante j y la clasificación de curvas elípticas

Sea E una curva elíptica definida sobre un cuerpo K de característica distinta de 2 y 3, dada por la ecuación de Weierstrass reducida $y^2 = x^3 + Ax + B, \quad A, B \in K$. Al aplicar un cambio de variable del tipo

$$x \mapsto \mu^2 x, \quad y \mapsto \mu^3 y, \quad \mu \neq 0 \in K,$$

la ecuación se transforma en otra curva elíptica E' de la forma:

$$y_1^2 = x_1^3 + A_1 x_1 + B_1,$$

donde los nuevos coeficientes son:

$$A_1 = \mu^4 A, \quad B_1 = \mu^6 B.$$

Este tipo de transformación preserva la estructura del grupo y el comportamiento geométrico de la curva. En particular, motiva la definición siguiente.

Definición 2.4.1. El **invariante** j de una curva elíptica $E : y^2 = x^3 + Ax + B$ se define como:

$$j(E) = 1728 \cdot \frac{4A^3}{4A^3 + 27B^2}.$$

El denominador es el negativo del discriminante de la cúbica $x^3 + Ax + B$, y se supone distinto de cero para que la curva sea no singular. Este valor es invariante bajo los cambios de variable anteriores.

El siguiente teorema muestra que el invariante j clasifica completamente las curvas elípticas sobre un cuerpo algebraicamente cerrado.

Teorema 2.4.2. *Sean $E_1 : y_1^2 = x_1^3 + A_1x_1 + B_1$ y $E_2 : y_2^2 = x_2^3 + A_2x_2 + B_2$ dos curvas elípticas definidas sobre un cuerpo algebraicamente cerrado K, con invariantes j_1 y j_2 respectivamente.*

Si $j_1 = j_2$, entonces existe $\mu \neq 0 \in K$ tal que:

$$A_2 = \mu^4 A_1, \quad B_2 = \mu^6 B_1,$$

y el cambio de variables

$$x_2 = \mu^2 x_1, \quad y_2 = \mu^3 y_1$$

transforma una ecuación en la otra. Es decir, E_1 y E_2 son isomorfas sobre K.

Demostración. Supongamos primero que $A_1 \neq 0$, lo cual implica que $j_1 \neq 0$, y por igualdad de invariantes, también $A_2 \neq 0$.

Como $j_1 = j_2$, se tiene:

$$\frac{4A_1^3}{4A_1^3 + 27B_1^2} = \frac{4A_2^3}{4A_2^3 + 27B_2^2}.$$

Multiplicando en cruz:

$$4A_1^3 \cdot (4A_2^3 + 27B_2^2) = 4A_2^3 \cdot (4A_1^3 + 27B_1^2).$$

Simplificando y cancelando términos comunes:

$$4A_1^3 \cdot 27B_2^2 = 4A_2^3 \cdot 27B_1^2 \Rightarrow \frac{B_2^2}{A_2^3} = \frac{B_1^2}{A_1^3}.$$

Ahora elegimos $\mu \neq 0 \in K$ tal que $A_2 = \mu^4 A_1$. Entonces:

$$A_2^3 = \mu^{12} A_1^3 \Rightarrow \frac{B_2^2}{\mu^{12} A_1^3} = \frac{B_1^2}{A_1^3} \Rightarrow B_2^2 = \mu^{12} B_1^2.$$

Esto implica que $B_2 = \pm\mu^6 B_1$. Si $B_2 = \mu^6 B_1$, el resultado queda demostrado.

En caso de que $B_2 = -\mu^6 B_1$, basta tomar $\mu' = i\mu$, donde $i^2 = -1$, y se sigue teniendo $A_2 = (\mu')^4 A_1$ y $B_2 = (\mu')^6 B_1$, ya que $i^6 = -1$.

Si $A_1 = 0$, entonces también $A_2 = 0$ (pues $j = 1728$ implica que $A = 0$), y se tiene $B_1, B_2 \neq 0$. Entonces basta tomar $\mu \neq 0 \in K$ tal que $B_2 = \mu^6 B_1$.

En todos los casos, existe $\mu \neq 0 \in K$ que transforma una curva en la otra mediante:

$$x_2 = \mu^2 x_1, \quad y_2 = \mu^3 y_1.$$

□

Existen dos valores particulares del invariante j que surgen con frecuencia: $j = 0$ y $j = 1728$.

- El caso $j = 0$ corresponde a curvas de la forma $y^2 = x^3 + B$, es decir, con $A = 0$.
- El caso $j = 1728$ corresponde a curvas de la forma $y^2 = x^3 + Ax$, es decir, con $B = 0$.

Estas curvas presentan simetrías adicionales: en el primer caso, la curva admite el automorfismo $(x,y) \mapsto (\zeta x, -y)$, donde ζ es una raíz cúbica de la unidad; en el segundo, se tiene el automorfismo $(x,y) \mapsto (-x, iy)$, donde $i^2 = -1$. Estas simetrías adicionales pueden hacer que la curva sea más vulnerable en ciertos contextos criptográficos, por lo que es habitual evitarlas en la práctica.

Por otro lado, dado un valor $j \in K$ con $j \neq 0, 1728$, se puede construir una curva elíptica con invariante j mediante la fórmula

$$y^2 = x^3 + \frac{3j}{1728 - j}x + \frac{2j}{1728 - j}.$$

Esta expresión establece una biyección entre los valores de $j \in K$ y las clases de isomorfismo (sobre K) de curvas elípticas definidas sobre K. Es decir, a cada valor de j le corresponde, salvo isomorfismo, una única curva elíptica definida sobre K.

Aunque en esta sección se ha supuesto que la característica del cuerpo base es distinta de 2 y 3, el invariante j también puede definirse en esas características, y cumple propiedades similares. Sin embargo, las fórmulas explícitas y las condiciones sobre la no singularidad cambian, y deben tratarse con más cuidado[29].

2.5. Puntos de torsión

Se denomina *punto racional* sobre un cuerpo K a todo punto de una curva elíptica cuyas coordenadas x e y pertenecen a K. El conjunto de estos puntos se denota por $E(K)$. Aunque el término "racional" proviene históricamente del caso $K = \mathbb{Q}$, su uso se ha extendido por analogía a cualquier cuerpo.

En el estudio de curvas elípticas sobre un cuerpo K, los puntos cuyas coordenadas pertenecen a K forman un grupo abeliano. Un subconjunto particularmente importante está formado por aquellos puntos que tienen orden finito. Estos puntos se conocen como *puntos de torsión*, y sus propiedades revelan la estructura interna del grupo $E(K)$.

Sea E una curva elíptica definida sobre un cuerpo K y sea n un entero positivo. Nos interesa estudiar el conjunto siguiente denominado subgrupo de n-torsión de E.

$$E[n] = \{P \in E(\overline{K}) \mid nP = O\}, \tag{2.6}$$

donde $\overline{K}$ denota la clausura algebraica de K. Es importante destacar que los puntos de $E[n]$ pueden no tener coordenadas en K, sino en su clausura algebraica.

Caso $n = 2$

Proposición 2.5.1. Sea E una curva elíptica sobre un cuerpo K. Si la característica de K no es 2, entonces

$$E[2] \cong \mathbb{Z}_2 \oplus \mathbb{Z}_2.$$

Si la característica de K es 2, entonces

$$E[2] \cong 0 \quad \text{o} \quad E[2] \cong \mathbb{Z}_2.$$

Demostración. Si la característica de K es distinta de 2, E puede escribirse en la forma $y^2 = (x-e_1)(x-e_2)(x-e_3)$ con $e_1, e_2, e_3 \in K$. Un punto P verifica $2P = \infty$ si y solo si la tangente en P es vertical, lo cual implica que $y = 0$. Por tanto:

$$E[2] = \{\infty, (e_1,0), (e_2,0), (e_3,0)\},$$

y como grupo abstracto, $E[2] \cong \mathbb{Z}_2 \oplus \mathbb{Z}_2$.

En característica 2, la situación es más sutil. Existen dos formas normales para E:

(I) $y^2 + xy = x^3 + a_2x^2 + a_6$ (con $a_6 \neq 0$),

(II) $y^2 + a_3y = x^3 + a_4x + a_6$ (con $a_3 \neq 0$).

En el caso (I), la derivada parcial respecto de y se anula si y solo si $x = 0$. Sustituyendo en la ecuación, obtenemos $y^2 = a_6$, por lo que el único punto de orden 2 es $(0, \sqrt{a_6})$:

$$E[2] = \{\infty, (0, \sqrt{a_6})\} \cong \mathbb{Z}_2.$$

En el caso (II), como $\frac{\partial}{\partial y} = a_3 \neq 0$, no hay puntos de orden 2 distintos de ∞:

$$E[2] = \{\infty\} \cong 0.$$

□

Caso $n = 3$

Supongamos que la característica de K no es 2 ni 3, de modo que E puede escribirse como $y^2 = x^3 + Ax + B$. Un punto P verifica $3P = \infty$ si y solo si $2P = -P$, lo cual implica que la coordenada x de $2P$ es igual a la de P.

Calculamos:

$$m^2 - 2x = x, \text{donde} \quad m = \frac{3x^2 + A}{2y},$$

usando la expresión algebraica de la curva llegamos a

$$(3x^2 + A)^2 = 12x(x^3 + Ax + B),$$

que se reduce a

$$3x^4 + 6Ax^2 + 12Bx - A^2 = 0.$$

Este polinomio tiene discriminante $\neq 0$, por lo que posee cuatro raíces distintas. Cada raíz x da lugar a dos posibles valores de y, de modo que hay 8 puntos de orden 3. Sumando el punto en el infinito, se concluye:

$$E[3] \cong \mathbb{Z}_3 \oplus \mathbb{Z}_3.$$

Teorema 2.5.2. *Sea E una curva elíptica definida sobre un cuerpo K y n un entero positivo.*

- *Si la característica de K no divide a n, o es cero, entonces:*

$$E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n.$$

- *Si la característica de K es $p > 0$ y $p \mid n$, escribiendo $n = p^r n'$ con $p \nmid n'$, entonces:*

$$E[n] \cong \mathbb{Z}_{n'} \oplus \mathbb{Z}_{n'} \quad o \quad \mathbb{Z}_n \oplus \mathbb{Z}_{n'}.$$

Demostración. Consultar la sección 3.2 del libro [29] □

Representaciones

Sea n un entero positivo no divisible por la característica de K. Si $\{\beta_1, \beta_2\}$ es una base de $E[n] \cong \mathbb{Z}_n \oplus \mathbb{Z}_n$, todo $P \in E[n]$ puede escribirse de forma única como $P = m_1\beta_1 + m_2\beta_2$, con $m_1, m_2 \in \mathbb{Z}_n$.

Sea $\alpha : E(\overline{K}) \to E(\overline{K})$ un homomorfismo. Entonces:

$$\alpha(\beta_1) = a\beta_1 + c\beta_2, \quad \alpha(\beta_2) = b\beta_1 + d\beta_2,$$

para ciertos $a, b, c, d \in \mathbb{Z}_n$. Es decir, α puede representarse como una matriz:

$$\alpha_n = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

La composición de homomorfismos corresponde a la multiplicación de matrices.

Ejemplo 2.5.3. Sea E la curva elíptica definida sobre $\mathbb{R}$ por $y^2 = x^3 - 2$ y $n = 2$. Entonces:

$$E[2] = \left\{\infty, \left(2^{1/3}, 0\right), \left(\zeta 2^{1/3}, 0\right), \left(\zeta^2 2^{1/3}, 0\right)\right\},$$

donde ζ es una raíz cúbica primitiva de la unidad. Definimos:

$$\beta_1 = (2^{1/3}, 0), \quad \beta_2 = (\zeta 2^{1/3}, 0).$$

Entonces $\{\beta_1, \beta_2\}$ es una base de $E[2]$, y $\beta_1 + \beta_2 = (\zeta^2 2^{1/3}, 0) = \beta_3$.

Sea $\alpha : E(\mathbb{C}) \to E(\mathbb{C})$ la conjugación compleja: $\alpha(x, y) = (\overline{x}, \overline{y})$. Como todas las fórmulas del grupo son reales, se tiene:

$$\alpha(P_1 + P_2) = \alpha(P_1) + \alpha(P_2).$$

En particular:

$$\alpha(\beta_1) = \beta_1, \quad \alpha(\beta_2) = \beta_3 = \beta_1 + \beta_2.$$

La matriz correspondiente a α es:

$$\alpha_2 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix},$$

y cumple $\alpha_2^2 = I$, como se espera de una conjugación.

2.6. Curvas elípticas sobre cuerpos finitos

En esta sección estudiaremos el comportamiento de las curvas elípticas cuando el cuerpo base es finito, es decir, cuando trabajamos con $\mathbb{F}_q$, donde $q = p^n$ es una potencia de un primo.

El conjunto de puntos $E(\mathbb{F}_q)$ con la operación de suma definida geométricamente, forma un grupo abeliano finito. Uno de los primeros resultados importantes es que dicho grupo tiene una estructura muy concreta:

Teorema 2.6.1. *Sea E una curva elíptica definida sobre un cuerpo finito $\mathbb{F}_q$. Entonces el grupo de puntos $E(\mathbb{F}_q)$ es isomorfo a uno de los siguientes tipos:*

$$\mathbb{Z}_n \quad \textit{o bien} \quad \mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2},$$

para ciertos enteros $n \geq 1$, o $n_1, n_2 \geq 1$ tales que $n_1 \mid n_2$.

Demostración. Un resultado básico de teoría de grupos afirma que todo grupo abeliano finito es isomorfo a una suma directa de grupos cíclicos

$$\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2} \oplus \cdots \oplus \mathbb{Z}_{n_r},$$

donde $n_i \mid n_{i+1}$ para todo $i \geq 1$. Dado que, para cada i, el grupo $\mathbb{Z}_{n_i}$ tiene exactamente n_1 elementos de orden que divide a n_1, concluimos que $E(\mathbb{F}_q)$ contiene al menos n_1^r elementos de orden que divide a n_1. Sin embargo, por el Teorema 2.5.2, el número total de tales puntos está acotado por n_1^2. Por tanto, debe cumplirse:

$$n_1^r \leq n_1^2 \quad \Rightarrow \quad r \leq 2.$$

Este es el resultado deseado. El caso en que $r = 0$ corresponde a un grupo trivial, lo cual también está cubierto en la afirmación del teorema tomando $n = 1$. □

Una cuestión fundamental en aplicaciones criptográficas es cuántos puntos tiene una curva elíptica sobre un cuerpo finito. Este número no es arbitrario, está acotado por el teorema de Hasse que afirma $\left|\#E(\mathbb{F}_q) - (q+1)\right| \leq 2\sqrt{q}$. Vamos a enunciar el teorema y su demostración más adelante (asumiendo sin demostrar un lema importante), pero necesitamos primero conocer una serie de resultados previos.

2.6.1. El endomorfismo de Frobenius

Sea $\mathbb{F}_q$ un cuerpo finito con clausura algebraica $\overline{\mathbb{F}}_q$, y sea

$$\begin{array}{rccc} \phi_q : \overline{\mathbb{F}}_q & \longrightarrow & \overline{\mathbb{F}}_q \\ x & \longmapsto & x^q \end{array} \tag{2.7}$$

el endomorfismo de Frobenius sobre $\mathbb{F}_q$. Sea E una curva elíptica definida sobre $\mathbb{F}_q$. Entonces ϕ_q actúa sobre las coordenadas de los puntos en $E(\mathbb{F}_q)$ mediante:

$$\phi_q(x,y) = (x^q, y^q), \quad \phi_q(\infty) = \infty.$$

Lema 2.6.2. Sea E una curva definida sobre $\mathbb{F}_q$, y sea $(x,y) \in E(\mathbb{F}_q)$.

1. $\phi_q(x,y) \in E(\mathbb{F}_q)$,

2. $(x,y) \in E(\mathbb{F}_q)$ si y solo si $\phi_q(x,y) = (x,y)$.

Demostración. Un hecho que necesitamos es que $(a+b)^q = a^q + b^q$ cuando q es una potencia de la característica del cuerpo. También necesitamos que $a^q = a$ para todo $a \in \mathbb{F}_q$.

Como la demostración es la misma para la ecuación de Weierstrass estándar y la generalizada, trabajamos con la forma general. Tenemos:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

con $a_i \in \mathbb{F}_q$. Elevamos la ecuación a la q-ésima potencia y obtenemos:

$$(y^q)^2 + a_1(x^q y^q) + a_3(y^q) = (x^q)^3 + a_2(x^q)^2 + a_4(x^q) + a_6.$$

Esto significa que (x^q, y^q) pertenece a E, lo cual prueba la parte (1).

Para la parte (2), sabemos que $x \in \mathbb{F}_q$ si y solo si $\phi_q(x) = x$, y de forma similar para y. Por lo tanto:

$$(x,y) \in E(\mathbb{F}_q) \Leftrightarrow x,y \in \mathbb{F}_q \Leftrightarrow \phi_q(x) = x \text{ y } \phi_q(y) = y \Leftrightarrow \phi_q(x,y) = (x,y).$$

□

El siguiente resultado es clave para contar puntos en curvas elípticas sobre cuerpos finitos. Como ϕ_q es un endomorfismo de E, también lo son $\phi_q^2 = \phi_q \circ \phi_q$ y $\phi_q^n = \phi_q \circ \phi_q \circ \cdots \circ \phi_q$ para cada $n \geq 1$. Como la multiplicación por -1 también es un endomorfismo, la suma $\phi_q^n - 1$ es un endomorfismo de E.

Proposición 2.6.3. Sea E una curva definida sobre $\mathbb{F}_q$ y sea $n \geq 1$.

1. $\ker(\phi_q^n - 1) = E(\mathbb{F}_{q^n})$,
2. $\phi_q^n - 1$ es un endomorfismo separable, por lo tanto $\#E(\mathbb{F}_{q^n}) = \deg(\phi_q^n - 1)$.

Demostración. Como ϕ_q^n es el endomorfismo de Frobenius para el cuerpo $\mathbb{F}_{q^n}$, la parte (1) es simplemente una reformulación del Lema 2.6.2. De hecho, sea $P \in E(\overline{\mathbb{F}}_q)$. Entonces $P \in \ker(\phi_q^n - 1)$ si y solo si

$$(\phi_q^n - 1)(P) = O \quad \Leftrightarrow \quad \phi_q^n(P) = P,$$

lo cual significa que P es un punto fijo por el Frobenius de orden n, es decir, que sus coordenadas están en $\mathbb{F}_{q^n}$. Por tanto:

$$\ker(\phi_q^n - 1) = E(\mathbb{F}_{q^n}).$$

La demostración de la parte (2) se puede consultar en [29]. □

2.6.2. Teorema de Hasse

El número de puntos $\#E(\mathbb{F}_q)$ sobre una curva elíptica no es evidente a priori y puede parecer arbitrario. Sin embargo, para aplicaciones criptográficas es fundamental saber cuántos puntos hay en una curva, ya que de ello depende la estructura del grupo y, por tanto, la seguridad de los sistemas que lo usan. El **teorema de Hasse** establece una cota precisa que garantiza que $\#E(\mathbb{F}_q)$ está siempre cerca de $q+1$, lo que permite construir curvas con grupos de tamaño controlado y evita elecciones inseguras.

Podemos ahora demostrar el teorema de Hasse. Sea

$$a = q+1-\#E(\mathbb{F}_q) = q+1-\deg(\phi_q-1). \tag{2.8}$$

y asumimos el siguiente lema

Lema 2.6.4. Sean $r,s \in \mathbb{Z}$ tales que $\gcd(s,q)=1$. Entonces:

$$\deg(r\phi_q - s) = r^2q + s^2 - rsa,$$

donde a es como en la ecuación (2.8).

Entonces podemos enunciar y demostrar el teorema de Hasse

Teorema 2.6.5 (Hasse)**.** *Sea E una curva elíptica sobre el cuerpo finito $\mathbb{F}_q$. Entonces el orden de $E(\mathbb{F}_q)$ satisface:*

$$\left|q+1-\#E(\mathbb{F}_q)\right| \leq 2\sqrt{q}.$$

Demostración. Como $\deg(r\phi_q - s) \geq 0$, el lema implica que

$$q\left(\frac{r}{s}\right)^2 - a\left(\frac{r}{s}\right) + 1 \geq 0$$

para todos los enteros r,s tales que $\gcd(s,q)=1$.

Vamos a ver que el conjunto de números racionales $\frac{r}{s}$ con $\gcd(s,q)=1$ es denso en $\mathbb{R}$. Tomemos s como una potencia de 2 o de 3; al menos una de ellas será coprima con q. Como los racionales de la forma $r/2^m$ y $r/3^m$ son densos en $\mathbb{R}$, se concluye que el conjunto de números racionales $\frac{r}{s}$ con $\gcd(s,q)=1$ es denso en $\mathbb{R}$.

Por tanto, la desigualdad

$$qx^2 - ax + 1 \geq 0$$

se cumple para todo $x \in \mathbb{R}$. Esto implica que el discriminante del polinomio cuadrático es negativo o nulo, es decir:

$$a^2 - 4q \leq 0 \quad \Rightarrow \quad |a| \leq 2\sqrt{q}.$$

Esto concluye la demostración del teorema de Hasse. □

2.7. Cómo encontrar puntos en una curva

En esta sección nos centraremos en curvas elípticas definidas sobre cuerpos finitos $\mathbb{F}_q$, ya que son las que se utilizan en la mayoría de aplicaciones criptográficas. A diferencia del caso sobre $\mathbb{Q}$ o $\mathbb{R}$, el hecho de que $E(\mathbb{F}_q)$ sea un grupo finito y computable es lo que permite definir el problema del logaritmo discreto en curvas elípticas, base de la seguridad en criptografía elíptica.

El problema de encontrar puntos sobre una curva elíptica definida sobre un cuerpo finito $\mathbb{F}_q$ es crucial en criptografía. Una curva elíptica E sobre $\mathbb{F}_q$ está dada por una ecuación de la forma (2.4), donde $a, b \in \mathbb{F}_q$ y la condición $4a^3 + 27b^2 \neq 0$ garantiza que E sea no singular. Para encontrar puntos $(x, y) \in E(\mathbb{F}_q)$, es necesario determinar para cada $x \in \mathbb{F}_q$ si el valor $x^3 + ax + b$ es un cuadrado en $\mathbb{F}_q$.

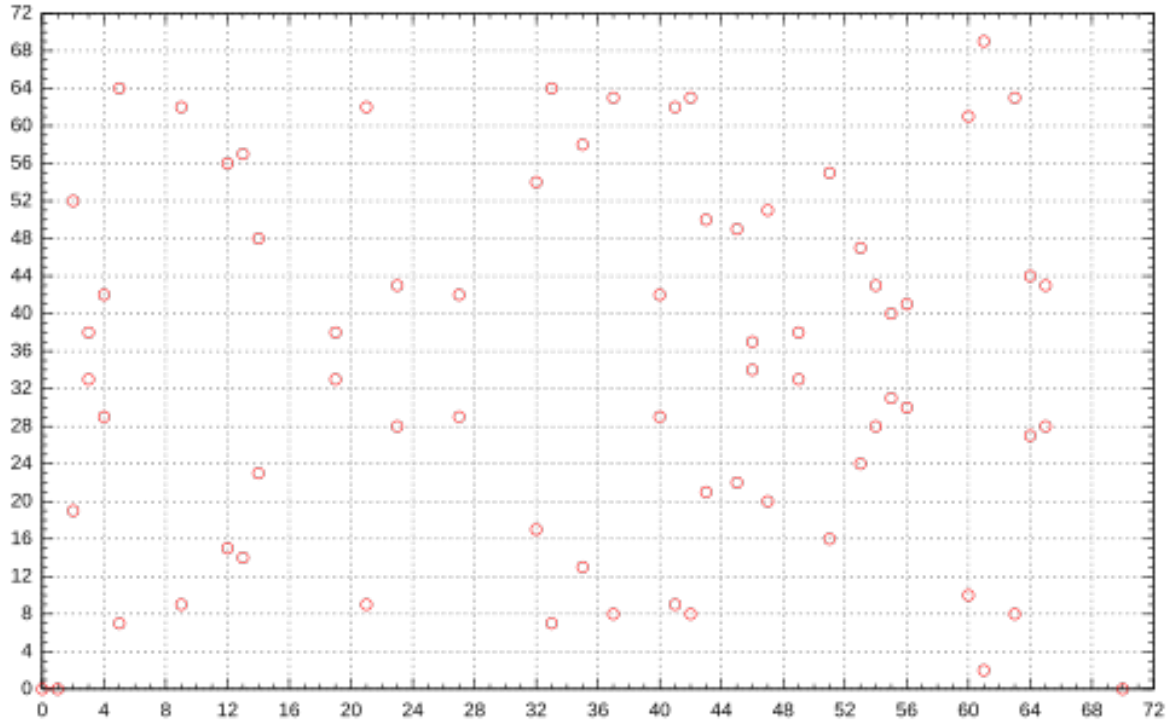


Figura 2.2: Conjunto de puntos afines de la curva elíptica $y^2 = x^3 - x$ sobre el cuerpo finito $\mathbb{F}_{71}$

2.7.1. Cálculo eficiente de raíces cuadradas en $\mathbb{F}_q$

Sea p un número primo impar. Un entero a se dice *residuo cuadrático* módulo p si existe algún $x \in \mathbb{F}_p$ tal que $x^2 \equiv a \pmod{p}$. En caso contrario, se dice que a es un *no residuo cuadrático*. El *símbolo de Legendre* es una función que permite determinar de manera eficiente si un número dado es un residuo cuadrático módulo p, y se define como:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{si } a \text{ es un residuo cuadrático y } a \not\equiv 0 \pmod{p}, \\ -1 & \text{si } a \text{ es un no residuo cuadrático,} \\ 0 & \text{si } a \equiv 0 \pmod{p}. \end{cases}$$

Legendre definió originalmente este símbolo a través de la siguiente congruencia:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}, \quad \text{con } \left(\frac{a}{p}\right) \in \{-1,0,1\}.$$

Esta definición es equivalente al *criterio de Euler*, descubierto previamente, y permite computar el símbolo de Legendre mediante una única exponenciación modular.

Este criterio es esencial en el contexto de curvas elípticas sobre $\mathbb{F}_p$, ya que para cada punto $(x,y) \in E(\mathbb{F}_p)$, el valor y^2 debe igualar un valor $v = x^3 + ax + b$ módulo p. Por tanto, verificar si v es un cuadrado en $\mathbb{F}_p$ equivale a verificar si $\left(\frac{v}{p}\right) = 1$. Si esto se cumple, entonces existen exactamente dos soluciones distintas para y, dadas por $y = \sqrt{v}$ y $y = -\sqrt{v}$ en $\mathbb{F}_p$.

Encontrar raíces cuadradas módulo p de manera eficiente es, por tanto, un paso clave en la construcción del conjunto de puntos $E(\mathbb{F}_p)$. Mientras que una búsqueda por fuerza bruta requeriría $O(p)$ operaciones, existen algoritmos más rápidos que permiten hallar raíces cuadradas cuando estas existen. En los casos particulares en los que $p \equiv 3 \pmod 4$ o $p \equiv 5 \pmod 8$, se pueden emplear fórmulas directas. Sin embargo, para el caso general, especialmente cuando $p \equiv 1 \pmod 8$, es necesario utilizar el algoritmo de Tonelli–Shanks, que presentaremos a continuación.

2.7.1.1. El método Tonelli-Shanks

Sea p un número primo impar y supongamos que hemos verificado que a es un residuo cuadrático módulo p, es decir, que el símbolo de Legendre $\left(\frac{a}{p}\right) = 1$. Esto garantiza, por definición, que existe algún $x \in \mathbb{F}_p$ tal que $x^2 \equiv a \pmod p$. La pregunta es: ¿cómo encontrar tal x?

Una búsqueda por fuerza bruta requeriría probar todos los posibles valores de x hasta hallar uno tal que $x^2 \equiv a \pmod p$, lo cual implica un tiempo de ejecución de orden $O(p)$. Esto es completamente ineficiente incluso para primos de tamaño moderado, por lo que necesitamos un algoritmo más rápido para resolver este problema.

Antes de presentar el algoritmo de Tonelli–Shanks, que funciona para cualquier primo impar, exploraremos soluciones más simples que funcionan en ciertos casos particulares. Estas soluciones, aunque no generalizables, permiten ilustrar algunas ideas importantes y constituyen una buena introducción al problema.

En primer lugar, existe una solución directa cuando $p \equiv 3 \pmod 4$. En este caso, afirmamos que una raíz cuadrada de a módulo p está dada por

$$x \equiv a^{(p+1)/4} \pmod{p}.$$

La validez de esta fórmula se puede comprobar fácilmente: dado que a es un residuo cuadrático, entonces $a^{(p-1)/2} \equiv 1 \pmod p$, por el criterio de Euler. Por tanto,

$$x^2 = \left(a^{(p+1)/4}\right)^2 = a^{(p+1)/2} = a \cdot a^{(p-1)/2} \equiv a \cdot 1 = a \pmod{p},$$

como se quería.

Un segundo caso especial es cuando $p \equiv 5 \pmod 8$. Aquí, se tiene nuevamente que $a^{(p-1)/2} \equiv 1 \pmod p$, y además, como $\mathbb{F}_p$ es un cuerpo, se sigue que $a^{(p-1)/4} \equiv \pm 1 \pmod p$. Si se da el caso favorable en que

$a^{(p-1)/4} \equiv 1 \pmod{p}$, entonces se puede probar que [11]

$$x \equiv a^{(p+3)/8} \pmod{p}$$

es una solución válida de $x^2 \equiv a \pmod{p}$. En el caso en que $a^{(p-1)/4} \equiv -1 \pmod{p}$, se puede usar que $2^{(p-1)/2} \equiv -1 \pmod{p}$, lo cual es cierto cuando $p \equiv 5 \pmod{8}$, y comprobar que

$$x \equiv 2a \cdot (4a)^{(p-5)/8} \pmod{p}$$

es una raíz cuadrada de a.

En resumen, estos dos casos permiten calcular raíces cuadradas módulo p de forma eficiente siempre que $p \equiv 3 \pmod{4}$ o $p \equiv 5 \pmod{8}$. Sin embargo, cuando $p \equiv 1 \pmod{8}$, la situación es considerablemente más difícil. Aunque existen métodos que producen infinitas familias de soluciones en este caso, ninguno de ellos resulta práctico para el cálculo general de raíces cuadradas módulo p.

Es precisamente para cubrir este caso general que introducimos el algoritmo de Tonelli–Shanks, un método eficiente y aplicable a cualquier primo impar p, cuya idea se basa en la estructura del grupo multiplicativo $\mathbb{F}_p^*$ y en la descomposición de $p-1$ como $2^s \cdot q$ con q impar. El método descrito aquí es el elaborado por Shanks.

Dado $n \in \mathbb{F}_q$ con $\left(\frac{n}{q}\right) = 1$, es decir, n es un residuo cuadrático, el objetivo es hallar x tal que $x^2 \equiv n \pmod{q}$.

El método se basa en descomponer $q-1$ en la forma

$$q - 1 = 2^s m, \quad \text{con } m \text{ impar.}$$

Se elige un elemento $z \in \mathbb{F}_q$ que sea un no residuo cuadrático, $\left(\frac{z}{q}\right) = -1$. Luego se definen

$$c = z^m, \quad t = n^m, \quad r = n^{\frac{m+1}{2}}.$$

Mientras $t \neq 1$, se busca el menor $i \geq 0$ tal que $t^{2^i} \equiv 1 \pmod{q}$. Entonces se calcula

$$b = c^{2^{s-i-1}},$$

y se actualizan

$$r = rb, \quad t = tb^2, \quad c = b^2.$$

Cuando $t = 1$, se ha encontrado la raíz cuadrada:

$$x = r.$$

Este procedimiento tiene una complejidad computacional razonable y resulta especialmente útil para calcular las raíces cuadradas necesarias en el proceso de hallar puntos en la curva elíptica.

2.7.2. Generación de puntos adicionales mediante multiplicación escalar

Una vez localizado un punto $P = (x_0, y_0) \in E(\mathbb{F}_q)$, es posible generar nuevos puntos aplicando la operación de suma de puntos, que es la operación de grupo definida en la curva:

$$P + P = 2P,$$

Repitiendo esta operación podemos calcular kP de forma eficiente, donde k es un entero. Esto es fundamental en criptografía, ya que el cálculo rápido de kP sobre curvas elípticas es la base de los algoritmos involucrados en sistemas de clave pública, intercambio de claves y firmas digitales.

En resumen, el proceso para encontrar y generar puntos sobre una curva elíptica definida en un cuerpo finito $\mathbb{F}_q$ comienza con la determinación de valores x tales que $x^3 + ax + b$ sea un residuo cuadrático y, por ende, pueda calcularse su raíz $\sqrt{v}$ eficientemente (por ejemplo, con Tonelli-Shanks). Una vez hallado un punto no trivial, la estructura de grupo de la curva permite generar múltiples puntos a partir de él mediante la multiplicación escalar, un paso imprescindible en las aplicaciones criptográficas actuales.

2.8. Cómo contar los puntos de una curva

El conteo de puntos sobre curvas elípticas definidas sobre cuerpos finitos es un problema central en teoría de números y criptografía. Un avance notable en la eficiencia de estos conteos provino del trabajo de Schoof [26], quien redujo la complejidad de métodos previos (del orden de $O(q^{1/4+\varepsilon})$), en el algorítmo Baby-step Giant-step (BSGS), a aproximadamente $O((\log q)^s)$, con $s > 0$ [27]. Este resultado ha convertido al algoritmo de Schoof en la base de los métodos modernos para contar puntos en curvas elípticas.

La idea fundamental radica en la traza de Frobenius. Si E es una curva elíptica sobre $\mathbf{F}_q$, el número de puntos en $E(\mathbf{F}_q)$ satisface el Teorema de Hasse 2.6.5:

$$\#E(\mathbf{F}_q) = q + 1 - t \quad \text{con} \quad |t| \leq 2\sqrt{q}.$$

Conocer t permite determinar el orden del grupo $E(\mathbf{F}_q)$.

El algoritmo de Schoof calcula t a través de sus restos módulo varios primos ℓ. Se seleccionan primos ℓ hasta encontrar uno, llamado $\ell_{\text{máx}}$, tal que

$$\prod_{2<\ell\leq\ell_{\text{máx}}} \ell > 4\sqrt{q}.$$

Con los valores de t mód ℓ para cada uno de estos primos, el Teorema Chino del Resto (CRT) permite recuperar t de forma única.

Para lograr esto, se utiliza el endomorfismo de Frobenius $\varphi : E(\mathbf{F}_q) \to E(\mathbf{F}_q)$, definido por $(x, y) \mapsto (x^q, y^q)$ y $\varphi(O) = O$. Para cualquier $P \in E(\mathbf{F}_q)$ se cumple:

$$\varphi^2(P) - [t]\varphi(P) + [q]P = O.$$

Al trabajar sobre $E[\ell]$, que recordemos es el subgrupo de ℓ-torsión de E descrito en la ecuación 2.6, y considerar t y q módulo ℓ, es posible determinar t mód ℓ sin tener que conocer explícitamente el punto P. Esto se logra mediante el uso de polinomios de división[7] y cálculos simbólicos en $\mathbf{F}_q[x]/(f_\ell(x))$, evitando así el trabajo directo en extensiones de grado elevado.

Aunque la complejidad asintótica puede parecer alta, en la práctica el algoritmo de Schoof resulta factible para valores de q relevantes en criptografía, sobre todo gracias a estrategias de optimización en la multiplicación de polinomios y la elección inteligente de polinomios de división $f_\ell(x)$.

2.8.1. Esquema del algoritmo de Schoof

A continuación se presenta un esquema simplificado de los pasos clave del algoritmo de Schoof:

1. **Cálculo inicial de t mód 2:** En característica impar, verificar la irreducibilidad de $X^3 + aX + b$ sobre $\mathbf{F}_q$ determina si $\#E(\mathbf{F}_q) \equiv 1 \pmod 2$. En característica 2 (no supersingular), se tiene directamente $t \equiv 1 \pmod 2$.

2. **Determinación de t mód ℓ para primos $\ell > 2$:** Seleccionar un primo ℓ y considerar la ecuación inducida por el endomorfismo de Frobenius:

   $$\varphi^2(P) - [t]\varphi(P) + [q]P = O.$$

   Trabajando en $E[\ell]$ y reduciendo t y q módulo ℓ, se busca un r tal que:

   $$\varphi^2(P) + [q]P = \pm[r]\varphi(P).$$

   Este procedimiento, apoyado en operaciones sobre polinomios en $\mathbf{F}_q[x]/(f_\ell(x))$, determina t mód ℓ sin calcular directamente P. Este es un procedimiento complejo. Se pueden ver más detalles tanto del procedimiento como de los polinomios de división en los Capitulos III y VII de Blake[7].

3. **Repetición con primos crecientes:** Incrementar ℓ a través de los primos, repitiendo el proceso anterior. Se continúa hasta que el producto de los primos considerados supere $4\sqrt{q}$.

4. **Reconstrucción de t:** Una vez determinados los valores de t mód ℓ_i para un conjunto de primos distintos $\ell_1, \ell_2, \ldots, \ell_n$, se utiliza el Teorema Chino del Resto (CRT) para recuperar t de manera única, considerando que los productos de los primos cubren un rango que excede $4\sqrt{q}$.

   Supongamos que se han obtenido las congruencias:

   $$t \equiv t_1 \pmod{\ell_1}, \quad t \equiv t_2 \pmod{\ell_2}, \quad \ldots, \quad t \equiv t_n \pmod{\ell_n}.$$

   El procedimiento es el siguiente:

   a) **Formación del producto total:** Sea

   $$L = \ell_1 \cdot \ell_2 \cdot \ldots \cdot \ell_n.$$

   Éste será el módulo final respecto del cual se obtendrá una solución única para t.

   b) **Cálculo de módulos parciales:** Para cada i, definir

   $$L_i = \frac{L}{\ell_i}.$$

   Así, L_i es el producto de todos los primos excepto ℓ_i.

c) **Cálculo de los inversos multiplicativos:** Para cada i, se requiere un entero M_i tal que

$$L_i \cdot M_i \equiv 1 \pmod{\ell_i}.$$

El número M_i es el inverso multiplicativo de L_i módulo ℓ_i, que puede encontrarse mediante el algoritmo de Euclides extendido.

d) **Combinación lineal de los residuos:** Una vez que se han obtenido t_i, L_i y M_i para cada i, se construye t como:

$$t \equiv \sum_{i=1}^{n} t_i \cdot L_i \cdot M_i \pmod{L}.$$

La idea es que cada término $t_i \cdot L_i \cdot M_i$ está construido de forma que:

- Sea congruente a t_i módulo ℓ_i.
- Sea congruente a 0 módulo ℓ_j para $j \neq i$.

Así, cuando se suman todos los términos, el resultado final es congruente a t_i módulo ℓ_i para cada i, cumpliendo todas las congruencias a la vez.

Esto funciona gracias a la construcción de L_i y M_i:

- $L_i \cdot M_i \equiv 1 \pmod{\ell_i}$ lo que asegura que el término $t_i \cdot L_i \cdot M_i$ reproduzca el residuo t_i en la congruencia asociada a ℓ_i.
- Para cualquier otro ℓ_j, como $\ell_j \mid L_i$ (cuando $j \neq i$), el término $t_i \cdot L_i \cdot M_i$ será múltiplo de ℓ_j, quedando congruente a 0 módulo ℓ_j.

De este modo, la suma final es la única solución que satisface simultáneamente todas las congruencias, completando así el paso de la combinación de los residuos.

5. **Cálculo del orden del grupo:** Finalmente, se obtiene el orden del grupo elíptico

$$\#E(\mathbf{F}_q) = q + 1 - t.$$

2.9. El problema del logaritmo discreto en curvas elípticas (ECDLP)

Hemos visto que la seguridad de la criptografía de curva elíptica se fundamenta en el *Elliptic Curve Discrete Logarithm Problem* (ECDLP), que es una variación del problema del logaritmo discreto clásico. Recordemos, según la sección 1.4, que el problema del logaritmo discreto (DLP) se formula del siguiente modo:

> Dados un generador g de un grupo finito G, un número primo p y un elemento $h \in G$, el problema consiste en encontrar un entero x tal que
>
> $$g^x \equiv h \pmod{p}.$$

De forma análoga, el *problema del logaritmo discreto en curvas elípticas* se plantea del siguiente modo:

Dado un punto P sobre una curva elíptica E definida sobre un cuerpo finito $\mathbb{F}_q$, y un múltiplo $Q = kP$, determinar el escalar $k \in \mathbb{Z}$ tal que

$$Q = kP.$$

Este problema es computacionalmente difícil cuando E, $\mathbb{F}_q$ y el punto generador P se eligen adecuadamente, y constituye la base de seguridad de numerosos protocolos criptográficos basados en curvas elípticas.

2.9.1. El algoritmo Index Calculus

Existen dos propiedades bien conocidas de todos los logaritmos que son aprovechadas por el método Index Calculus[28]:

$$\log_g(a \times b) = \log_g a + \log_g b \quad \text{y} \quad \log_g(a^e) = e \cdot \log_g a.$$

Estas equivalencias se utilizan para expresar el logaritmo de un elemento "suave"del grupo como una combinación lineal de los logaritmos de sus factores. Un número entero n se dice ***B*-suave** si todos sus factores primos son menores o iguales a un valor B.

Por ejemplo, supongamos que se sabe que $\log_g r = u$ y que la factorización de r es conocida, por ejemplo:

$$r = p_1^{e_1} \times \cdots \times p_k^{e_k}.$$

Entonces, tenemos la relación lineal

$$u = \log_g(r) = e_1 \log_g p_1 + \cdots + e_k \log_g p_k$$

Del álgebra lineal se sabe que, con suficientes relaciones como esta, el sistema de ecuaciones puede resolverse para los logaritmos desconocidos $\log_g p_i$.

2.9.1.1. Resumen del algoritmo

La idea es elegir una base de factores, que típicamente se selecciona como los primeros r números primos (para algún valor de r):

$$F = \{2, 3, 5, 7, \ldots, f_r\}.$$

Luego, si deseamos encontrar x tal que $g^x \equiv h \pmod{p}$, tomamos $k = 1, 2, \ldots$ y verificamos si podemos factorizar $g^k \pmod{p}$ como

$$2^{e_2} 3^{e_3} 5^{e_5} \cdots f_r^{e_r},$$

es decir, utilizando únicamente los primos en F. Para la mayoría de los valores de k, esto no funciona, porque $g^k \pmod{p}$ tendrá factores primos mayores que f_r. Sin embargo, ocasionalmente (y con suficiente frecuencia) sí funciona[24]. Hacemos una lista de los valores de k y la factorización de $g^k \pmod{p}$ en las ocasiones en que esto funciona.

Una vez que tenemos suficientes relaciones, podemos resolver un sistema de ecuaciones módulo $p-1$[24]. Así obtenemos, para cada $f \in F$, el logaritmo discreto $\log_g(f)$ de f, es decir, un número $\log_g(f)$ tal que $g^{\log_g(f)} \equiv f \pmod{p}$.

Una vez hecho esto, comenzamos de nuevo, intentando factorizar $g^m h \pmod p$ sobre la base de factores, para $m = 1, 2, \ldots$. Si encontramos una factorización de tal elemento para algún m, entonces $g^m h = f_1^{e_1} \times \cdots \times f_r^{e_r}$, entonces es fácil calcular el logaritmo discreto de h.

$$\log_g h = \log_g (g^{-m} f_1^{e_1} \times \cdots \times f_r^{e_r}) = -m + e_1 \log_g f_1 + \cdots + e_r \log_g f_r \pmod p$$

2.9.2. Comparación de complejidad

La diferencia más importante que hace brillar al ECDLP frente al tradicional DLP es su resistencia al ataque index calculus. Para ver la importancia de este algorítmo, vamos a comparar las complejidades de dos algoritmos que resuelven el DLP: el Index Calculus y el Baby-Step Giant-Step (BSGS).

El algoritmo de Index Calculus resuelve el problema del logaritmo discreto en el grupo multiplicativo $\mathbb{F}_p^*$ con una complejidad subexponencial expresada mediante la notación L, de la forma [20]:

$$L[p, 2, 1/2] = \exp\left((2 + o(1))(\log p)^{1/2}(\log\log p)^{1/2}\right),$$

donde p es el tamaño del grupo. Esta notación representa una clase de algoritmos cuyo tiempo de ejecución crece más rápido que cualquier potencia de $\log p$, pero más lento que p^ε para cualquier $\varepsilon > 0$, es decir, tienen complejidad subexponencial. Para ver más detalles consultar el anexo A.

Este algoritmo es altamente eficiente en $\mathbb{F}_p^*$. Sin embargo, no puede aplicarse de manera general al caso del ECDLP, ya que en las curvas elípticas no existe una noción análoga de factorización sobre una base eficiente de puntos. Esta resistencia al Index Calculus es una de las principales razones por las que la criptografía de curva elíptica ofrece una seguridad comparable con claves mucho más pequeñas.

El BSGS es un algorítmo que resuelve el problema del logaritmo discreto en cualquier grupo cíclico finito. Este algorítmo le veremos más adelante en detalle en la sección 4.1. La tabla 2.1 resume las diferencias de complejidad entre los dos algoritmos:

Algoritmo	**Complejidad**
Baby-Step Giant-Step (BSGS)	$O\left(e^{(1/2)\log(p)}\right) = O\left(\sqrt{p}\right)$
Index Calculus	$O\left(\exp\left(2 \cdot (\log p)^{1/2}(\log\log p)^{1/2}\right)\right)$

Cuadro 2.1: Comparación de complejidad entre el Baby-Step Giant-Step (BSGS) y el Index Calculus.

Aunque la complejidad $O(\sqrt{p})$ puede parecer polinomial cuando se expresa en función del número primo p, en realidad es exponencial respecto al tamaño de la entrada. Esto se debe a que en teoría de la complejidad se mide el coste computacional en función del número de bits necesarios para representar la entrada, es decir, $n = \log_2 p$. Por tanto, $\sqrt{p} = \sqrt{2^n} = 2^{n/2}$, lo cual implica que $O(\sqrt{p}) = O(2^{n/2})$ y, por consiguiente, la complejidad es exponencial en n.

Capítulo 3

Protocolos basados en curvas elípticas

3.1. Firma digital

La firma electrónica se ha convertido en una herramienta esencial para verificar la identidad del remitente y certificar que un mensaje no ha sido alterado durante su transmisión. Esta tecnología es utilizada en aplicaciones como contratos electrónicos, comunicaciones seguras y sistemas de autenticación en línea.

El algoritmo de firma digital (DSA)[22] era uno de los métodos más utilizados para implementar firmas electrónicas. Su funcionamiento se divide en tres fases principales: la generación de claves, donde se crean las claves pública y privada asociadas al firmante; la creación de la firma electrónica, que utiliza la clave privada para generar una firma única para cada mensaje; y la verificación de la firma, que permite a los receptores confirmar la autenticidad del mensaje utilizando la clave pública del firmante.

3.1.1. Algoritmo de Firma Digital (DSA)

El algorítmo DSA es un algorítmo clásico que ya se encuentra desfasado y sustituido por ECDSA. De todas formas nos sirve para introducir esta sección y familiarizarnos con los mecanismos de firma.

La generación de claves en DSA consta de dos fases[25]: la generación de parámetros globales y el cálculo de claves específicas para cada usuario. En la primera fase, se elige una función hash H, como SHA-2, y una longitud de clave L. Se selecciona un número primo q de N bits y otro primo p de L bits tal que $p-1$ sea múltiplo de q. También se elige un valor h en el rango $[2, p-2]$ para calcular $g = h^{(p-1)/q} \mod p$. Si $g = 1$, se elige un nuevo h. Los parámetros globales son (p,q,g). En la segunda fase, cada usuario selecciona una clave privada x en $[1, q-1]$ y calcula su clave pública $y = g^x \mod p$.

La firma de un mensaje m es un par (r,s) y designemos el hash del mensaje $H(m)$. Para calcular la firma se empieza con r, se selecciona un valor aleatorio k en $[1, q-1]$ y se calcula $r = (g^k \mod p) \mod q$. Si $r = 0$, se elige un nuevo k. Luego para s, se calcula $s = k^{-1}(H(m) + x \cdot r) \mod q$ donde k^{-1} es el inverso modular de k respecto a q. Si $s = 0$, se elige un nuevo k.

$$\begin{cases} r = (g^k \mod p) \mod q \\ s = k^{-1}(H(m) + x \cdot r) \mod q \end{cases}$$

Para verificar una firma (r,s) asociada a un mensaje m, primero se valida que r y s estén en el rango $[1, q-1]$. Luego, se calcula $w = s^{-1} \mod q$, seguido de los valores intermedios $u_1 = H(m) \cdot w \mod q$ y

$u_2 = r \cdot w \mod q$. Finalmente, se calcula $v = (g^{u_1} \cdot y^{u_2} \mod p) \mod q$. La firma es válida[31] si y solo si $v = r$:

$$v = (g^{u_1} \cdot y^{u_2} \mod p) \mod q$$

3.1.2. Firma electrónica con curvas elípticas (ECDSA)

El algoritmo ECDSA (*Elliptic Curve Digital Signature Algorithm*)[22] es una mejora de la versión del algoritmo DSA adaptada al contexto de curvas elípticas. Está estandarizado en FIPS 186-5 y se utiliza para la generación y verificación de firmas digitales. Al igual que DSA, su seguridad se basa en la dificultad del problema del logaritmo discreto, pero en este caso sobre grupos de puntos de una curva elíptica. Las claves generadas para ECDSA no deben utilizarse para otros fines (como intercambio de claves), con el objetivo de evitar ataques por reutilización.

Una variante del algoritmo, llamada ECDSA determinista, genera el número secreto por mensaje como una función del mensaje y de la clave privada, en lugar de seleccionarlo aleatoriamente. Esto elimina riesgos asociados a fuentes de aleatoriedad deficientes, especialmente en dispositivos con recursos limitados como tarjetas inteligentes o sistemas embebidos. Sin embargo, la implementación de ECDSA debe protegerse contra ataques de canal lateral o por fallos que puedan revelar material criptográfico sensible.

3.1.3. Parámetros de dominio

ECDSA requiere que las claves se generen respecto a un conjunto fijo de parámetros de dominio, que pueden ser compartidos y públicos. Estos parámetros son[22]:

$$(q, \text{FR}, h, n, \text{Type}, a, b, G, \{\text{domain\_parameter\_seed}\})$$

Aquí, q es el tamaño del cuerpo finito $\mathbb{F}_q$, y a y b son los coeficientes que definen la ecuación de la curva elíptica. El punto base $G = (x_G, y_G)$ es de orden primo n, y h es el cofactor, definido como el cociente entre el número total de puntos en la curva y n. Los elementos FR y Type indican detalles técnicos sobre la base del cuerpo y el modelo de curva. El valor opcional domain_parameter_seed puede incluirse si la curva fue generada a partir de una semilla verificable.

La generación de claves implica elegir una clave privada $d \in [1, n-1]$, y calcular la clave pública $Q = dG$. Estas claves deben asociarse con un **único conjunto de parámetros de dominio**. La seguridad de la firma requiere que tanto las claves como los parámetros estén protegidos contra modificaciones no autorizadas.

3.1.4. Generación del número secreto por mensaje

Antes de generar una firma digital, debe generarse un número secreto $k \in (0, n)$. Este valor debe mantenerse confidencial y **no reutilizarse**[22] bajo ninguna circunstancia, ya que su revelación compromete la clave privada.

Existen dos formas de generar k: aleatoriamente, usando un generador de bits aleatorios aprobado; o determinísticamente, como función del mensaje y la clave privada (según RFC 6979). Esta segunda opción es especialmente recomendable cuando no se dispone de buena entropía.

Una vez generado k, se computa su inverso módulo n, denotado $k^{-1} \mod n$, el cual se necesita para el cálculo de la firma.

3.1.5. Generación y verificación de firma digital

Generación: A partir del mensaje M, la clave privada d, los parámetros de dominio D, y un k generado como se indicó, se sigue el siguiente proceso:

1. Calcular $H = \text{Hash}(M)$.
2. Derivar un entero e desde H, truncando si es necesario.
3. Calcular el punto $R = [k]G = (x_R, y_R)$.
4. Definir $r = x_R \mod n$. Si $r = 0$, reiniciar.
5. Calcular $s = k^{-1}(e + r \cdot d) \mod n$. Si $s = 0$, reiniciar.
6. La firma es el par (r, s).

Verificación: Dados el mensaje M, la firma (r, s), la clave pública Q, y los parámetros D:

1. Verificar que $r, s \in [1, n-1]$.
2. Calcular $H = \text{Hash}(M)$ y derivar el entero e.
3. Calcular $w = s^{-1} \mod n$.
4. Calcular $u = ew \mod n$ y $v = rw \mod n$.
5. Calcular el punto $R_1 = [u]G + [v]Q$.
6. Si $R_1 = O$, rechazar.
7. Calcular $r' = x_{R_1} \mod n$.
8. Aceptar la firma si $r' = r$; en caso contrario, rechazar.

Tanto el proceso de generación como el de verificación deben emplear la misma función hash y los mismos parámetros de dominio. La resistencia criptográfica de ECDSA depende de la función hash usada y la longitud en bits de n.

3.1.6. Importancia de elegir k verdaderamente aleatorio

Un aspecto crucial tanto en DSA como en ECDSA es que el secreto efímero k debe ser verdaderamente aleatorio y único para cada firma. Si k se reutiliza al firmar dos mensajes distintos m y m', ambos firmados con la misma clave privada x, las firmas generadas serán (r, s) y (r', s'), respectivamente. Como r depende exclusivamente de k y el generador G, será igual en ambas firmas. Esto permite deducir k directamente, ya que $k = (e + xr)/s \mod q = (e' + xr)/s' \mod q$, donde $e = H(m)$ y $e' = H(m')$. A partir de esta relación, es posible despejar x como:

$$x = \frac{se' - s'e}{r(s - s')} \mod q.$$

Este ejemplo ilustra cómo la reutilización de k compromete directamente la clave privada x, lo que pone en riesgo la seguridad del sistema.

Además, Howgrave-Graham y Smart demostraron[16] que incluso una fuga parcial de k puede ser suficiente para recuperar la clave privada x. Este ataque, conocido como "lattice attack", explota información parcial de k obtenida a través de análisis por canal lateral o generadores de números aleatorios deficientes. Por ejemplo, si ciertos bits de k se filtran de manera repetida en múltiples firmas, es posible reconstruir x.

3.2. Intercambio de claves Diffie-Hellman

En el ámbito de la criptografía, uno de los desafíos fundamentales es establecer una clave compartida entre dos usuarios que necesitan comunicarse de manera segura, sin que un tercero pueda descubrirla, incluso si toda la comunicación ocurre en un canal público. Este problema resulta especialmente crítico cuando las partes no tienen la posibilidad de acordar previamente una clave en privado.

La solución a esta necesidad fue introducida con el primer criptosistema de clave pública, el intercambio de claves de Diffie-Hellman [13] en 1976. Este protocolo permite a los usuarios, como Alicia y Bob, generar una clave común a partir de valores públicos, asegurando que solo ellos puedan conocerla, sin importar que sus comunicaciones sean observadas. Este enfoque revolucionó el campo de la criptografía y estableció las bases para sistemas modernos de intercambio seguro de claves.

3.2.1. El problema Diffie-Hellman original (DH)

El protocolo Diffie-Hellman es un método para que dos partes generen una clave secreta compartida mediante un intercambio de información pública. Su funcionamiento se basa en la dificultad computacional del problema del logaritmo discreto en grupos cíclicos, lo que asegura su seguridad para valores suficientemente grandes.

Para comenzar, ambas partes acuerdan públicamente dos parámetros: un número primo p y un generador g de un grupo cíclico multiplicativo módulo p. Estos valores son públicos y accesibles tanto para las partes que desean comunicarse como para posibles interceptores.

Cada usuario selecciona de forma privada un número aleatorio como clave secreta. Por ejemplo, Alicia elige un número a, mientras que Bob elige b. A partir de sus claves privadas, calculan sus claves públicas. Alicia calcula $A = g^a \mod p$ y Bob calcula $B = g^b \mod p$. Estas claves públicas A y B se envían a través de un canal público.

Con las claves públicas intercambiadas, ambas partes pueden calcular de manera independiente una clave compartida. Alicia utiliza la clave pública de Bob (B) junto con su clave privada (a) para obtener $K_A = B^a \mod p$, mientras que Bob utiliza la clave pública de Alicia (A) junto con su clave privada (b) para obtener $K_B = A^b \mod p$. La razón por la que ambos llegan al mismo valor radica en la conmutatividad de la exponenciación modular.

$$\begin{cases} K_A = B^a = (g^b)^a = g^{ab} \mod p, \\ K_B = A^b = (g^a)^b = g^{ab} \mod p. \end{cases}$$

De este modo, tanto Alicia como Bob obtienen la misma clave secreta compartida $K = g^{ab} \mod p$, sin que un atacante pueda calcularla únicamente con los valores públicos.

La seguridad del protocolo radica en la dificultad de calcular el logaritmo discreto. Aunque un atacante puede interceptar los valores públicos p, g, A y B, no puede derivar la clave compartida K sin conocer las claves privadas a o b, lo que resulta computacionalmente inviable para valores grandes de p.

3.2.2. Relación entre el problema Diffie-Hellman y el problema del logaritmo discreto

El Problema de Diffie-Hellman (DHP) y el Problema del Logaritmo Discreto (DLP) están relacionados, pero no son equivalentes en términos de dificultad computacional. El DLP consiste en determinar el exponente x dado un generador g de un grupo cíclico G y un elemento $h = g^x \mod p$. Por otro lado, el DHP busca calcular $g^{ab} \mod p$ dado g^a y g^b, sin conocer a ni b.

La principal diferencia es que resolver el DLP permite resolver el DHP, ya que conocer a y b hace trivial el cálculo de g^{ab}. Sin embargo, el inverso no es necesariamente cierto: no se ha demostrado que resolver el DHP implique resolver el DLP. Esto deja abierta la posibilidad de que existan algoritmos eficientes para resolver el DHP sin necesidad de resolver el DLP, lo que indica que el DHP podría ser más fácil que el DLP en ciertos contextos.

En la práctica, ambos problemas suelen considerarse de dificultad similar en grupos criptográficamente seguros[15], como los grupos de puntos de curvas elípticas que vamos a ver ahora. Sin embargo, esta diferencia teórica es relevante para analizar la seguridad de protocolos como el intercambio de claves Diffie-Hellman, ya que la dificultad del DHP está garantizada solo si el DLP es efectivamente difícil en el grupo elegido.

3.2.3. Diffie-Hellman en curvas elípticas (ECDH)

El protocolo ECDH es una adaptación del protocolo Diffie-Hellman que emplea curvas elípticas para establecer una clave secreta compartida entre dos partes a través de un canal inseguro. Para comenzar, ambas partes acuerdan un conjunto de parámetros de dominio (K, E, q, h, G). Aquí, K es un cuerpo finito, E es una curva elíptica definida sobre K, G es un punto generador de orden q, y h es el cofactor, que satisface $\#E(K) = h \cdot q$.

El protocolo procede de la siguiente manera. Primero, Alice elige un escalar aleatorio a como su clave privada y calcula su clave pública efímera $[a]G$, donde $[a]G$ representa la multiplicación escalar del punto generador G por el escalar a. Alice envía $[a]G$ a Bob. De manera similar, Bob selecciona un escalar aleatorio b como su clave privada y calcula su clave pública efímera $[b]G$, que envía a Alice.

Una vez que Alice recibe $[b]G$, utiliza su clave privada a para calcular la clave secreta compartida como $K_A = [a]([b]G) = [ab]G$. De manera análoga, Bob utiliza $[a]G$ y su clave privada b para calcular $K_B = [b]([a]G) = [ab]G$. Como resultado, ambos obtienen la misma clave secreta $[ab]G$. La clave privada compartida entre Alice y Bob permite que ambos utilizen un cifrado simetrico para la comunicación posterior. El cifrado simétrico es mucho más eficiente para transmitir grandes volumenes de datos que la criptografia de clave pública

La seguridad del protocolo ECDH se basa en la dificultad del problema de Diffie-Hellman sobre curvas elípticas (ECDHP), que consiste en calcular $[ab]G$ a partir de las claves públicas $[a]G$ y $[b]G$. Este problema

es intratable sin resolver primero el problema del logaritmo discreto sobre curvas elípticas (ECDLP). Además, ECDH es eficiente en términos de ancho de banda, especialmente cuando se utiliza la compresión de puntos[18], lo que lo convierte en una alternativa atractiva frente al protocolo Diffie-Hellman basado en cuerpos finitos.

3.3. Curvas utilizadas en las implementaciones más comunes

En las diferentes implementaciones de protocolos criptográficos con curvas elípticas, una serie de curvas se han hecho las más populares debido a sus propiedades. A continuación vamos a hablar de tres familias diferentes que se han vuelto las favoritas para hacer firmas digitales, intercambio de claves y zero-knowledge proofs en redes blockchain.

3.3.1. Curvas de Edwards - Firmas digitales

Las curvas de Edwards son un tipo específico de curva elíptica definidas por la ecuación

$$x^2 + y^2 = 1 + dx^2y^2, \tag{3.1}$$

donde $d \neq 0$ es un parámetro que determina la forma de la curva. Estas curvas se definen sobre cuerpos finitos y son destacables por su eficiencia en operaciones criptográficas, particularmente en la suma y duplicación de puntos.

Una de las principales ventajas de las curvas de Edwards radica en sus fórmulas de suma y duplicación uniformes y simétricas[4]. A diferencia de otras representaciones de curvas elípticas, las operaciones de suma y duplicación de puntos en las curvas de Edwards pueden implementarse utilizando la misma fórmula, lo que simplifica la aritmética y reduce el riesgo de errores en la implementación. Por ejemplo, dados dos puntos $P = (x_1, y_1)$ y $Q = (x_2, y_2)$, su suma $R = (x_3, y_3)$ puede calcularse de manera eficiente sin necesidad de distinguir entre los casos de suma y duplicación:

$$(x_3, y_3) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right), \tag{3.2}$$

Otra propiedad crucial de las curvas de Edwards es su fuerte resistencia a los ataques de canal lateral[4]. Esta resistencia proviene de la uniformidad de sus fórmulas de suma, que garantiza que el comportamiento computacional no filtre información sobre las claves privadas a través de análisis de tiempo o consumo de energía. Además, el uso de coordenadas extendidas permite realizar operaciones sin inversiones costosas, lo que mejora aún más la eficiencia y la seguridad.

Estas propiedades hacen que las curvas de Edwards se utilicen ampliamente en esquemas de firmas digitales. Un ejemplo notable es el esquema de firmas Ed25519[6], cuyo nombre viene del cuerpo sobre el que está definido $\mathbb{F}_q$ con $q = 2^{255} - 19$ y utiliza una curva twisted Edwards. A diferencia de ECDSA, Ed25519 utiliza el algoritmo EdDSA, en el que el valor secreto por mensaje se genera de forma determinista como un hash de la clave privada y el mensaje. Esto elimina completamente la dependencia de una fuente de aleatoriedad segura, evitando vulnerabilidades asociadas a nonces débiles o reutilizados, que han comprometido la seguridad de muchas implementaciones de ECDSA. Además, las operaciones de EdDSA se realizan en coordenadas de Edwards con tiempo de ejecución constante, lo que proporciona

resistencia a ataques de canal lateral. También se beneficia de una verificación más eficiente, lo que lo hace especialmente adecuado para entornos de recursos limitados o aplicaciones de alto rendimiento. Estas ventajas combinadas han llevado a una adopción generalizada de Ed25519 en protocolos modernos donde la seguridad y la eficiencia son críticas.

3.3.2. Curvas de Montgomery - Intercambio de claves

Las curvas de Montgomery son una clase de curvas elípticas particularmente eficientes para la multiplicación escalar, lo que las hace ideales para aplicaciones criptográficas como el intercambio de claves. Estas curvas se definen por la ecuación[21]

$$By^2 = x^3 + Ax^2 + x, \tag{3.3}$$

donde A y B son parámetros de la curva, y $B \neq 0$.

Una de las principales fortalezas de las curvas de Montgomery es su eficiencia en la multiplicación escalar, que consiste en la suma repetida de un punto P en la curva. La multiplicación escalar, kP, es la base de la criptografía con curvas elípticas. Las curvas de Montgomery logran esta eficiencia mediante el uso del algoritmo de escalera de Montgomery, que calcula la multiplicación escalar de forma constante en el tiempo.

La escalera de Montgomery (Montgomery ladder)[5] es un algoritmo diseñado para realizar la multiplicación escalar kP, donde k es un escalar y P es un punto en la curva, de manera eficiente y segura. El algoritmo mantiene dos puntos, R_0 y R_1, inicialmente definidos como $R_0 = \mathcal{O}$ (el punto al infinito) y $R_1 = P$, respectivamente. Luego, procesa el escalar k bit a bit, de forma binaria, desde el bit más significativo hasta el menos significativo. El bit más significativo es aquel que representa un mayor valor, por ejemplo en 101 el primer 1 es el más significativo porque representa un valor en decimal de 4, mientras que el último 1 representa un valor en decimal de 1.

En cada paso, realiza exactamente una operación de duplicación de punto y una suma de puntos, independientemente del valor del bit actual. La actualización de los puntos se realiza de la siguiente manera

$$\begin{cases} R_0 = 2R_0, \quad R_1 = R_0 + P & \text{si el bit es 0,} \\ R_1 = 2R_1, \quad R_0 = R_0 + P & \text{si el bit es 1.} \end{cases} \tag{3.4}$$

Este enfoque garantiza que todas las iteraciones tomen el mismo tiempo, logrando una ejecución de tiempo constante, lo que protege contra ataques de canal lateral, como el análisis de tiempo o de consumo energético.

Gracias a su eficiente multiplicación escalar y su fuerte resistencia a los ataques de canal lateral, las curvas de Montgomery se utilizan ampliamente en protocolos de intercambio de claves. Un ejemplo destacado es Curve25519[2][3], que emplea una curva de Montgomery para el algoritmo de intercambio de claves X25519.

3.3.3. Curvas BLS12-381 - zk-SNARKs

Las curvas BLS12-381 son una familia de curvas elípticas de tipo Barreto-Lynn-Scott (BLS)[14], diseñadas para aplicaciones criptográficas como las pruebas de conocimiento cero (zk-SNARKs). Estas curvas están definidas sobre un cuerpo primo $\mathbb{F}_p$ y utilizan una extensión de cuerpo de grado 12, lo que permite que se

pueda definir un emparejamiento bilineal. La ecuación de la curva base está dada por:

$$y^2 = x^3 + 4, \tag{3.5}$$

definida sobre el cuerpo $\mathbb{F}_p$, donde p es un primo suficientemente grande.

Una característica clave de las curvas BLS12-381 es su capacidad para soportar emparejamientos bilineales, que son funciones matemáticas de la forma:

$$e : G_1 \times G_2 \to G_T, \tag{3.6}$$

donde G_1, G_2 y G_T son grupos cíclicos definidos sobre extensiones específicas del cuerpo $\mathbb{F}_p$. En el caso de las BLS12-381[14], G_1 está definido sobre $\mathbb{F}_p$, G_2 sobre una extensión cuadrática $\mathbb{F}_{p^2}$, y G_T es un subgrupo del grupo multiplicativo del cuerpo de extensión $\mathbb{F}_{p^{12}}$.

La utilidad de estos emparejamientos reside en su *bilinealidad*, es decir, la propiedad de que:

$$e(aP, bQ) = e(P, Q)^{ab},$$

para todos los $P \in G_1$, $Q \in G_2$ y $a, b \in \mathbb{Z}$. Esta propiedad permite construir primitivas criptográficas como firmas agregadas, pruebas de conocimiento cero no interactivas (zk-SNARKs), identificación basada en identidad, y esquemas de cifrado funcional, entre otras.

Las curvas BLS12-381 son ampliamente utilizadas en sistemas basados en zk-SNARKs, que permiten demostrar la validez de cálculos sin revelar los datos subyacentes. Este enfoque es esencial en aplicaciones que requieren privacidad y escalabilidad, como las criptomonedas (por ejemplo, Zcash[8]) y las soluciones de escalabilidad para blockchain, como los zk-rollups en Ethereum.

3.4. Requisitos de implementación y curvas del NIST

Antes de adentrarnos en los criterios técnicos que deben cumplir las curvas elípticas para su uso en criptografía, es importante introducir brevemente el papel del **NIST** en este ámbito.

El *National Institute of Standards and Technology* (NIST) es una agencia del gobierno de los Estados Unidos que desarrolla y promueve estándares tecnológicos, incluidos aquellos relacionados con la seguridad informática. En particular, el NIST ha tenido un papel central en la estandarización de algoritmos criptográficos ampliamente utilizados a nivel mundial, incluyendo aquellos basados en curvas elípticas.

En el contexto de la criptografía de curva elíptica, el NIST no solo propone algoritmos específicos, sino que también define criterios estrictos que deben cumplir las curvas utilizadas, con el fin de garantizar un nivel adecuado de seguridad frente a ataques criptográficos conocidos y facilitar implementaciones eficientes en distintas plataformas.

En esta sección, se presentarán los **requisitos criptográficos** que, según las recomendaciones del NIST, debe satisfacer una curva elíptica para ser considerada segura. Posteriormente, se describirán algunas de las curvas más utilizadas en la práctica, como `P-521` y `Curve25519`, analizando tanto sus propiedades matemáticas como las ventajas que ofrecen en términos de eficiencia computacional.

3.4.1. Requisitos criptográficos para la implementación de curvas elípticas

Toda curva elíptica utilizada en criptografía debe cumplir ciertos criterios fundamentales que garanticen su seguridad frente a ataques conocidos y aseguren su eficacia en aplicaciones prácticas. Según las

recomendaciones del NIST [10], las curvas recomendadas satisfacen los siguientes requisitos criptográficos generales:

1. **Cuerpo base.** El cuerpo finito subyacente $\mathbb{F}_q$ debe ser un cuerpo primo $\mathbb{F}_p$, o bien de característica dos $\mathbb{F}_{2^m}$, donde m es un número primo.
2. **Orden de la curva.** Cada curva E definida sobre $\mathbb{F}_q$ debe tener orden $\#E(\mathbb{F}_q) = h \cdot n$, donde:
   - n es un número primo grande,
   - h es el cofactor, un número pequeño tal que $\gcd(h,n) = 1$,
   - se exige que $h \leq 2^{10}$.
3. **Punto base.** Cada curva debe tener un punto base $G \in E(\mathbb{F}_q)$ de orden primo n, el cual se usa como generador del subgrupo criptográfico.
4. **Evitar curvas anómalas.** Para evitar ataques por transferencia aditiva (anomalous curve attack), se requiere que el orden de la curva no coincida con el tamaño del cuerpo: $\#E(\mathbb{F}_q) \neq q$.
5. **Alto grado de incrustación.** El problema del logaritmo discreto sobre curvas elípticas puede transferirse al cuerpo $\mathbb{F}_{q^k}$, donde k es el menor entero tal que $q^k \equiv 1 \mod n$ (llamado grado de incrustación). Para que esta transferencia no sea eficiente, se requiere que $k \geq 2^{10}$. Las curvas recomendadas por el NIST tienen grados de incrustación significativamente mayores.
6. **Discriminante del cuerpo de endomorfismos.** Sea t la traza de Frobenius de la curva E sobre $\mathbb{F}_q$. Se define el discriminante como $\text{Disc} = t^2 - 4q$, que se relaciona con el cuerpo de endomorfismos de E. El NIST no impone restricciones adicionales sobre el valor cuadrado libre de $|\text{Disc}|$, ya que no existe evidencia técnica que lo justifique en curvas no utilizadas para emparejamientos.

3.4.2. Curvas recomendadas por el NIST y consideraciones de implementación

Las curvas elípticas utilizadas en criptografía deben cumplir no solo requisitos de seguridad, sino también criterios prácticos de eficiencia. El NIST recomienda una serie de curvas, conocidas como las curvas `P-x`, diseñadas específicamente para facilitar implementaciones rápidas y seguras. Entre ellas destacan `P-256`, `P-384` y `P-521`. Además, en implementaciones modernas se incluye frecuentemente `Curve25519`, aunque no forme parte del estándar NIST, debido a su excelente balance entre seguridad y rendimiento.

Un aspecto clave en estas curvas es que están definidas sobre cuerpos primos cuya estructura permite una implementación eficiente de la aritmética modular. En general, el cálculo de $A \mod m$, donde $A < m^2$, implica una división entera costosa. Sin embargo, cuando el módulo m es un *número de Mersenne generalizado* o un *primo de Crandall*, esta operación puede sustituirse por una combinación de sumas y restas de unas pocas copias de m, gracias a identidades algebraicas particulares.

Por ejemplo, todos los primos utilizados en las curvas `P-x` tienen esta estructura especial. Esto permite implementar operaciones como la multiplicación y reducción módulo p sin divisiones, mejorando sustancialmente la eficiencia en hardware y software. Lo mismo ocurre con `Curve25519`, cuya forma del primo también permite simplificaciones notables.

En las siguientes subsecciones se estudiarán en detalle dos curvas ampliamente utilizadas en la práctica: `P-521`, como representante de las curvas estandarizadas por el NIST, y `Curve25519`, como ejemplo de curva optimizada para eficiencia y seguridad práctica.

Curva P-521

La curva `P-521` [10] es una de las curvas elípticas recomendadas por el NIST para aplicaciones criptográficas de alta seguridad. Está definida sobre el cuerpo primo $\mathbb{F}_p$, donde el módulo es:

$$p = 2^{521} - 1$$

Este número primo tiene una forma especial conocida como *número de Mersenne generalizado*, lo que permite realizar operaciones modulares de forma muy eficiente.

Dado un entero $A < p^2$, podemos descomponerlo como:

$$A = A_1 \cdot 2^{521} + A_0$$

donde A_0 y A_1 son enteros menores que 2^{521}. Es decir, la representación binaria de A puede considerarse como la concatenación de dos palabras de 521 bits:

$$A = (A_1 \parallel A_0)$$

Gracias a la forma especial del primo p, la reducción módulo p puede realizarse simplemente como:

$$B = (A_0 + A_1) \mod p$$

Esto evita divisiones enteras complejas, y permite implementar la reducción modular de manera más rápida y segura. Esta propiedad es especialmente útil para operaciones críticas como la multiplicación escalar, ampliamente utilizada en protocolos como ECDH y ECDSA.

Curve25519

La curva `Curve25519` es ampliamente utilizada en aplicaciones modernas de cifrado como X25519 (intercambio de claves) y Ed25519 (firmas digitales), gracias a su excelente equilibrio entre seguridad, eficiencia y facilidad de implementación. Está definida sobre el cuerpo primo $\mathbb{F}_p$, con un módulo de la forma:

$$p = 2^{255} - 19$$

Este primo es un ejemplo de *Crandall prime*, lo cual permite optimizaciones aritméticas similares a las de los números de Mersenne.

Cualquier entero $A < p^2$ puede expresarse como:

$$A = A_9 \cdot 2^{230} + A_8 \cdot 2^{204} + A_7 \cdot 2^{179} + A_6 \cdot 2^{153} + A_5 \cdot 2^{128} + A_4 \cdot 2^{102} + A_3 \cdot 2^{77} + A_2 \cdot 2^{51} + A_1 \cdot 2^{26} + A_0$$

donde cada A_i es un entero de 64 bits. En términos de palabras binarias, se puede representar como:

$$A = (A_9 \parallel A_8 \parallel \cdots \parallel A_0)$$

Gracias a esta estructura irregular pero cuidadosamente diseñada, la reducción módulo p puede realizarse mediante una combinación de desplazamientos, sumas y restas entre palabras de 64 bits, sin necesidad de operaciones de división. Esta característica, junto con la posibilidad de evitar ramificaciones condicionales en el código, facilita implementaciones resistentes a ataques por canal lateral.

Para más detalles sobre optimizaciones específicas, incluyendo técnicas basadas en instrucciones vectoriales (`NEON` en ARM), se puede consultar la literatura especializada citada en la bibliografía del NIST[10].

Tanto `P-521` como `Curve25519` ilustran cómo la elección del primo subyacente influye directamente en la eficiencia y seguridad de la implementación criptográfica, y por ello han sido seleccionadas como curvas recomendadas en aplicaciones de alta seguridad.

Capítulo 4

Algorítmos para resolver el ECDLP

Los ataques generales contra la criptografía basada en curvas elípticas buscan resolver el problema del logaritmo discreto en curvas elípticas sin depender de vulnerabilidades específicas de la curva utilizada. Entre los algoritmos más importantes se encuentran Pollard's Rho y Baby Step Giant Step. Aunque son algoritmos de tiempo exponencial, son los más rápidos que conocemos, siendo mucho más eficientes que la búsqueda exhaustiva.

Pollard's Rho resuelve $Q = xP$ generando una secuencia pseudoaleatoria de puntos $X_i = a_iP + b_iQ$, con coeficientes conocidos a_i, b_i. Cuando se detecta una colisión $X_i = X_j$ con $i \neq j$, se obtiene una ecuación $a_iP + b_iQ = a_jP + b_jQ$ de la que se puede despejar x.

Por otro lado, el algoritmo Baby Step Giant Step (BSGS) resuelve $Q = xP$ buscando una colisión entre dos conjuntos de puntos. Primero calcula los **baby steps** jP para $j = 0, 1, \ldots, m-1$, con $m = \lceil \sqrt{n} \rceil$. Luego recorre los **giant steps** $Q - imP$ hasta hallar una coincidencia, de la cual se deduce $x = im + j$.

Ambos métodos ofrecen estrategias distintas para abordar el problema del logaritmo discreto, pero se tienen una diferencia fundamental. Mientras que ambos tienen la misma complejidad computacional $O(\sqrt{n})$, el BSGS tiene una complejidad espacial de $O(\sqrt{n})$ y el Pollard's Rho $O(1)$.

4.1. Baby-Step Giant-Step (BSGS)

Antes de entrar en los detalles del algoritmo, conviene hacer una observación importante: cualquier entero n se puede escribir como $n = im + j$, donde i, j son enteros arbitrarios y m es un entero positivo. Por ejemplo, $47 = 4 \times 10 + 7$.

Aplicando esta idea, podemos reescribir el problema del logaritmo discreto $Q = kP$ como

$$Q = (im + j)P = imP + jP = i(mP) + jP.$$

El algoritmo Baby-Step Giant-Step (BSGS) es una estrategia de *"encuentro en el medio"*, que mejora significativamente la búsqueda forzada. En vez de calcular todos los valores kP para diferentes k hasta encontrar Q, el algoritmo calcula un número pequeño de valores para j ("baby steps") y otro número pequeño de valores para i ("giant steps"), buscando una coincidencia entre ellos.

El algoritmo funciona de la siguiente manera:

1. Calculamos $m = \lceil \sqrt{n} \rceil$, donde n es el orden del grupo.

2. Para cada j en $\{0, 1, \ldots, m-1\}$:
    - Calculamos jP y almacenamos el par (jP, j) en una tabla hash.
3. Calculamos mP.
4. Para cada i en $\{0, 1, \ldots, m-1\}$:
    - Calculamos $Q - i(mP)$.
    - Consultamos si este valor aparece en la tabla hash.
    - Si encontramos una coincidencia $Q - i(mP) = jP$, entonces hemos resuelto el problema, ya que $k = im + j$.

4.1.1. Intuición detrás del algoritmo

Inicialmente, realizamos pasos pequeños (baby steps) incrementando solo j, mientras almacenamos los resultados. Posteriormente, hacemos saltos grandes (giant steps) restando múltiples de mP de Q, buscando si alguno coincide con un baby step almacenado.

Para entender por qué este método es correcto, notemos que

- Cuando $i = 0$, comparamos Q con todos los valores jP.
- Cuando $i = 1$, comparamos $Q - mP$ con todos los valores jP.
- Y así sucesivamente hasta $i = m - 1$.

En resumen, el algoritmo explora todas las posibilidades para k usando a lo sumo $2m$ operaciones de suma/multiplicación, en vez de n como haría la fuerza bruta.

4.1.2. Complejidad

El algoritmo tiene una complejidad temporal y espacial de $O(\sqrt{n})$, dado que

- Realiza m baby steps y hasta m giant steps.
- Cada búsqueda en la tabla hash tiene coste $O(1)$.

Aunque sigue siendo un algoritmo de tiempo exponencial respecto al tamaño de entrada (el número de bits del orden de la curva), es una mejora considerable frente al ataque por fuerza bruta.

4.1.3. Consideraciones Prácticas

Para comprender mejor lo que implica la complejidad $O(\sqrt{n})$ en la práctica, consideremos una curva estándar como `prime192v1` (también conocida como `secp192r1` o `ansiX9p192r1`). Esta curva tiene un orden de aproximadamente[12]

$$n = 0xffffffff\,ffffffff\,ffffffff\,99def836\,146bc9b1\,b4d22831$$

La raíz cuadrada de n es aproximadamente $7{,}92 \times 10^{28}$.

Ahora bien, almacenar $7{,}92 \times 10^{28}$ puntos en una tabla hash sería inviable. Si cada punto ocupa 32 bytes, la tabla requeriría aproximadamente $2{,}5 \times 10^{30}$ bytes de memoria. Para ponerlo en perspectiva, la capacidad total de almacenamiento mundial está estimada en el orden de un zettabyte (10^{21} bytes)[1], casi diez órdenes de magnitud menos.

Incluso si cada punto ocupara solo 1 byte, seguiríamos estando muy lejos de poder almacenar todos ellos.

Este hecho resulta aún más impresionante si consideramos que `prime192v1` es una de las curvas con menor orden. Curvas más seguras, como `secp521r1`, tienen órdenes cercanos a $6{,}9 \times 10^{156}$.

Por tanto, aunque el algoritmo BSGS mejora la búsqueda por fuerza bruta, en la práctica sigue siendo inviable atacar curvas estándares con este método.

4.2. Pollard's Rho

El algoritmo *Pollard's Rho* es un método para calcular logaritmos discretos. Su complejidad temporal es asintóticamente igual a la del algoritmo *baby-step giant-step*, pero con una gran ventaja: su complejidad espacial es mucho menor, prácticamente constante. Esto lo convierte en una alternativa interesante cuando el uso de memoria es una limitación crítica.

4.2.1. Planteamiento del problema

Sea $E(\mathbb{F}_q)$ una curva elíptica definida sobre un cuerpo finito $\mathbb{F}_q$ y sea $P \in E(\mathbb{F}_q)$ un punto de orden n. Dado otro punto $Q \in \langle P \rangle$, el problema del logaritmo discreto en curvas elípticas consiste en encontrar un entero k tal que

$$Q = kP, \quad \text{con } k \in \mathbb{Z}_n.$$

4.2.2. La idea

El algoritmo de Pollard resuelve una versión ligeramente diferente[12]: dado un punto generador P y un punto Q, se buscan enteros a_1, b_1, a_2, b_2 tales que $a_1P + b_1Q = a_2P + b_2Q$.

Podemos usar la relación $Q = kP$ para obtener el valor de k.

$$\begin{aligned} & a_1P + b_1Q = a_2P + b_2Q \\ \Rightarrow \quad & a_1P + b_1kP = a_2P + b_2kP \\ \Rightarrow \quad & (a_1 - a_2)P = (b_2 - b_1)kP \end{aligned}$$

Como nuestro subgrupo es cíclico y de orden n, todos los coeficientes de la expresión están en módulo n.

$$(a_1 - a_2)P = (b_2 - b_1)kP \quad \Rightarrow \quad a_1 - a_2 \equiv (b_2 - b_1)k \mod n$$

Finalmente hemos obtenido el valor de k.

$$k \equiv \frac{a_1 - a_2}{b_2 - b_1} \mod n \tag{4.1}$$

siempre que $b_1 \not\equiv b_2 \mod n$ y el inverso de $b_2 - b_1$ módulo n exista.

4.2.3. El algoritmo

El algoritmo consiste en generar una secuencia pseudoaleatoria de puntos $X_0, X_1, X_2, \ldots$, todos pertenecientes al subgrupo generado por P, donde cada $X_i = a_i P + b_i Q$. Esta secuencia se define a través de una función pseudoaleatoria f, que depende del punto anterior

$$X_{i+1} = f(X_i)$$

La función f determina también cómo se actualizan los coeficientes a_i y b_i. Aunque el diseño exacto de f puede variar, lo importante es que sea determinista y que nos permita conocer los valores de a_i y b_i en cada paso.

Dado que el número de posibles puntos es finito, la secuencia eventualmente entra en un ciclo: existirán índices $i < j$ tales que $X_i = X_j$. Este comportamiento cíclico es lo que da nombre al algoritmo, ya que la trayectoria generada se asemeja a la letra griega ρ (rho).

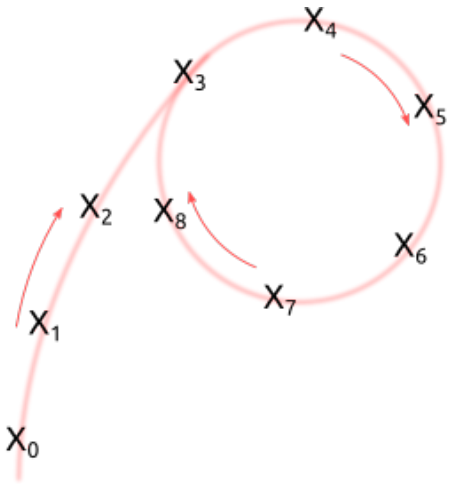


Figura 4.1: Muestra de como el camino pseudoaleatorio acaba convergiendo en un ciclo. Se empieza por los puntos X_0, X_1, X_2 y se acaba entrando en un ciclo con $X_3 = X_9$ [12]

Una vez detectado el ciclo, podemos usar las relaciones anteriores entre los coeficientes para resolver el logaritmo discreto. Ahora tenemos que lidiar con los problemas que conllevaría implementar este algoritmo, y el primero de ellos es que encontrar un ciclo de manera inocente es muy ineficiente.

4.2.4. Detección de colisiones

Primero vamos a construir una manera sencilla de detectar colisiones. El algoritmo se podría ver así:

1. Almacenamos el punto inicial
2. Generamos el siguiente punto, lo almacenamos y comparamos con el punto inicial
3. Generamos el tercer punto, lo almacenamos y comparamos con los anteriores
4. Repetimos el proceso hasta que el enésimo punto coincida con uno de los anteriores. Entonces habremos detectado una colisión.

Vamos a ver la escala del problema. Una de las curvas elípticas más grandes resueltas hasta ahora es una Barreto–Naehrig Pairing friendly sobre un cuerpo de orden 2^{114}.

Tendríamos que almacenar todos los puntos que se vayan calculando desde el punto inicial hasta que coincida con alguno de los X_i. Cada punto contiene dos coordenadas de 114 bits = 14 bytes aproximadamente, por lo tanto tenemos que por punto se necesitan 28 bytes. Por la paradoja del cumpleaños no necesitamos almacenar todos los elementos del grupo, en la mayoría de casos bastaría con 2^{57} elementos. Esto corresponde a un total de $2^{57} * 28$ bytes. Para ponerlo en perspectiva 2^{60} bytes es aproximadamente un exabyte y este es mil veces inferior que todo el contenido en internet en 2021[1]. Esto es inabarcable incluso para las organizaciones más grandes.

4.2.5. Algoritmo de Floyd para encontrar ciclos

El algoritmo anterior tiene una complejidad espacial y temporal de $O(\sqrt{n})$. La limitación la encontramos en el almacenamiento, pues no podemos almacenar 2^{60} bytes. Pero no necesitamos hacerlo.

El algoritmo de Floyd, también conocido como el algoritmo de la tortuga y la libre, reduce la complejidad espacial a $O(1)$ manteniendo la misma complejidad temporal. Se basa en la observación de que, si la secuencia generada por la función iterativa contiene un ciclo, entonces existe un entero λ que representa la longitud del bucle, y un índice μ que señala el comienzo del ciclo. En ese caso, para todo $i \geq \mu$, se cumple que $X_i = X_{i+k\lambda}$ para algún $k \geq 0$. En particular, si $i = k\lambda$, entonces $X_i = X_{2i}$, lo que permite detectar la presencia de un ciclo simplemente comparando X_i con X_{2i}.

El algoritmo de Floyd mantiene dos punteros: la tortuga en X_i y la liebre en X_{2i}. En cada paso, ambos avanzan (uno con paso 1 y otro con paso 2) y se comparan sus valores. La primera colisión $X_i = X_{2i}$ indica la detección de una repetición en la secuencia, y se produce en un índice ν múltiplo de la longitud λ del ciclo.

$$\begin{cases} \text{Tortuga:} & X_1,\ X_2,\ X_3,\ X_4,\ \ldots \\ \text{Liebre:} & X_2,\ X_4,\ X_6,\ X_8,\ \ldots \end{cases}$$

Una vez encontrada esta colisión, se reinicia uno de los punteros al comienzo de la secuencia y ambos avanzan con el mismo paso hasta encontrarse nuevamente. Este segundo encuentro ocurre exactamente en la posición μ, el primer índice del ciclo. Conocido μ, basta recorrer desde allí hasta que la secuencia se

repita para obtener λ. En el contexto del logaritmo discreto, este ciclo representa una colisión algebraica aprovechable para resolver el problema.

En nuestro caso para el algoritmo de Pollard's rho estamos interesados en hallar los coeficientes dados en la ecuación 4.1 que determinan la clave privada. Al llegar a una colisión, se llega al mismo punto aunque con pares de coeficientes distintos. O lo que es lo mismo, la tortuga encontrará un punto $X_i = a_i P + b_i Q$, mientras que la liebre alcanzará el mismo punto $X_j = a_j P + b_j Q$, con $i \neq j$ y

$$a_i P + b_i Q = a_j P + b_j Q$$

De esta manera, ya podríamos reconstruir la clave privada con los coeficientes encontrados.

4.2.6. Complejidad

Este algoritmo tiene la ventaja de requerir memoria constante, es decir, una complejidad espacial de $O(1)$, ya que solo se necesitan almacenar dos puntos en cada momento.

Determinar la complejidad temporal asintótica es más delicado, pero se puede construir una demostración probabilística que muestra que el número esperado de pasos antes de una colisión es $O(\sqrt{n})$[30], como se había anticipado. Esta conclusión se basa en la llamada paradoja del cumpleaños, que describe la probabilidad de que dos personas compartan fecha de nacimiento. En nuestro contexto, el interés está en la probabilidad de que dos combinaciones de coeficientes (a, b) generen el mismo punto sobre la curva.

Capítulo 5

Implementación práctica

El objetivo de este capítulo es poner en práctica todos los conceptos e ideas que hemos ido viendo a lo largo del trabajo. Como siempre suele ocurrir el paso de la teoría a la práctica no es tan fácil como puede parecer, por eso en este capítulo tratamos de acercar estos dos mundos creando un criptosistema completo. Este va a consistir en tres partes principales, la primera de ellas trata sobre la generación de curvas seguras y eficientes, la segunda sobre la materialización del protocolo de intercambio de claves ECDH y finalmente atacaremos, mediante los algoritmos BSGS (sec. 4.1) y Pollard's rho (sec. 4.2), el protocolo ECDH como si fuéramos atacantes externos. Esto nos permite tener una visión integral viendo desde la construcción de un sistema criptografico real que podría ser operativo hasta las implicaciones de seguridad de emplear claves pequeñas.

La implementación del criptosistema se ha llevado a cabo íntegramente en SageMath, aprovechando su potente entorno para el cálculo simbólico y numérico sobre curvas elípticas. Para ello se ha configurado un cuaderno (notebook) de SageMath que integra funciones nativas de Python y componentes específicos de la librería de curvas elípticas de SageMath, facilitando así la definición de parámetros del sistema, la generación de claves, y la ejecución de los algoritmos de cifrado y descifrado. Cada función empleada en el código —tanto las propias de Python (como estructuras de datos y operaciones aritméticas de alta precisión) como las provistas por SageMath (por ejemplo, la construcción de puntos sobre la curva, operaciones de grupo y generación de curvas seguras)— está debidamente documentada y anotada con comentarios que explican su propósito y su relación con la teoría presentada en capítulos anteriores.

El código fuente está organizado en módulos independientes para mejorar la legibilidad y la mantenibilidad: un módulo gestiona la inicialización de la curva y la generación de parámetros, otro se encarga del protocolo criptográfico ECDH, y un tercero se emplea para realizar los ataques al ECDLP. En ellos, se han incluido ejemplos de uso en forma de scripts autónomos y celdas interactivas que ilustran paso a paso cada uno de los protocolos basados en curvas elípticas estudiados. Todo el desarrollo está sometido a control de versiones mediante Git, lo que garantiza la trazabilidad de los cambios y facilita la colaboración y la reproducibilidad de los experimentos.

El repositorio con el código completo y las guías de ejecución se encuentra disponible públicamente en [9] De este modo, el lector puede reproducir íntegramente los resultados y adaptar el criptosistema a sus propios entornos de prueba.

5.1. Análisis detallado del generador de curvas

Para garantizar que la curva elíptica resultante sea a la vez segura y eficiente, es imprescindible controlar cuidadosamente cada fase de su generación. En primer lugar, se selecciona un primo p de tamaño adecuado (habitualmente entre 256 y 521 bits) que ofrezca resistencia ante ataques de fuerza bruta y métodos subexponenciales. A continuación, se muestrean los parámetros (a,b) evitando curvas singulares (discriminante $\Delta = 4a^3 + 27b^2 \neq 0$) y se verifica la cota de Hasse para asegurar que el número de puntos $\#E(\mathbb{F}_p)$ quede próximo a $p+1$. Seguidamente, se emplea el método SEA (Schoof–Elkies–Atkin) para calcular con rigor la cardinalidad de la curva y se descartan órdenes compuestas mediante pruebas rápidas y definitivas de primalidad. Finalmente, se selecciona un punto generador de orden primo y se registran métricas de rendimiento (tiempos de generación, número de ejecuciones de SEA, etc.) para calibrar y optimizar el algoritmo según las necesidades prácticas de la aplicación criptográfica.

En esta sección realizamos un recorrido pormenorizado por todas las piezas que conforman el script `generate_prime_order_curve.sage`, con el fin de entender cómo cada bloque de código contribuye a la construcción de una curva elíptica $E/\mathbb{F}_p$ de orden primo. Comenzaremos señalando las dependencias y módulos iniciales necesarios en SageMath, para luego inspeccionar las comprobaciones de seguridad que garantizan la no singularidad y la primalidad del grupo de puntos. A continuación, describiremos el bucle principal de generación, desde el muestreo del primo p y los parámetros (a,b) hasta la ejecución del SEA y las pruebas de primalidad, y finalizaremos con la selección de un punto generador y el manejo de posibles fallos tras agotar los intentos permitidos.

5.1.1. Entorno e importaciones

Antes de definir cualquier función, cargamos en SageMath los módulos necesarios para aritmética de campos finitos, curvas elípticas y control de tiempo. Estas importaciones permiten construir curvas elípticas sobre cuerpos finitos, generar parámetros aleatorios y medir tiempos de ejecución de forma precisa, sin necesidad de librerías externas, ya que SageMath incorpora todas ellas nativamente

```
from sage.all import (
    ZZ, sqrt, randint, random_prime, Integer,
    EllipticCurve, GF, walltime
)
```

Listing 5.1: Importaciones iniciales en SageMath

Aquí:

- `random_prime` y `randint` permiten muestrear primos y enteros aleatorios.
- `EllipticCurve` e `GF` construyen la curva sobre $\mathbb{F}_p$.
- `ZZ` y `sqrt` facilitan los cálculos de Hasse.
- `walltime` mide tiempos de ejecución para benchmarking.

5.1.2. Verificaciones de seguridad

Estas comprobaciones son esenciales para descartar curvas degeneradas o criptográficamente inseguras, y garantizar que el grupo generado es de orden primo, lo que impide ataques basados en la factorización del orden del grupo. Para garantizar esto, definimos la rutina `safety_checks`:

```
def safety_checks(E, p, n):
    # 1. Cota de Hasse
    t = p + 1 - n
    assert abs(t) <= 2*sqrt(p), "Nos aseguramos de que no viola la cota de Hasse"
    # 2. #E es primo
    assert ZZ(n).is_prime(), "Nos aseguramos de que el orden de E sea primo"
```

Listing 5.2: Sanity checks: Cota de Hasse y comprobación de orden primo

Podemos expresar estás comprobaciones con lenguaje matemático:

$$t = p+1-\#E(\mathbb{F}_p), \quad |t| \leq 2\sqrt{p} \quad \text{y} \quad \#E(\mathbb{F}_p) \text{ primo.}$$

5.1.3. Función principal de generación

La función `generate_prime_order_curve` implementa el procedimiento completo para construir una curva elíptica $E/\mathbb{F}_p$ cuyo grupo de puntos tenga orden primo. El enfoque es probabilístico: se generan múltiples curvas aleatorias hasta encontrar una que satisfaga las propiedades requeridas. El parámetro `k` determina la cantidad de bits del primo p, mientras que `max_trials` impone un límite superior al número de intentos permitidos. Las variables `sea_runs` y `candidates` se emplean para medir la eficiencia del proceso, registrando, respectivamente, cuántas veces se ha ejecutado el algoritmo SEA y cuántas curvas no singulares se han considerado.

Cada iteración del bucle realiza los siguientes pasos: (1) se muestrea un primo aleatorio $p \in [2^{k-1}, 2^k)$; (2) se seleccionan coeficientes $a, b \in \mathbb{F}_p$ al azar y se verifica que la curva $y^2 = x^3 + ax + b$ sea no singular; (3) se calcula el orden del grupo de puntos mediante el algoritmo SEA, y (4) se comprueba si este orden es primo mediante una prueba de primalidad rápida. Si todos los filtros se superan, se aplican comprobaciones matemáticas finales (`safety_checks`) y se devuelve la curva junto con un punto generador aleatorio.

Este procedimiento, si bien no garantiza éxito inmediato, es eficiente en la práctica: al trabajar con primos grandes y coeficientes aleatorios, la probabilidad de obtener una curva de orden primo es estadísticamente significativa. Además, el uso de pruebas rápidas permite descartar candidatos no válidos sin un coste computacional elevado. En caso de que no se encuentre una curva válida tras agotar los intentos permitidos, se lanza una excepción para indicar el fallo. La rutina `generate_prime_order_curve` engloba el proceso completo:

```
def generate_prime_order_curve(k, max_trials=10_000, log_every=50):
    sea_runs = 0
    candidates = 0
    for _ in range(max_trials):
        ...
```

Listing 5.3: Firma y variables iniciales

5.1.3.1. Selección de un primo de k bits

```
p = random_prime(2**k, lbound=2**(k-1))
F = GF(p)
```

Listing 5.4: Muestreo de p aleatorio

Aquí p es un primo uniforme en $[2^{k-1}, 2^k)$. A continuación trabajamos en el cuerpo finito $F = \mathbb{F}_p$.

5.1.3.2. Muestreo de parámetros (a, b)

```
a = randint(0, p-1)
b = randint(0, p-1)
# skip singular curves
if (4*a**3 + 27*b**2) % p == 0:
            continue
candidates += 1
E = EllipticCurve(F, [a, b])
```

Listing 5.5: Generación aleatoria de a, b

Se descartan automáticamente las curvas singulares (discriminante cero) al construir E.

5.1.3.3. Cálculo del orden con Schoof–Elkies–Atkin

SEA es un algoritmo eficiente para calcular el número de puntos sobre curvas elípticas definidas sobre $\mathbb{F}_p$, combinando los métodos de Schoof, Elkies y Atkin. El algoritmo original de Schoof se basa en calcular el traza de Frobenius $t = p + 1 - \#E(\mathbb{F}_p)$ módulo varios primos pequeños ℓ, y luego reconstruir t mediante el teorema chino del resto. Las mejoras introducidas por Elkies y Atkin permiten clasificar estos primos en dos tipos, optimizando el cálculo: los primos Elkies permiten reducir el problema a cálculos más simples mediante isogenias, mientras que los primos Atkin se tratan de forma distinta cuando no existe tal simplificación. Estas mejoras reducen significativamente la complejidad práctica del algoritmo, permitiendo calcular la cardinalidad de curvas sobre campos primos de tamaño criptográfico en tiempos razonables.

```
sea_runs += 1
n = E.cardinality()
```

Listing 5.6: Invocación al SEA para $\#E$

Cada llamada a `cardinality()` ejecuta el método SEA (Algoritmo Schoof–Elkies–Atkin) y se cuenta el número de ejecuciones del algoritmo en `sea_runs`.

5.1.3.4. Filtrado rápido de primalidad

```
if not n.is_prime(proof=False):
    continue
```

Listing 5.7: Prueba rápida de primalidad de $\#E$

La opción `proof=False` utiliza tests de primalidad probabilísticos, como el de Miller–Rabin, que permiten descartar rápidamente órdenes compuestos sin necesidad de realizar una prueba determinista completa.

5.1.3.5. Pruebas matemáticas completas

```
    safety_checks (E, p, n)
```

Listing 5.8: Llamada a `safety_checks`

Aquí aplicamos la función `safety_checks` descrita más arriba para verificar la cota de Hasse y la primalidad definitiva de $\#E$.

5.1.3.6. Retorno de (E, G)

```
    G = E.random_point()
    return E, G
```

Listing 5.9: Punto generador y retorno

Se elige un punto generador $G \in E(\mathbb{F}_p)$ al azar y se devuelve junto con la curva. Como el orden de la curva es primo, cualquier punto no nulo generado aleatoriamente tiene orden máximo, por lo que es un generador válido del grupo $E(\mathbb{F}_p)$.

5.1.3.7. Manejo de fallo

Si se agotan los `max_trials` sin éxito, se lanza excepción:

```
    raise RuntimeError(f"No prime-order curve found")
```

5.1.4. Gráficas de las curvas generadas para 32, 40 y 48 bits

Con el objetivo de ilustrar visualmente el resultado del generador `generate_prime_order_curve`, se muestran a continuación las gráficas de los primeros 100 puntos de curvas elípticas generadas sobre cuerpos finitos de 32, 40 y 48 bits. Sean elegido cuerpos finitos suficientemente pequeños para que sea posible resolverlos en un tiempo viable mediante los ataques presentados en el capítulo 4.

Para cada caso, se ha seleccionado un primo aleatorio p dentro del rango correspondiente, y se han buscado parámetros (a, b) tales que la curva $E : y^2 = x^3 + ax + b$ mód p cumpla las condiciones de seguridad descritas anteriormente: no singularidad, orden primo, y cumplimiento de la cota de Hasse. Una vez construida la curva, se ha elegido un punto generador aleatorio $G \in E(\mathbb{F}_p)$, y se ha registrado el número de candidatos evaluados, las ejecuciones del algoritmo SEA, así como el tiempo total de generación. Las gráficas representan los puntos $(x, y) \in E(\mathbb{F}_p)$ en coordenadas afines.

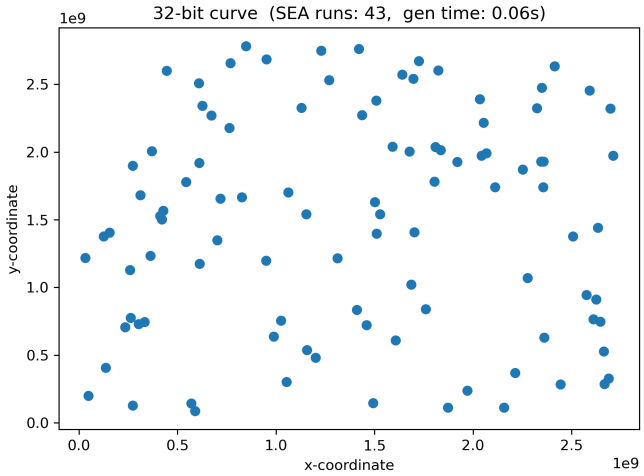


Figura 5.1: Curva elíptica generada sobre el cuerpo finito $p = 2\,786\,035\,199$ con 43 ejecuciones de SEA y 43 candidatos evaluados. La ecuación de la curva es $y^2 = x^3 + 51\,016\,516\,x + 2\,185\,363\,664$ mód p, con $\#E = 2\,786\,064\,251$ puntos. El generador G tiene coordenadas homogéneas $(672\,793\,040 : 2\,269\,103\,569 : 1)$, y el proceso de generación tomó 0,06 s.

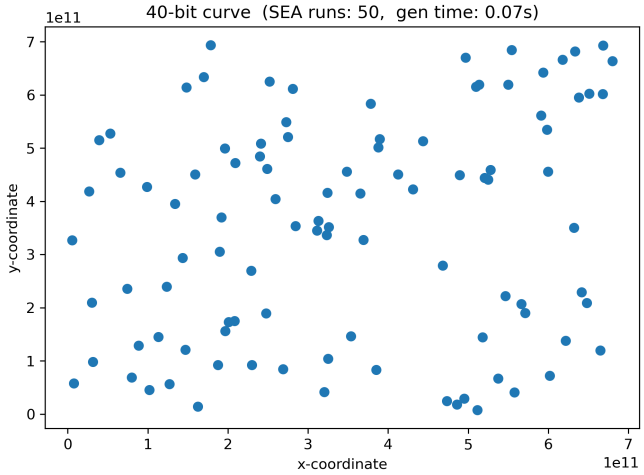


Figura 5.2: Curva elíptica generada sobre el cuerpo finito $p = 696\,676\,849\,667$ con 50 ejecuciones de SEA y 50 candidatos evaluados. La ecuación de la curva es $y^2 = x^3 + 117\,510\,864\,672\,x + 608\,566\,364\,394$ mód p, con $\#E = 696\,677\,404\,117$ puntos. El generador G tiene coordenadas homogéneas $(74\,126\,815\,521 : 235\,474\,348\,420 : 1)$, y el tiempo de generación fue de 0,07 s.

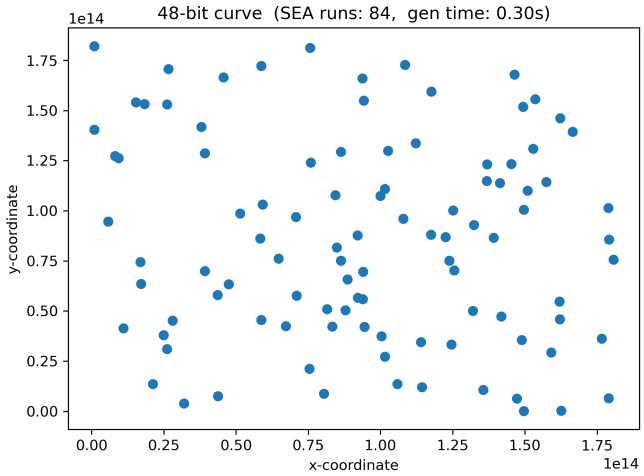


Figura 5.3: Curva elíptica generada sobre el cuerpo finito $p = 183\,697\,331\,409\,577$ con 84 ejecuciones de SEA y 84 candidatos evaluados. La ecuación de la curva es $y^2 = x^3 + 179\,954\,138\,900\,895\,x + 45\,849\,564\,335\,211 \pmod{p}$, con $\#E = 183\,697\,308\,090\,533$ puntos. El generador G tiene coordenadas homogéneas $(132\,020\,972\,057\,916 : 50\,122\,718\,111\,061 : 1)$ y el tiempo de generación fue de 0,30 s.

5.2. Implementación de un protocolo criptográfico

Este capítulo muestra cómo implementar paso a paso un protocolo criptográfico basado en curvas elípticas, con especial énfasis en el intercambio de claves ECDH (Elliptic Curve Diffie–Hellman), usado en versiones modernas de TLS (como la 1.3, especificada en el RFC 8446 [23]) para establecer claves simétricas de sesión de forma segura. A diferencia de protocolos clásicos como RSA, ECDH ofrece mayor seguridad y eficiencia a igual tamaño de clave, lo que lo convierte en la opción preferente en contextos modernos de cifrado como HTTPS.

5.2.1. Implementación del intercambio ECDH

Cabe destacar que, en esta implementación, las curvas $E/\mathbb{F}_p$ usadas han sido generadas aleatoriamente para propósitos educativos. En sistemas reales, se emplean curvas estandarizadas (como `secp256r1` o `Curve25519`) que han sido auditadas y optimizadas para garantizar tanto seguridad como interoperabilidad. Además, aunque en este ejemplo el secreto compartido se representa como un punto $S = abG \in E(\mathbb{F}_p)$, en la práctica suele utilizarse solo una de sus coordenadas (normalmente la coordenada x) para derivar claves simétricas, como se muestra en la siguiente sección.

El siguiente bloque de código realiza una implementación práctica del protocolo ECDH (*Elliptic Curve Diffie-Hellman*) sobre un conjunto de curvas previamente generadas. Para cada curva $E/\mathbb{F}_p$, se simula el intercambio de claves entre dos partes: `Alice` y `Bob`.

```
ecdh_data = []
for k, (E, G) in curves.items():
    n = E.cardinality()
```

```
# Claves privadas
alice_priv = randint (1, n-1)
bob_priv   = randint (1, n-1)

# Claves publicas
alice_pub = alice_priv * G
bob_pub   = bob_priv   * G

# Secreto compartido
alice_secret = alice_priv * bob_pub
bob_secret   = bob_priv   * alice_pub
assert alice_secret == bob_secret
```

Listing 5.10: Simulación del intercambio ECDH para cada curva.

Descripción del código:

- **Entrada:** un diccionario `curves` que asocia a cada tamaño de bits k una curva E y un generador G.
- **Cardinalidad del grupo:** se calcula una vez con `E.cardinality()`.
- **Claves privadas:** tanto `Alice` como `Bob` generan un número aleatorio $a, b \in [1, n-1]$, donde $n = \#E(\mathbb{F}_p)$.
- **Claves públicas:** se obtiene $A = aG$ y $B = bG$, que son compartidas públicamente.
- **Clave compartida:** cada parte multiplica su clave privada por la clave pública del otro, es decir

$$S_{\text{Alice}} = aB = abG\,, \quad S_{\text{Bob}} = bA = baG\,.$$

- **Verificación:** se asegura que ambos lados obtienen el mismo punto $S \in E(\mathbb{F}_p)$ mediante `assert`.

Este esquema refleja fielmente la lógica del protocolo ECDH, donde el secreto $S = abG$ nunca se transmite, y la seguridad se basa en la dificultad computacional del problema del logaritmo discreto elíptico (ECDLP).

5.2.2. Derivación de una clave simétrica desde el secreto ECDH

La derivación de una clave simétrica a partir del secreto compartido es un paso fundamental en cualquier protocolo de establecimiento de claves como ECDH. Aunque el intercambio asegura que ambas partes obtienen el mismo secreto $S = abG$, este se representa como un punto sobre la curva elíptica, cuya estructura no es directamente utilizable en algoritmos de cifrado. Aquí es donde entra en juego la criptografía simétrica: una vez transformado S en una clave binaria segura y de longitud fija (por ejemplo, 256 bits), esta puede emplearse con algoritmos como AES (Advanced Encryption Standard) para cifrar y descifrar mensajes de manera eficiente. A diferencia de la criptografía asimétrica, la simétrica permite procesar grandes volúmenes de datos a alta velocidad, lo que la hace ideal para la transmisión segura de información una vez establecida la clave mediante ECDH.

Para ello, es necesario aplicar un proceso de transformación que convierta el punto S en una clave adecuada para su uso simétrico. Este paso es esencial porque:

- El punto S vive en el grupo aditivo de la curva elíptica $E(\mathbb{F}_p)$, y sus coordenadas no tienen estructura ni longitud apropiada para ser usadas directamente como clave binaria.
- La derivación mediante una función hash criptográfica (como SHA-256) garantiza una distribución uniforme, resistencia a colisiones y compatibilidad con estándares como AES-256.

Esta transformación se lleva a cabo mediante una función de derivación de clave (*Key Derivation Function*, KDF). En este caso se utiliza `SHA-256`, una función ampliamente adoptada en criptografía moderna, que asegura que la clave resultante tenga buena entropía y no revele información sobre el secreto original. Existen variantes más complejas como HKDF (HMAC-based Key Derivation Function), que ofrecen mayor flexibilidad en contextos más exigentes.

El siguiente bloque de código convierte el valor del secreto S —concretamente su coordenada x— en una clave binaria de 256 bits aplicando SHA-256:

```
def derive_aes_key(shared_secret: int) -> bytes:
    """
    Derive a 256-bit AES key from the ECDH shared secret integer
    by hashing it with SHA-256.
    """
    # Ensure it's a Python integer
    if isinstance(shared_secret, list):
        shared_secret = shared_secret[0]
    # Convert secret to big-endian bytes (at least 1 byte)
    length = max(1, (shared_secret.bit_length() + 7) // 8)
    b = shared_secret.to_bytes(length, 'big')
    # Derive 32-byte key
    return hashlib.sha256(b).digest()
```

Listing 5.11: Derivación de clave AES desde un secreto ECDH

Descripción:

- Se asegura que el valor recibido (`shared_secret`) sea un entero de Python.
- Si proviene como una lista (por ejemplo, coordenadas $[x, y]$), se toma únicamente x.
- Se convierte a bytes en orden big-endian, con la longitud mínima necesaria.
- Se aplica `SHA-256` al valor serializado para obtener una clave de 256 bits.

Una vez definida esta función, se aplica a los resultados de ECDH previamente almacenados en un archivo JSON

```
# 1. Leer resultados ECDH
with open('ecdh_results.json', 'r') as f:
    ecdh_data = json.load(f)
```

```
# 2. Derivar AES key para cada curva
derived = []
for entry in ecdh_data:
    k = entry['bit_size']
    shared = entry['shared_secret']
    key = derive_aes_key(shared)
    derived.append({
        'bit_size':      k,
        'alice_priv':    entry['alice_priv'],
        'bob_priv':      entry['bob_priv'],
        'shared_secret': shared,
        'aes_key_hex':   key.hex()
    })
```

Listing 5.12: Cálculo de claves AES desde los secretos ECDH guardados

Resultado: Para cada curva de tamaño k bits, se almacena la clave derivada en hexadecimal. Estas claves pueden utilizarse directamente en cifrados como AES-GCM para asegurar la confidencialidad de los datos intercambiados.

5.2.3. Cifrado simétrico con AES-CBC

¿Y por qué hablar de AES cuando el TFG es sobre curvas elípticas? La respuesta está en el papel complementario que desempeña la criptografía simétrica dentro de los protocolos modernos. Las curvas elípticas se utilizan para resolver de forma segura el problema del intercambio de claves, como hemos visto con ECDH. Sin embargo, una vez que dos partes comparten una clave secreta, cifrar directamente con curvas elípticas sería ineficiente y poco práctico para grandes volúmenes de datos. En cambio, se recurre a algoritmos simétricos como AES (Advanced Encryption Standard), mucho más rápidos y optimizados para el cifrado masivo de información. Esta sección ilustra cómo la clave compartida obtenida mediante curvas elípticas se utiliza en la práctica para cifrar mensajes mediante AES, completando así el flujo real de un protocolo criptográfico como TLS.

Una vez derivada una clave simétrica segura mediante el protocolo ECDH y una función de hash como SHA-256, es posible cifrar mensajes con algoritmos simétricos modernos. En este trabajo se utiliza el algoritmo **AES** (*Advanced Encryption Standard*), un estándar ampliamente adoptado por su seguridad y eficiencia.

El modo elegido es **CBC** (*Cipher Block Chaining*), que requiere un vector de inicialización (`IV`) aleatorio y un esquema de *padding* para asegurar que el texto tenga longitud múltiplo del bloque (16 bytes). El padding utilizado es PKCS#7.

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes

# 1. Leer la clave derivada para la curva de 32 bits
with open('derived_keys.json', 'r') as f:
```

```
    derived = json.load(f)

# Buscar la entrada de 32 bits
key_hex = next(entry['aes_key_hex'] for entry in derived if entry['bit_size'] == 32)
key = bytes.fromhex(key_hex)

# 2. Mensaje a cifrar: cita histrica de Carl Friedrich Gauss
plaintext = (
    "Mathematics■is■the■queen■of■the■sciences■and■number■theory■is■the■queen■of■mathematics.
        ■Carl■Friedrich■Gauss"
).encode('utf-8')

# 3. Cifrar con AES-256-CBC y PKCS#7 padding
iv = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
```

Listing 5.13: Cifrado de una cita con AES-256-CBC usando la clave derivada

Descripción del proceso:

- Se carga la clave derivada correspondiente a la curva de 32 bits desde un archivo `JSON`.
- Se codifica el texto original en UTF-8.
- Se genera un `IV` aleatorio de 16 bytes.
- Se construye un objeto de cifrado AES en modo CBC.
- Se aplica relleno (`pad`) al texto para que su longitud sea múltiplo del bloque.
- Finalmente, se cifra el texto en bloque con la clave y el `IV`.

El siguiente fragmento mostrará la cita original junto con su versión cifrada en `Base64`

Texto original:

```
Mathematics is the queen of the sciences and number theory is the
queen of mathematics. Carl Friedrich Gauss
```

Texto cifrado (Base64):

```
IV (Base64): 89PuzxknoQXGUEHByJ9wlg==
Ciphertext (Base64):
TUQzcBuCTpRmYzX3ZKOax3+eUnN6Knuv8xXeLZ9vDuoYwLtU
gFK7Fp1jzUt8OI8u/NMhnFcLTMEyS7F2ue/ODqLSwlfV2xC/
rbG9hHOdOOc/sHi0RWI8nSS17VaTLPvmv74IVzC+zQR659R0
3363jg==
```

5.2.4. Descifrado con AES-CBC

Una vez cifrada la información mediante una clave simétrica derivada, el último paso en la transferencia segura de datos es su descifrado en el destinatario. Este procedimiento es fundamental en protocolos como HTTPS: el navegador y el servidor establecen una clave compartida mediante un intercambio de claves (por ejemplo, ECDH), y utilizan esa clave para cifrar y descifrar toda la comunicación subsecuente.

En el cifrado por bloques, como AES en modo CBC (*Cipher Block Chaining*), se requiere un vector de inicialización (IV) adicionalmente a la clave. El IV no necesita ser secreto, pero sí debe ser único por sesión y transmitirse junto con el texto cifrado. Por ello, durante el cifrado se obtienen dos salidas codificadas en Base64

- El **IV** en Base64.
- El **Ciphertext** (texto cifrado) en Base64.

Ambos elementos son necesarios para que el receptor pueda reconstruir correctamente el mensaje original.

El siguiente fragmento de código realiza el proceso de descifrado:

```
from Crypto.Util.Padding import unpad

# 3. Decrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted = cipher.decrypt( ciphertext )

# 4. Output the result
print("Decrypted■text:", decrypted.decode('utf-8'))
```

Listing 5.14: Descifrado de un mensaje cifrado con AES-256-CBC

Descripción de los pasos:

- Se decodifica el texto cifrado desde Base64 para obtener los datos binarios.
- Se inicializa un objeto `AES` en modo CBC, reutilizando la clave secreta derivada y el vector de inicialización original.
- Se descifra el contenido binario, obteniendo el mensaje original en bytes.
- Finalmente, se decodifica a texto UTF-8 para su presentación en forma legible.

El resultado del proceso de descifrado es el siguiente

Texto descifrado:

```
Mathematics is the queen of the sciences and number theory is the queen of
mathematics. Carl Friedrich Gauss
```

5.3. Resolviendo el ECDLP

Una vez generadas curvas elípticas, es fundamental estudiar la resistencia de estas curvas frente a ataques que intentan resolver el **Problema del Logaritmo Discreto en Curvas Elípticas (ECDLP)**. Recordamos que este problema consiste en, dado un punto generador $G \in E(\mathbb{F}_p)$ y otro punto $B = kG$, recuperar el entero secreto k. Este problema es la base de la seguridad de múltiples esquemas criptográficos modernos como ECDH, ECDSA o ECIES; su dificultad computacional es la que impide a un adversario obtener la clave privada a partir de información pública.

Cómo hemos visto, existen distintos algoritmos genéricos que pueden aplicarse para intentar resolver el ECDLP. En esta sección se comparan dos de los más conocidos descritos en el capítulo 4:

- **Pollard's Rho:** algoritmo probabilístico con complejidad esperada $O(\sqrt{n})$, baja memoria y alta eficiencia práctica. Ideal para curvas grandes.
- **Baby-Step Giant-Step (BSGS):** algoritmo determinista con la misma complejidad teórica $O(\sqrt{n})$, pero requiere memoria proporcional a la raíz cuadrada del orden del grupo, lo cual lo hace menos práctico para curvas grandes.

En entornos distribuidos, el algoritmo Baby-Step Giant-Step (BSGS) ofrece una ventaja significativa en términos de paralelización, ya que permite dividir de forma trivial el espacio de búsqueda entre distintos procesos o nodos: cada uno puede calcular y comparar subconjuntos de pasos gigantes de manera independiente, sin necesidad de comunicación entre ellos. En cambio, la versión clásica de Pollard's Rho depende de una caminata pseudoaleatoria cuyos ciclos deben detectarse globalmente, lo cual complica su paralelización. Aunque existen variantes paralelas de Pollard's Rho (como el método de rho distinguido o el enfoque de múltiples caminatas con sincronización periódica), estas requieren mecanismos de coordinación más sofisticados para detectar colisiones compartidas, lo que puede afectar la escalabilidad y el rendimiento efectivo en sistemas distribuidos.

Ambos algoritmos se van a aplicar sobre curvas generadas de 32, 34, 36, 38, 40, 42 y 44 bits, y se van a comparar en cuanto al tiempo necesario para recuperar la clave secreta k usada en el punto $B = kG$.

```
from sage.groups.generic import discrete_log
import time

DL_TIMES = {}

for bits in EXTRA_BITS:
    if bits not in PUBLIC:
        continue

    E, G, B = PUBLIC[bits]["E"], PUBLIC[bits]["G"], PUBLIC[bits]["B"]
    n       = CURVES[str(bits)]["n"]

    # Pollards
    t0      = time.time()
    k_rho   = discrete_log(B, G, n, operation='+', algorithm='rho')
```

```
    rho_t   = time.time() - t0

    # Baby-Step Giant-Step
    t0      = time.time()
    k_bsgs  = discrete_log(B, G, n, operation='+', algorithm='bsgs')
    bsgs_t  = time.time() - t0

    # Registro de resultados
    DL_TIMES[bits] = {"rho": rho_t, "bsgs": bsgs_t}
    print(f"{bits}-bit    :{rho_t:.4f}s    BSGS:{bsgs_t:.4f}s")

print("\nSecret keys recovered and timings recorded for newly generated curves.")
```

Listing 5.15: Comparación entre Pollard's Rho y BSGS para resolver el ECDLP

Descripción del código:

- Para cada curva de tamaño definido en `EXTRA_BITS`, se recupera el generador G, el punto $B = kG$, y el orden n.
- Se aplica `discrete_log` de SageMath utilizando dos algoritmos distintos: `rho"` y `"bsgs"`.
- Se mide el tiempo de ejecución para cada método con `time.time()` y se almacena en un diccionario llamado `DL_TIMES`.
- Finalmente, se imprime el resultado indicando los segundos que tardó cada técnica en recuperar la clave secreta.

La siguiente gráfica muestra visualmente los resultados del benchmark:

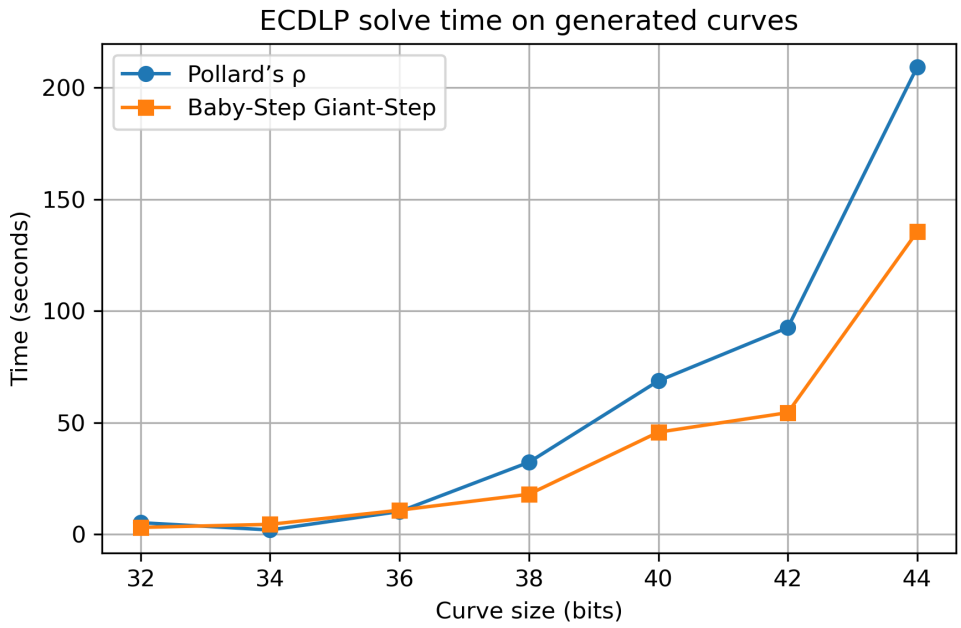


Figura 5.4: Tiempo de resolución del ECDLP con Pollard's Rho y BSGS para diferentes ordenes de la curva.

La figura 5.4 muestra cómo varía el tiempo necesario para resolver el problema del logaritmo discreto en curvas elípticas (ECDLP) en función del tamaño del cuerpo, comparando los algoritmos Pollard's Rho y Baby-Step Giant-Step (BSGS). En todos los casos, ambos algoritmos muestran un crecimiento exponencial del tiempo conforme aumenta el número de bits del grupo, como era de esperar dada su complejidad temporal$O(\sqrt{p})$. Sin embargo, se observan diferencias importantes en la eficiencia práctica. Para curvas de hasta 36 bits, ambos métodos presentan tiempos similares, pero a partir de los 38 bits, Pollard's Rho comienza a superar significativamente a BSGS en tiempo de ejecución. Esta diferencia se acentúa con curvas más grandes, alcanzando en el caso de 44 bits más de 200 segundos frente a los poco más de 100 segundos de BSGS. Esta aparente ventaja de BSGS desaparece según subes a tamaños del cuerpo más elevados. A partir de cierto punto, BSGS se vuelve inviable debido a su complejidad espacial de $O(\sqrt{p})$, mientras que la de Pollard's Rho es constante $O(1)$. Ya en 44 bits se estaban alcanzando los límites de memoria del portatil con el que se han realizado las simulaciones, y la demanda de memoria crece exponencialmente con el número de bits. De este modo podemos llegar a una conclusión, para tamaños de cuerpo para los que es viable ejecutar BSGS, este es más rápido que Pollard's Rho. En cambio, para cuerpos suficientemente grandes BSGS se vuelve inviable y el mejor algoritmo para resolver el ECDLP es Pollard's Rho.

Apéndice A

Teoría de la complejidad

A.1. Notación Big-O

El análisis de algoritmos y su eficiencia requiere una forma precisa de describir cómo crece el tiempo de ejecución (o el uso de memoria) en función del tamaño de la entrada. La notación *Big-O* cumple este papel al proporcionar una cota superior para el comportamiento de una función a medida que el tamaño de la entrada tiende a infinito.

Definición A.1.1 (Notación Big-O)**.** Sean $f(n)$ y $g(n)$ funciones definidas para todo $n \geq n_0$, con valores positivos. Decimos que:

$$f(n) = O(g(n))$$

si existe una constante $C > 0$ tal que:

$$f(n) \leq C \cdot g(n), \quad \text{para todo } n \geq n_0$$

En palabras simples, esto significa que $f(n)$ crece a lo sumo tan rápido como una constante por $g(n)$, cuando n es suficientemente grande. A pesar de la notación con signo igual, $O(g(n))$ representa una clase de funciones y debe interpretarse como una cota superior más que como una igualdad.

Cuando utilizamos esta notación, lo que nos interesa es el comportamiento asintótico de la función, es decir, cómo crece cuando n es muy grande. Por ejemplo, si $f(n) = 2n^2 + 3n - 3$, diremos que $f(n) = O(n^2)$, porque existe una constante (por ejemplo, $C = 3$) tal que $f(n) \leq C \cdot n^2$ para todo n suficientemente grande.

La notación Big-O ignora constantes multiplicativas y términos menores, centrándose en el orden de crecimiento más significativo. Esto permite comparar de forma simple la eficiencia de diferentes algoritmos. Si $f(n)$ es un polinomio de grado d, con coeficiente principal positivo, entonces $f(n) = O(n^d)$, ya que el término dominante determina el crecimiento de la función.

Ejemplo A.1.2. Sea $f(n) = 5n^3 + 100n \log n + 1000$. El término dominante cuando $n \to \infty$ es $5n^3$, así que decimos que:

$$f(n) = O(n^3)$$

En términos de límites, si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$ con $L \in \mathbb{R}$, entonces $f(n) = O(g(n))$. Si ese límite es cero, también es correcto escribir $f(n) = O(g(n))$, aunque en ese caso decimos que $f(n) = o(g(n))$ (notación "little-o"),

es decir, $f(n)$ crece mucho más lentamente que $g(n)$. Si el cociente tiende a 1, entonces $f(n) \sim g(n)$, lo que significa que son asintóticamente equivalentes.

Esta notación también nos permite hacer estimaciones rápidas sobre el comportamiento de un algoritmo. Por ejemplo, si una función $f(n)$ es $O(n)$, entonces al duplicar el valor de n, el valor de $f(n)$ también se duplica aproximadamente. Si $f(n) = O(n^2)$, al duplicar n, el valor de $f(n)$ se cuadruplica. Si la función está acotada por $O(n^3)$, entonces duplicar n multiplicará $f(n)$ por un factor cercano a 8. Cuando se tiene $f(n) = O(2^n)$, basta aumentar n en una unidad para que $f(n)$ se duplique aproximadamente.

Finalmente, esta herramienta es fundamental para clasificar algoritmos según su eficiencia. Hablamos de algoritmos de tiempo lineal cuando $f(n) = O(n)$, cuadrático si es $O(n^2)$, cúbico si es $O(n^3)$, exponencial si es $O(2^n)$, y logarítmico si es $O(\log n)$, como en el caso de la búsqueda binaria. Esta clasificación permite anticipar la viabilidad práctica de un algoritmo en función del tamaño de los datos con los que trabajará.

A.1.1. Notaciones relacionadas: little-o, Omega y Theta

Además de la notación Big-O, existen otras notaciones asintóticas que ofrecen descripciones más precisas del comportamiento de una función en el infinito.

Definición A.1.3 (Notación little-o)**.** Decimos que $f(n) = o(g(n))$ si, para toda constante $\varepsilon > 0$, existe n_0 tal que:

$$f(n) < \varepsilon \cdot g(n), \quad \text{para todo } n \geq n_0.$$

Esto significa que $f(n)$ crece estrictamente más lento que $g(n)$. En términos de límites:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Definición A.1.4 (Notación Omega)**.** Decimos que $f(n) = \Omega(g(n))$ si existe una constante $C > 0$ y un n_0 tal que:

$$f(n) \geq C \cdot g(n), \quad \text{para todo } n \geq n_0.$$

Esta notación proporciona una cota inferior del crecimiento de la función. Si $f(n) = \Omega(g(n))$, entonces $f(n)$ crece al menos tan rápido como $g(n)$.

Definición A.1.5 (Notación Theta)**.** Decimos que $f(n) = \Theta(g(n))$ si existen constantes $C_1, C_2 > 0$ y n_0 tal que:

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n), \quad \text{para todo } n \geq n_0.$$

En este caso, $f(n)$ y $g(n)$ crecen al mismo ritmo asintótico. La notación $\Theta(g(n))$ indica que $g(n)$ es una cota tanto superior como inferior para $f(n)$, hasta constantes.

Notación	Significado	Interpretación
$O(g(n))$	Cota superior asintótica	$f(n)$ crece como máximo tan rápido como $g(n)$. Existen constantes C, n_0 tales que $f(n) \leq Cg(n)$ para $n \geq n_0$.
$o(g(n))$	Cota superior estricta	$f(n)$ crece estrictamente más lento que $g(n)$. Para todo $\varepsilon > 0$, existe n_0 tal que $f(n) < \varepsilon g(n)$.
$\Omega(g(n))$	Cota inferior asintótica	$f(n)$ crece al menos tan rápido como $g(n)$. Existen constantes C, n_0 tales que $f(n) \geq Cg(n)$ para $n \geq n_0$.
$\Theta(g(n))$	Cota ajustada (superior e inferior)	$f(n)$ crece al mismo ritmo que $g(n)$. Existen constantes C_1, C_2, n_0 tales que $C_1 g(n) \leq f(n) \leq C_2 g(n)$.

Cuadro A.1: Comparación de notaciones asintóticas

Son herramientas fundamentales en el análisis de algoritmos, especialmente cuando queremos distinguir entre distintos niveles de eficiencia con mayor precisión.

A.2. Estimación de Tiempos

Como primer paso tenemos que encontrar alguna manera de estimar el tiempo que se necesita para ejecutar un algoritmo. Asumiremos que toda la aritmética se realiza en binario, es decir, con 0's y 1's.

A.2.1. Operaciones con Bits

Empecemos con un problema aritmético muy simple, la suma de dos enteros binarios[19]. Por ejemplo:

$$\begin{array}{r} 1111000 \\ +0011110 \\ \hline 10010110 \end{array}$$

Supongamos que ambos números tienen k bits; si uno de los enteros tiene menos bits que el otro, rellenamos con ceros a la izquierda, como en este ejemplo, para que tengan la misma longitud. Aunque este ejemplo involucra enteros pequeños (con $k = 7$), debemos pensar en k como un número muy grande, como 500 o 1000.

Analicemos en detalle lo que implica esta suma. Básicamente, debemos repetir los siguientes pasos k veces:

1. Mirar el bit superior e inferior y también si hay un acarreo por encima del bit superior.
2. Si ambos bits son 0 y no hay acarreo, escribir un 0 y continuar.
3. Si (a) ambos bits son 0 y hay acarreo, o (b) uno de los bits es 0, el otro es 1, y no hay acarreo, escribir un 1 y continuar.

4. Si (a) uno de los bits es 0, el otro es 1, y hay acarreo, o (b) ambos bits son 1 y no hay acarreo, escribir un 0, poner un acarreo en la siguiente columna y continuar.

5. Si ambos bits son 1 y hay acarreo, escribir un 1, poner un acarreo en la siguiente columna y continuar.

Realizar este procedimiento una vez se llama una *operación de bits*. Sumar dos números de k bits requiere k operaciones de bits. Veremos que tareas más complicadas también se pueden descomponer en operaciones de bits. La cantidad de tiempo que una computadora tarda en realizar una tarea es esencialmente proporcional al número de operaciones de bits.

Cuando hablamos de estimar el "tiempo"que lleva realizar algo, nos referimos a encontrar una estimación del número de operaciones de bits requeridas.

Por lo tanto, el tiempo requerido (es decir, el número de operaciones de bits) para sumar dos números es igual a la longitud máxima de los dos números (En bits). Escribimos:

$$\text{Tiempo}(k\text{-bit} + l\text{-bit}) = \text{máx}(k,l).$$

A.3. Algoritmos

En general, al estimar el número de operaciones de bits necesarias para realizar algo, el primer paso es decidir y escribir un esquema de un procedimiento detallado para llevar a cabo la tarea. Esto lo hicimos antes en el caso de nuestro problema de suma de dos números. Un procedimiento explícito paso a paso para realizar cálculos se llama un *algoritmo*.

Por supuesto, puede haber muchos algoritmos diferentes para hacer lo mismo. Se puede optar por usar el más fácil de escribir, o el más rápido que se conoce, o bien se puede elegir comprometerse y hacer un equilibrio entre simplicidad y velocidad.

Ejemplo Estimar el tiempo requerido para calcular $n!$.

Utilizamos el siguiente algoritmo. Primero multiplicamos 2 por 3, luego el resultado por 4, luego ese resultado por 5, y así sucesivamente hasta llegar a n. En el paso $j-1$, estamos multiplicando $j!$ por $j+1$. En total, hay $n-2$ multiplicaciones, donde cada multiplicación implica multiplicar un producto parcial (es decir, $j!$) por el siguiente entero. El producto parcial se volverá muy grande. Como una estimación en el peor de los casos para el número de bits que tiene, tomamos el número de dígitos binarios del último producto, es decir, $n!$, $\text{longitud}(n!) = O(n\ln n)$.

Por lo tanto, en cada una de las $n-2$ multiplicaciones en el cálculo de $n!$, estamos multiplicando un entero con a lo sumo $O(\ln n)$ bits (es decir, $j+1$) por un entero con $O(n\ln n)$ bits (es decir, $j!$). Esto requiere $O(n\ln^2 n)$ operaciones de bits. Debemos hacer esto $n-2 = O(n)$ veces. Así que el número total de operaciones de bits es $O(n\ln^2 n)\cdot O(n) = O(n^2\ln^2 n)$. Terminamos con la estimación:

$$\text{Tiempo}(n!) = O(n^2\ln^2 n).$$

A.3.1. De Tiempo Polinomial a Tiempo Exponencial

Ahora introducimos una definición fundamental en el estudio de los algoritmos.

Algoritmos de Tiempo Polinomial Un algoritmo para realizar un cálculo se dice que es un *algoritmo de tiempo polinomial* si existe un entero d tal que el número de operaciones de bits necesarias para ejecutar el algoritmo en enteros de longitud total a lo sumo k es $O(k^d)$.

Por ejemplo, las operaciones aritméticas usuales $(+,-,\times,\div)$ son ejemplos de algoritmos de tiempo polinomial, al igual que la conversión de una base a otra. Por otro lado, el cálculo de $n!$ no lo es. (Sin embargo, si uno se conforma con saber $n!$ solo con un cierto número de cifras significativas, por ejemplo, los primeros 1000 dígitos binarios, se puede obtener mediante un algoritmo de tiempo polinomial usando la fórmula de aproximación de Stirling para $n!$.)

Algoritmos de Tiempo Exponencial Una clase de algoritmos que están muy lejos de ser de tiempo polinomial es la clase de los algoritmos de tiempo exponencial. Estos tienen una estimación de tiempo de la forma $O(e^{ck})$, donde c es una constante. Aquí, k es la longitud binaria total de los enteros a los que se aplica el algoritmo.

A.3.2. Complejidad Subexponencial

En teoría de la complejidad, un algoritmo se considera *subexponencial* si su tiempo de ejecución crece más rápido que cualquier función polinomial, pero más lento que cualquier función exponencial pura. Para describir formalmente este comportamiento intermedio, se utiliza la siguiente notación[20]:

$$L[x,c,\alpha] = \exp\left((c+o(1))(\ln x)^{\alpha}(\ln\ln x)^{1-\alpha}\right),$$

donde:

- x es el tamaño del espacio de entrada,
- $c > 0$ es una constante dependiente del algoritmo,
- $0 < \alpha < 1$ determina el grado de subexponencialidad.

Esta notación es especialmente útil para analizar algoritmos relacionados con problemas aritméticos difíciles, como la factorización de enteros o el cálculo de logaritmos discretos.

Cabe observar que:

- Si $\alpha = 0$, entonces $L[x,c,0]$ es polinomial en $\ln x$, es decir, el algoritmo tiene complejidad polinomial.
- Si $\alpha = 1$, se recupera el caso exponencial clásico, ya que $L[x,c,1] = \exp((c+o(1))\ln x) = x^{c+o(1)}$.
- Para $0 < \alpha < 1$, el algoritmo es *estrictamente subexponencial*.

Un algoritmo subexponencial es asintóticamente más rápido (más lento respectivamente) que un algoritmo cuyo tiempo de ejecución es polinomial (exponencial respectivamente) sobre el tamaño de entrada.

Esta clase de algoritmos cobra gran relevancia en criptografía, ya que la existencia de un algoritmo subexponencial para un problema sobre el que se basa un sistema criptográfico suele implicar que dicho sistema es inseguro para tamaños de clave moderados. Por ejemplo, el algoritmo de *Index Calculus* resuelve el problema del logaritmo discreto en $\mathbb{F}_p^*$ con complejidad $L[p, 2, 1/2]$, lo que lo convierte en una de las mejores estrategias conocidas para ese entorno. Sin embargo, esta estrategia no se traslada de manera efectiva al contexto de curvas elípticas, donde no se conocen algoritmos subexponenciales generales.

A.4. P, NP y el cracking problem

Para poder definir las clases *P* y *NP*, primero debemos modificar nuestros problemas de manera que se conviertan en *problemas de decisión*. Un problema de decisión es un problema cuya solución consiste en una respuesta de ”sí.º ”no”. Por otro lado, si la salida deseada es más que un ”sí.º ”no”— es decir, si queremos encontrar un número, una ruta en un mapa, etc. — entonces llamamos al problema un *problema de búsqueda*.

Vamos a ver que no se pierde generalidad al trabajar con problemas de decisión en lugar de problemas de búsqueda, trabajando con el problema de factorización de enteros. Primero, supongamos que tenemos un algoritmo para resolver el problema de búsqueda. Esto significa que, dado N, podemos aplicar el algoritmo para encontrar un factor no trivial M, luego aplicarlo nuevamente para encontrar factores no triviales de M y N/M, y así sucesivamente, hasta que N se haya descompuesto como un producto de potencias primas. Una vez que tenemos la factorización prima de N, podemos determinar inmediatamente si N tiene un factor en el intervalo $[2, k]$. Es decir, la respuesta a esta pregunta es ”sí”si y solo si el divisor primo más pequeño de N está en ese intervalo.

Por el contrario, supongamos que tenemos un algoritmo para resolver el problema de decisión. En ese caso, podemos usar la búsqueda binaria para aproximarnos al valor exacto de un factor, resolviendo así el problema de búsqueda de la factorización entera.

A.4.1. Problemas de clase P

Un problema de decisión P está en la clase P de problemas de tiempo polinomial si existe un polinomio $p(n)$ y un algoritmo tal que, si una instancia de P tiene una longitud de entrada $\leq n$, entonces el algoritmo responde correctamente en tiempo $\leq p(n)$. Una definición equivalente es: Un problema de decisión P está en P si existe una constante c y un algoritmo tal que, si una instancia de P tiene longitud de entrada $\leq n$, entonces el algoritmo responde en tiempo $O(n^c)$.

Observación Esta caracterización mediante un polinomio intenta capturar una clase de problemas que en la práctica pueden resolverse rápidamente. No está claro a priori que P sea la clase correcta para este propósito. Por ejemplo, un algoritmo con tiempo de ejecución n^{100}, donde n es la longitud de la entrada, es más lento que uno con tiempo de ejecución $e^{0,0001n}$ hasta que n supera aproximadamente los diez millones, aunque el primer algoritmo es de tiempo polinomial y el segundo es de tiempo exponencial.

Sin embargo, la experiencia ha sido que si un problema de interés práctico está en P, entonces existe un algoritmo para resolverlo cuyo tiempo de ejecución está acotado por una potencia pequeña de la longitud de la entrada. A veces, un problema que está en P o que se cree que está en P tiene un algoritmo eficiente en la práctica que no es de tiempo polinomial.

A.4.2. Problemas de clase NP

Un problema de decisión P está en la clase NP si, dada cualquier instancia de P, una persona con poder de cómputo ilimitado no solo puede responder la pregunta, sino que, en el caso de que la respuesta sea ”sí”, puede proporcionar evidencia que otra persona podría utilizar para verificar la corrección de la respuesta en tiempo polinomial. Su demostración de que su respuesta ”sí.es correcta se llama un çertificado”(más precisamente, un certificado de tiempo polinomial).

A.4.3. El cracking problem

Supongamos que estamos intentando criptoanalizar un sistema de cifrado de clave pública. Es decir, conocemos la clave pública de cifrado E y la función inyectiva f_E del conjunto P de unidades de mensaje en texto plano al conjunto C de unidades de mensaje en texto cifrado. Interceptamos algún $y \in C$ y queremos determinar el único $x \in P$ tal que $f_E(x) = y$. Esto se conoce como el problema de romper un sistema de clave pública. Formalmente, el problema es el siguiente:

ENTRADA: E, $f_E : P \to C$, $y \in C$.

SALIDA: $x \in P$ tal que $f_E(x) = y$.

A diferencia de los problemas en la sección anterior, el criptoanalista sabe algo más que la entrada. Es decir, sabe que existe un $x \in P$ tal que $f_E(x) = y$ (en otras palabras, y está contenido en la imagen de la función) y, además, x es único. Por lo tanto, el problema de romper el sistema es de un tipo ligeramente diferente a nuestros ejemplos anteriores, y se necesita una nueva definición que capture esta situación.

Problema con promesa Es un problema de búsqueda o decisión con una condición adjunta. Al analizar un algoritmo para un problema con promesa, no consideramos qué ocurre cuando la condición no se cumple, es decir, no nos importa si el algoritmo da una respuesta incorrecta o no da respuesta en absoluto en tal caso.

Ejemplo La siguiente es una versión con promesa del problema de búsqueda de factorización de enteros:

ENTRADA: Un entero $N > 1$.

PROMESA: N es un producto de dos números primos distintos.

SALIDA: Un factor no trivial M de N.

Un algoritmo eficiente para este problema con promesa sería suficiente para romper el RSA. Por supuesto, en el improbable caso de que se encontrara un algoritmo eficiente que pudiera factorizar un producto de dos primos pero no un producto de tres primos, sería fácil modificar RSA para usar módulos que sean productos de tres primos.

Referencias

[1] Alvy. *¿Cuánto mide Internet? En bytes y en dimensiones físicas*. Microsiervos. 2021. URL: https://www.microsiervos.com/archivo/tecnologia/tamano-internet-bytes-dimensiones-fisicas.html.

[2] Bernstein, Daniel J. "Curve25519: New Diffie-Hellman Speed Records". En: *Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography (PKC 2006)*. 2006.

[3] Bernstein, Daniel J. *Things that use Curve25519*. URL: https://ianix.com/pub/curve25519-deployment.html.

[4] Bernstein, Daniel J. y Lange, Tanja. *Faster Addition and Doubling on Elliptic Curves*. https://eprint.iacr.org/2007/286.pdf. 2007.

[5] Bernstein, Daniel J. y Lange, Tanja. "Montgomery Curves and the Montgomery Ladder". En: *Cryptology ePrint Archive* (2017).

[6] Bernstein, Daniel J. et al. *High-speed high-security signatures*. https://eprint.iacr.org/2011/368. 2011.

[7] Blake, Ian F., Seroussi, Gadiel y Smart, Nigel P. *Elliptic Curves in Cryptography*. Vol. 265. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[8] Bowe, Sean. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. 2017. URL: https://electriccoin.co/blog/new-snark-curve/.

[9] Cardeñoso, Francisco. *Código para el TFG de criptografía con curvas elípticas*. 2025. URL: https://github.com/Fcocard/elliptic-curve-cryptography.

[10] Chen, Lily et al. *Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*. NIST Special Publication 800-186. This publication is available free of charge from the URL above. National Institute of Standards y Technology, 2023. DOI: 10.6028/NIST.SP.800-186. URL: https://doi.org/10.6028/NIST.SP.800-186.

[11] Cohen, Henri. *A Course in Computational Algebraic Number Theory*. Vol. 138. Graduate Texts in Mathematics. Berlin Heidelberg: Springer-Verlag, 1993. ISBN: 978-3-540-55640-0.

[12] Corbellini, Andrea. *Elliptic Curve Cryptography: Breaking Security and a Comparison with RSA*. 2015. URL: https://andrea.corbellini.name/2015/06/08/elliptic-curve-cryptography-breaking-security-and-a-comparison-with-rsa/.

[13] Diffie, W. y Hellman, M. "New directions in cryptography". En: *IEEE Transactions on Information Theory* 22.6 (1976), págs. 644-654. DOI: 10.1109/TIT.1976.1055638.

[14] Edgington, Ben. *BLS12-381 For The Rest Of Us*. 2023. URL: https://hackmd.io/@benjaminion/bls12-381#About-curve-BLS12-381.

[15] Fifield, David. "The equivalence of the computational Diffie–Hellman and discrete logarithm problems in certain groups". En: (ene. de 2012). URL: https://theory.stanford.edu/~dfreeman/cs259c-f11/finalpapers/dlp-cdh.pdf.

[16] Howgrave-Graham, N. A. y Smart, N. P. "Lattice Attacks on Digital Signature Schemes". En: *Designs, Codes and Cryptography* 23.3 (ago. de 2001), págs. 283-290. ISSN: 1573-7586. DOI: 10.1023/A:1011214926272. URL: https://doi.org/10.1023/A:1011214926272.

[17] International Organization for Standardization. *Information security, cybersecurity and privacy protection — Information security management systems*. Standard. Geneva, CH, 2022.

[18] Jivsov, Alexey. *Compact representation of an elliptic curve point*. 2014. URL: https://www.ietf.org/archive/id/draft-jivsov-ecc-compact-04.pdf.

[19] Koblitz, Neal. *Algebraic aspects of cryptography*. Springer, 1998.

[20] Menezes, Alfred J. *Elliptic Curve Public Key Cryptosystems*. Vol. SECS 234. The Kluwer International Series in Engineering and Computer Science. Foreword by Neal Koblitz. Boston, MA: Springer, 1993. ISBN: 978-1-4613-6403-0. DOI: 10.1007/978-1-4615-3198-2.

[21] Montgomery, Peter L. "Speeding the Pollard and Elliptic Curve Methods of Factorization". En: *Mathematics of Computation* 48.177 (1987), págs. 243-264.

[22] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. FIPS Publication 186-5. U.S. Department of Commerce, 2023. URL: https://csrc.nist.gov/publications/detail/fips/186/5/final.

[23] Rescorla, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3*. https://datatracker.ietf.org/doc/html/rfc8446. Ago. de 2018.

[24] Rubinstein-Salzedo, Simon. *Cryptography*. Springer, 2018.

[25] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd. John Wiley & Sons, 1996.

[26] Schoof, R. "Elliptic curves over finite fields and the computation of square roots mod p". En: *Mathematics of Computation* 44 (1985), págs. 483-494.

[27] Schoof, R. "Counting points on elliptic curves over finite fields". En: *Journal de Theorie des Nombres de Bordeaux* 7 (1995), págs. 219-254.

[28] Studholme, Chris. *The Discrete Log Problem*. Jun. de 2002. URL: https://www.cs.toronto.edu/~cvs/dlog/research_paper.pdf.

[29] Washington, Lawrence C. *Elliptic Curves: Number Theory and Cryptography*. 2.ª ed. Chapman & Hall/CRC, Taylor & Francis Group, 2008.

[30] Wikipedia contributors. *Pollard's rho algorithm for logarithms — Complexity*. 2024. URL: https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms#Complexity.

[31] Wikipedia contributors. *Digital Signature Algorithm*. URL: https://en.wikipedia.org/wiki/Digital_Signature_Algorithm.