Trabajo Teoría

Curva Elíptica

8 de diciembre de 2019

Sofía Almeida Bruno
Pedro Manuel Flores Crespo
María Victoria Granados Pozo

Índice

Secciones **Página**

1. Introducción

En el desarrollo de la criptografía, con el objetivo de poder mantener comunicaciones de forma privada, se pasó de sistemas más simples, como el *cifrado de César*, a sistemas más complejos, como el *cifrado de Vigenère*. Estos criptosistemas que eran de tipo simétrico tenían un problema, el intercambio de claves. Tratando de evitar este problema surgen, en la década de los 70, los criptosistemas asimétricos o de clave pública. Dentro de este grupo se encuentra la criptografía de la curva elíptica que expondremos en este trabajo.

En 1985 es cuando por primera vez surge la idea de utilizar esta herramienta matemática con fines criptográficos, adaptando el problema del logaritmo discreto para obtener una analogía en puntos de una curva elíptica en vez de en un cuerpo finito.

El trabajo desarrollado comienza con una exposición, en la Sección 2, sobre las curvas elípticas. Se muestra su definición y las operaciones de suma y producto por escalares utilizadas para definir una estructura de grupo sobre las mismas. Además, se comenta cómo conseguir un subgrupo a partir de un punto de la curva, pues será necesario para las aplicaciones criptográficas. Continuamos en la Sección 3 formulando el problema del logaritmo discreto y la dificultad para resolverlo, pues en ello depende la seguridad de la criptografía de la curva elíptica. Pasamos entonces, en la Sección 4, a explicar dos algoritmos utilizados en la criptografía de la curva elíptica: ECDH (para el intercambio de claves) y ECDSA (para firmar mensajes). También recogemos, en la Sección 5, algunas de las curvas más usadas en la actualidad en este tipo de criptografía. A continuación, se compara, en la Sección 6, los sistemas basados en curvas elípticas con los basados en RSA, otro criptosistema de clave pública que compitió con la criptografía de la curva elíptica en sus inicios. Por último, se incluye la información necesaria para una demostración práctica. En la Sección **??** comentaremos dónde encontrar los archivos `.ipynb` implementados y las operaciones necesarias para trabajar con curvas elípticas a través de la herramienta `openssl`.

2. Curva Elíptica

Antes de entrar en cómo usar las curvas elípticas en criptografía debemos definirlas y estudiar su estructura de grupo.

Una curva elíptica será el conjunto de puntos que verifiquen la ecuación:

$$y^2 = x^3 + ax + b,$$

donde a y b son constantes y verifican $4a^3 + 27b \neq 0$[I]. Esta ecuación es la **ecuación de Weierstrass** para curvas elípticas[II]. Normalmente, a, b, x, y toman valores en un cuerpo. Por

[I] Imponemos esta condición para que la curva no presente raíces múltiples, que pueden dar problemas en la teoría general que expondremos al generar curvas con autointersecciones, picos, etc.

[II] Existe una versión generalizada de esta ecuación que aporta mayor flexibilidad. Se puede transformar fácilmente una forma a la otra (Washington, 2008).

ejemplo: los números reales $\mathbb{R}$, los números complejos $\mathbb{C}$, los números racionales $\mathbb{Q}$, un cuerpo finito $\mathbb{F}_p$ para un primo p (este es el que se utiliza en criptografía habitualmente), un cuerpo finito $\mathbb{F}_q$ donde $q = p^k$, $k \geq 0$, etc.

De forma general, consideramos un cuerpo K y definimos la curva elíptica E sobre él de la siguiente forma:

$$E = \{(x, y) \in K \times K : y^2 = x^3 + ax + b,\ 4a^3 + 27b \neq 0\} \cup \{\infty\}.$$

El punto ∞ lo incluimos por definición. Una línea que pase por este punto será una línea vertical, luego dos líneas verticales distintas se intersecan en este punto.

Podemos usar SageMath para visualizar algunos ejemplos de curvas elípticas. En el Código 6 observamos la forma genérica de definir en SageMath curvas elípticas.

```
sage:  E = EllipticCurve(K, [a, b]); E
Elliptic Curve defined by y^2  = x^3 + x + 3 over a field K
```

Código 1: Curva elíptica en Sage

El Código 2 es el necesario para crear la curva con ecuación de Weierstrass $y^2 = x^3 - x$.

```
sage: E =  EllipticCurve(RR, [-1, 0]);
      plot(E, (-4, 3), color=hue(0.6))
```

Código 2: Curva elíptica $y^2 = x^3 - x$

Podemos observar la salida obtenida en la Figura 1.

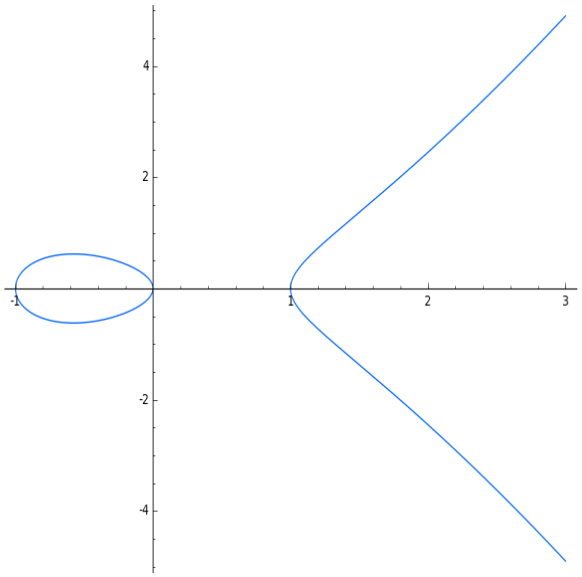

Figura 1: Salida Código 2

Programamos de forma similar el código para visualizar la gráfica de la curva dada por la ecuación $y^2 = x^3 + x$, véase el Código 3, cuya salida se encuentra en la Figura 2.

```
sage: E =  EllipticCurve(RR, [1, 0]);
      plot(E, (-2, 4), color=hue(0.8))
```

Código 3: Curva elíptica $y^2 = x^3 + x$

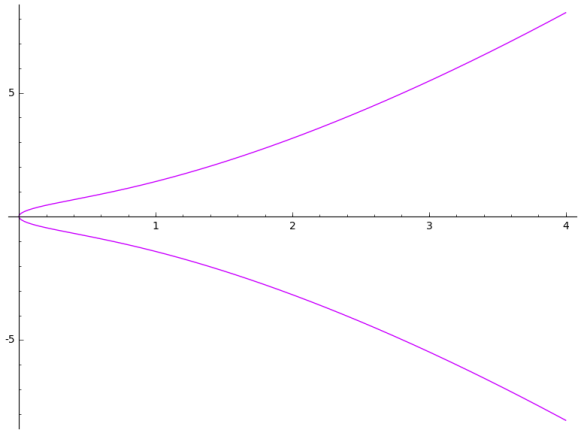

Figura 2: Salida Código 3

En la Figura 3 tenemos una tabla con la forma de las curvas elípticas para valores enteros de a entre -2 y 1 y b entre -1 y 2.

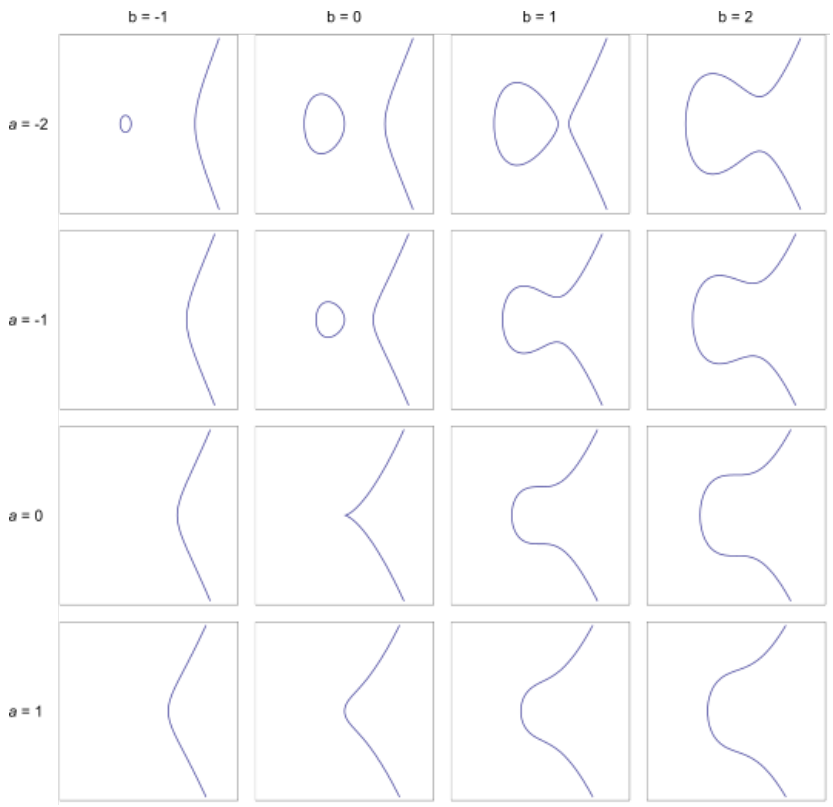

Figura 3: Ejemplos de curvas elípticas sobre $\mathbb{R}$ ("Elliptic curve catalog", 2008)

2.1. Estructura de grupo

Dada una curva elíptica podemos definir un grupo abeliano formado por puntos de la misma.

1. Tomamos como elemento neutro el punto del infinito ∞.

2. El opuesto de un punto P viene dado por su simétrico por el eje de abscisas. Más concretamente, si $P = (x_P, y_P)$ entonces $-P = (x_P, -y_P)$.

3. En general, dados dos puntos de la curva $P = (x_P, y_P)$ y $Q = (x_Q, y_Q)$ con $x_P \neq x_Q$, su suma es el punto $-R$ que viene definida por:

$$(x_R, -y_R) = (m^2 - x_P - x_Q, -y_P - m(x_R - x_P)) = (m^2 - x_P - x_Q, -y_Q - m(x_R - x_Q)),$$

donde

$$m = \frac{y_P - y_Q}{x_P - x_Q}.$$

Definimos $P + Q = -R$ debido al sentido geométrico de la suma en la curva. Esta operación debe seguir la siguiente regla: "dados tres puntos P, Q y R alineados y no nulos su suma $P+Q+R$ debe ser el elemento neutro". Por lo tanto, $P+Q$ es el opuesto

al punto de corte de la recta que une ambos puntos con la curva. De ahí también notamos la presencia del signo menos en la segunda coordenada tal y como indicamos en el punto anterior sobre los opuestos. El sentido geométrico nos aporta intuitivamente la conmutatividad ya que la recta tangente es la misma en el caso $P + Q$ y $Q + P$ y la asociatividad ya que $(P + Q) + R = P + (Q + R) = (P + R) + Q = \infty$.

No podemos usar estas fórmulas en algunos casos especiales, por ejemplo cuando $P = Q$. En este caso no existe una sola recta que pase por "ambos" puntos. Sin embargo, si nos aproximamos a Q mediante puntos Q' con $Q' \neq P$ en seguida vemos que la recta que obtendríamos es la tangente en P. Así, definimos su suma como el opuesto del punto de intersección de la recta tangente en P.

Otra situación interesante es cuando $P \neq Q$ pero no hay más puntos de corte. Nos encontramos en un caso parecido al anterior ya que uno de los puntos es tangente a la curva. Si suponemos que P es donde se da la tangencia, por el caso anterior tememos que $P + P = -Q$ por lo que $P + Q = -P$. En estas situaciones de tangencia tenemos que:

$$m = \frac{3x_P^2 + a}{2y_P}.$$

En ambos casos debemos comprobar que la suma pertenece a la curva y que los tres puntos están alineados.

La Figura 4 muestra la idea geométrica de la operación de suma.

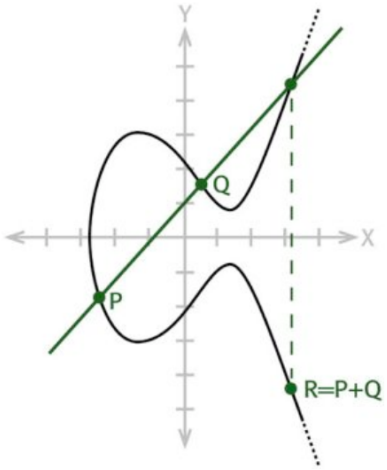

Figura 4: Suma de la curva elíptica $\mathbb{R}$ ("Elliptic Curve Cryptography", s.f.)

.

4. Finalmente, dado $n \in \mathbb{Z}$, el producto por escalares lo definimos como:

$$nP = \begin{cases} (P + \overset{n}{\cdots} + P) & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ (-P - \overset{n}{\cdots} - P) & \text{si } n < 0 \end{cases}.$$

2.2. Curvas elípticas sobre cuerpos finitos

Lo más habitual es que el cuerpo sobre el que trabajemos en criptografía sea un cuerpo finito, $\mathbb{F}_p$ con p un primo. En ese caso, la curva vendría dada por:

$$E = \{(x, y) \in K \times K : y^2 \equiv x^3 + ax + b \text{ mód } p,\ 4a^3 + 27b \not\equiv 0 \text{ mód } p\} \cup \{\infty\}.$$

Para ver la representación de las curvas en estos casos ejecutamos el Código 4. Podemos observar el resultado de la ejecución del código con $p = 113$ en la Figura 5 y para $p = 337$ en la Figura 6.

```
sage: E = EllipticCurve('37a')
      @interact
      def f(p=primes(2,500)):
          show(plot(E.change_ring(GF(p)),pointsize=30),
               axes=False, frame=True, gridlines="automatic",
               aspect_ratio=1)
```

Código 4: Curva elíptica $y^2 + y = x^3 - x$ sobre $\mathbb{F}_p$.

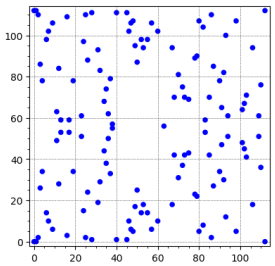

Figura 5: Código 4 para $p = 113$

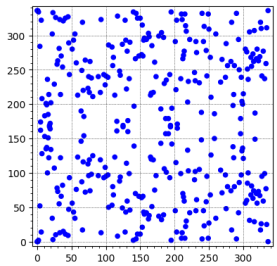

Figura 6: Código 4 para $p = 337$

En cuanto a las fórmulas para la operación del grupo, son las mismas que las definidas anteriormente solo que debemos añadir " mód p" a cada una de ellas. Ello conlleva, por ejemplo, la "repetición" de la recta que une ambos puntos. La discusión sobre su validez, especialmente en los casos especiales notados anteriormente, requiere de unas matemáticas más profundas en las que no vamos a entrar pero que están estudiadas de manera rigurosa.

2.3. Subrupos

Notamos que si, $n, m \in \mathbb{Z}$ y $P \in E$ se cumple que:

$$nP + mP = (n + m)P.$$

Así, los múltiplos de P son cerrados bajo la suma. Obtenemos de este modo que los múltiplos de un punto de la curva forman un subgrupo cíclico. Nos preguntamos ahora cuál es el orden de dichos subgrupos, es decir, debemos encontrar el mínimo $n \in \mathbb{N}$ que cumpla $nP = \infty$. Para su cálculo debemos tener en cuenta el Teorema de Lagrange que nos dice que si S es un subgrupo de H entonces el orden de S es un divisor del orden de H. El orden del grupo de la curva elíptica lo podemos calcular en tiempo polinomial gracias al algoritmo de Schoof (Wikipedia, s.f.). De este modo, como tenemos el orden del grupo formado por la curva elíptica que notamos como N, el orden del subgrupo con generador P queda determinado por:

$$\text{mín}\{n \in \mathbb{N} : \ nP = \infty \text{ y } n|N\}.$$

Normalmente, el proceso seguido en los algoritmos de criptografía de la curva elíptica es el opuesto, es decir, primero elegimos el orden del subgrupo que queremos y posteriormente hallamos el generador. Si el orden del subgrupo es un primo p el proceso a seguir es el siguiente: calculamos el cofactor $h = N/n$, tomamos un punto aleatorio $P \in E$ y obtenemos hP. Si es ∞, tomamos un nuevo punto y si no, P es el generador deseado. Los órdenes de los subgrupos se suelen escoger de orden primo ya que Pohlig y Hellman probaron que un criptosistema es seguro si, y solo si, no es factible el cálculo del logaritmo discreto en grupos con orden un primo (Pohlig y Hellman, 1978). En la siguiente sección exponemos el problema del logaritmo discreto en el contexto de las curvas elípticas.

3. El problema del logaritmo discreto

La seguridad de la criptografía de la curva elíptica reside en la dificultad para calcular logaritmos discretos. Este problema es análogo al de otros criptosistemas como el *Digital Signature Algorithm* (DSA) o el intercambio de claves mediante el mecanismo de Diffie-Hellman.

En nuestro caso, el problema consiste en lo siguiente: "si conocemos P y Q dos puntos de la curva elíptica, ¿podemos encontrar un k tal que $Q = kP$? ". Vemos que no estamos calculando un logaritmo como tal pero se sigue usando ese término por analogía a otros sistemas como se ha mencionado anteriormente.

Actualmente, no existe ninguna demostración matemática que efectivamente pruebe la dificultad de dicho cálculo. Solo sabemos que es difícil pero no podemos estar seguros. Con el siguiente ejemplo ilustramos su complejidad. Para ello usamos el algoritmo *Baby-step, giant-step* que se basa en el hecho de que cualquier entero x puede expresar como $x = am + b$ con a, m y b también enteros. Escribimos entonces:

$$\begin{aligned} Q &= xP \\ &= (am + b)P \\ &= amP + bP. \end{aligned}$$

Así, podemos expresar $Q - amP = bP$. El algoritmo consiste en calcular algunos valores de bP y otros de $Q - amP$ hasta que encontremos una correspondencia. El algoritmo también indica

que debemos escoger $m = \sqrt{n}$ y los números a y b se mueven entre 0 y m por lo que mientras bP tiene incrementos pequeños (*baby*) amP los tiene grandes (*huge*). Este método nos aporta una complejidad tanto en tiempo como en memoria de $O(\sqrt{n})$, es decir, exponencial pero es mejor que uno de fuerza bruta. De todos modos sigue siendo intratable ya que para un n tal que $\sqrt{n} = 7{,}922816251426434 \times 10^{28}$ el almacenamiento de datos que podemos llegar a necesitar es de $2,5 \times 10^{30}$ bytes de memoria (la capacidad de almacenamiento mundial es de aproximadamente 10^{21} bytes).

4. Algoritmos

Los algoritmos criptográficos basados en curvas elípticas (ECC, por sus siglas en inglés *Elliptic Curve Cryptography*) surgen por primera vez en el año 1984 y muchos de ellos son adaptaciones de algoritmos ya existentes para utilizarlos en el grupo definido por una curva elíptica. En esta sección nos centraremos en dos de esos algoritmos: ECDSA y ECDH, uno de firma y otro de intercambio de claves ("Elliptic Curve Cryptography: ECDH and ECDSA - Andrea Corbellini", s.f.).

Expuestas todas las herramientas necesarias, falta mencionar cuáles son los parámetros necesarios para los algoritmos. Serán una séxtupla (p, a, b, G, n, h), donde:

- p es un número primo.
- a, b son los coeficientes de la curva elíptica, hay que elegirlos con cuidado para garantizar que el algoritmo sea seguro.
- G es el generador del subgrupo.
- n es el orden del subgrupo.
- h es el cofactor del subgrupo.

En ocasiones también se añade a estos parámetros una semilla que se utiliza mediante una función *hash* para calcular el orden del grupo o los coeficientes de la curva, de esta forma no dependeríamos de la seguridad de una curva específica.

La ECC es un tipo de sistema de clave pública, luego es necesario definir cuáles serán las claves utilizadas.

- La **clave privada** es un entero aleatorio d elegido en el intervalo $[1, n-1]$.
- La **clave pública** es el punto de la curva $H = dG$.

Conociendo d y G es fácil encontrar H, mientras que conociendo G y H es complejo hallar d, como se explicó en la Sección 3, ya que requiere resolver el problema del logaritmo discreto.

4.1. ECDH

ECDH (*Elliptic Curve Diffie Hellman*) es un protocolo para el intercambio de claves basado en Diffie-Hellman. La situación a resolver es la siguiente: Alice y Bob quieren comunicarse de forma privada sin que Carol pueda descifrar sus mensajes. Alice y Bob pueden usar ECDH para generar e intercambiar las claves que les permitan mantener un intercambio de mensajes seguro siguiendo los pasos a continuación:

1. Comienzan generando sus propias claves. Alice generará su clave privada, d_A y a partir de ella calculará su clave pública $H_A = d_A G$. Bob actuará de la misma forma para obtener sus claves privada, d_B, y pública, $H_B = d_B G$. Notamos que los dos deben usar el mismo generador G en la misma curva elíptica definida sobre el mismo cuerpo finito.

2. Alice y Bob intercambian sus claves públicas, H_A y H_B, mediante un canal potencialmente inseguro. Carol podría interceptarlas, pero no será capaz de descubrir sus claves privadas sin resolver el problema del logaritmo discreto.

3. Para calcular la clave compartida, Alice calcula $C = d_A H_B$ usando su clave privada y la clave pública de Bob. Análogamente, Bob calcula esta clave compartida, será el punto $C = d_B H_A$. Comprobamos que, efectivamente, el punto C es el mismo en ambos casos:

$$C = d_A H_b = d_A(d_B G) = d_B(d_A G) = d_B H_A = C.$$

Carol, sin embargo, solo conoce H_A y H_B y a partir de estos puntos es incapaz de determinar la clave compartida C. Este es el problema de Diffie-Hellman que, en este contexto, se enuncia: "Dados tres puntos P, aP, y bP, ¿cuál es el punto resultante de abP?".

Se cree que el problema de Diffie-Hellman para curvas elípticas es tan complejo como el problema del logaritmo discreto, aunque no se ha probado.

Ahora que Alice y Bob tienen su clave compartida pueden intercambiar mensajes mediante cifrado simétrico, como AES o DES.

4.2. ECDSA

ECDSA es un algoritmo de firma digital, *Elliptic Curve Digital Signature Algorithm*, es una variante del algoritmo DSA (*Digital Signature Algortihm*) aplicado a curvas elípticas. Trabaja con el *hash* del mensaje en lugar de con el propio mensaje. La elección de la función *hash* es importante, de esto dependerá la seguridad del sistema criptográfico. El *hash* del mensaje tendrá una longitud de n bits[III], lo denotamos por z.

Imaginemos que Alice quiere firmar un mensaje con su llave privada, d_A, y la otra persona, Bob, quiere validar la firma con la clave pública de Alice, H_A. Alice es la única que puede

[III] Si la longitud fuera mayor que n, lo truncamos.

producir las firmas válidas, sin embargo todo el mundo que tenga su clave pública puede verificarlas.

Alice y Bob están usando los mismos parámetros del dominio.

Firma del mensaje

A partir de k y z se genera la firma con la clave privada de Alice:

1. Tomamos un entero k de forma aleatoria en el conjunto $\{1, ..., n-1\}$.
2. Calculamos $P = kG$.
3. Hallamos el número $r \equiv x_P \mod n$.
4. Si r es 0 entonces se toma otro k y se intenta de nuevo.
5. Calculamos $s \equiv k^{-1}(z + rd_A) \mod n$ con k^{-1} el inverso multiplicativo de k módulo n.
6. Si s es 0 entonces elegimos otro k y volvemos al principio.

Al final obtendremos la firma que será la pareja (r, s).

Verificación de la firma

Ahora entra en juego Bob que quiere validar la firma, (r, s), a partir del mensaje recibido con valor *hash* z y de la clave pública de Alice, H_A.

1. Calculamos $u_1 \equiv s^{-1}\ z \mod n$.
2. Calculamos $u_2 \equiv s^{-1}\ r \mod n$.
3. Calculamos el punto $P = u_1G + u_2H_A$.

La firma solo es válida si $r = x_P \mod n$.

Exactitud del algoritmo

Veamos que esta definición de P del punto 3 coincide con la definición original de $P = kG$. Partimos de $P = u_1G + u_2H_A$, sustituimos el valor de la clave pública $H_A = d_AG$

$$\begin{aligned} P &= u_1G + u_2H_A \\ &= u_1G + u_2d_AG \\ &= (u_1 + u_2d_A)G. \end{aligned} \tag{1}$$

Sustituimos los valores de u_1 y u_2 que hemos declarado en la verificación de la firma:

$$\begin{aligned} P &= (u_1 + u_2 d_A)G \\ &= (s^{-1}z + s^{-1}rd_A)G \\ &= s^{-1}(z + rd_A)G. \end{aligned} \tag{2}$$

Como el subgrupo cíclico de orden n en el que estamos trabajando está generado por G no hace falta poner " mód n" si multiplicamos por G.
Ahora nos fijamos en la definición de $s = k^{-1}(z+rd_A) \mod n$. Multiplicamos por k a ambos lados y dividimos por s y obtenemos: $k = s^{-1}(z + rd_A) \mod n$.

$$\begin{aligned} P &= s^{-1}(z + rd_A)G \\ &= kG. \end{aligned} \tag{3}$$

Importancia de k

Es importante el valor que escojamos de k el entero aleatorio utilizado para firmar. Si además siempre firmamos con el mismo valor de k y con los mismos valores aleatorios, para los atacantes será más fácil encontrar la clave privada.
Para descubrir la clave privada d_S necesitamos calcular k a partir de los valores *hash* z_1 y z_2 y de las correspondientes firmas (r_1, s_1) y (r_2, s_2).

1. r_1 es igual a r_2, puesto que $r = x_P \mod n$ y $P = kG$ es el mismo para las dos firmas.
2. Observamos que $(s_1 - s_2) \mod n \equiv k^{-1}(z_1 - z_2) \mod n$ viene directamente de la definición de s.
3. $k(s_1 - s_2) \mod n \equiv (z_1 - z_2) \mod n$.
4. $k \equiv (z_1 - z_2)(s_1 - s_2)^{-1} \mod n$.

Ahora despejamos d_S de la definición de s y obtenemos que:

$$\begin{aligned} s &\equiv k^{-1}(z + rd_S) \mod n \\ d_S &\equiv r^{-1}(sk - z) \mod n. \end{aligned} \tag{4}$$

Así hemos conseguido la clave privada con los valores que conocemos.

5. Curvas elípticas más utilizadas

Como hemos explicado, los participantes en los algoritmos de cifrado en la curva elíptica deben escoger los mismos parámetros para el dominio. Entre ellos se encuentra la propia curva.

Actualmente, existen curvas que son más usadas que otras debido, por ejemplo, al nivel de seguridad que presentan en el cálculo del problema del logaritmo discreto. Destacan:

1. Spec256k1 (o Curve25519): se utiliza para ECDSA en el modelo criptográfico de Bitcoin ("Secp256k1", s.f.). Viene dada por:

$$y^2 = x^3 + 7.$$

   Se suele utilizar como punto base

$$G = (0\mathtt{x}79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,\\ 0\mathtt{x}483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),$$

   que nos aporta un grupo de orden

$$n = 2^{256} - 0\mathtt{x}14551231950b75fc4402da1732fc9bebf.$$

2. NIST P-256: es la que trae *OpenSSL* por defecto y existen una gran cantidad de métodos para optimizar su uso.

3. Curve25519: es una de las curvas más rápidas que existen actualmente y su implementación de referencia es de dominio público (Langley y Hamburg, s.f.). Viene dada por

$$y^2 = x^3 + 48662x^2 + x$$

   definida sobre el cuerpo finito con $2^{255} - 19$ elementos el punto base tal que x 9. Esto nos aporta un subgrupo de orden $2^252 + 0\mathtt{x}14def9dea2f79cd65812631a5cf5d3ed$ ("Curve25519", s.f.).

4. Curve448: ofrece potencialmente 224 bits de seguridad y aporta un gran rendimiento ("Curva 448", s.f.). Su implementación base está disponible bajo una licencia del MIT.

6. Comparación con RSA

La principal diferencia entre la criptografía de la curva elíptica y la de RSA es el problema en el que basan su seguridad. La curva elíptica lo hace en el problema del logaritmo discreto, mientras que RSA lo hace en el problema de factorización. Otra diferencia fundamental es la cantidad de almacenamiento que se necesita para guardar una clave al mismo nivel de seguridad. Por ejemplo, si para guardar una clave de RSA se necesitan 1024 bits, para almacenar la de ECC se requiere de 160. Las diferencias son aún mayores cuando aumenta el nivel de seguridad, para RSA son necesarios 15360 bits mientras que para ECC tan solo 521 bits.

La relación entre los tamaños no es lineal, es decir, si multiplicamos por k el tamaño de RSA, no tenemos k veces el tamaño de ECC.
Por otro lado, la generación de la clave y la firma es considerablemente más rápida con ECC que con RSA.

Los algoritmos más rápidos para el cómputo de logaritmos discretos sobre una curva elíptica son Pollard's *rho* y *baby-step giant-step*. Destaca uno en particular que es el algoritmo "criba general del cuerpo de números" el más rápido conocido a día de hoy para la factorización de enteros, que además se puede usar para el logaritmo discreto.

7. Demostración práctica

7.1. SageMath

Se ha utilizado SageMath para representar curvas elípticas y para realizar una implementación propia del algoritmo ECDSA, puesto que esta herramienta software contiene funciones específicas para el tratamiento de curvas elípticas.
El código se encuentra en la carpeta `src`. En el archivo `elliptic_curves.ipynb` se recoge la implementación de los ejemplos mostrados en este archivo. En el archivo `Curvas elípticas.ipynb` está la implementación del algoritmo ECDSA y una situación hipotética de uso en la que Alice y Bob utilizan la firma de mensajes.

7.2. OpenSSL

Si en la práctica quisieramos usar la criptografía de la curva elíptica, no utilizaríamos SageMath, ni implementaciones propias del algoritmo. Probablemente aprovecharíamos la herramienta OpenSSL ("Command Line Elliptic Curve Operations - OpenSSLWiki", s.f.), que los únicos algoritmos de este tipo que soporta son, precisamente, ECDH y ECDSA.

Generar las claves

Las claves pública y privada se guardarán en archivos con formato `.pem`, cifrado o no cifrado. OpenSSL contiene muchas curvas predefinidas que podemos usar, para verlas solo hay que ejecutar el comando mostrado en el código 5.

```
$ openssl ecparam -list_curves
  secp112r1 : SECG/WTLS curve over a 112 bit prime field
  secp112r2 : SECG curve over a 112 bit prime field
  secp128r1 : SECG curve over a 128 bit prime field
  secp128r2 : SECG curve over a 128 bit prime field
  secp160k1 : SECG curve over a 160 bit prime field
  secp160r1 : SECG curve over a 160 bit prime field
  secp160r2 : SECG/WTLS curve over a 160 bit prime field
  secp192k1 : SECG curve over a 192 bit prime field
  secp224k1 : SECG curve over a 224 bit prime field
  secp224r1 : NIST/SECG curve over a 224 bit prime field
  secp256k1 : SECG curve over a 256 bit prime field
  secp384r1 : NIST/SECG curve over a 384 bit prime field
  secp521r1 : NIST/SECG curve over a 521 bit prime field
  prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
  prime192v2: X9.62 curve over a 192 bit prime field
  prime192v3: X9.62 curve over a 192 bit prime field
  prime239v1: X9.62 curve over a 239 bit prime field
  prime239v2: X9.62 curve over a 239 bit prime field
```

```
prime239v3: X9.62 curve over a 239 bit prime field
prime256v1: X9.62/SECG curve over a 256 bit prime field
sect113r1 : SECG curve over a 113 bit binary field
sect113r2 : SECG curve over a 113 bit binary field
sect131r1 : SECG/WTLS curve over a 131 bit binary field
...
```

Código 5: OpenSSL: listar curvas por defecto

Para generar un archivo con los parámetros de la curva ejecutamos el Código **??**, donde podemos cambiar `secp256k1` por la curva elegida.

```
$ openssl ecparam -name secp256k1 -out secp256k1.pem
$ cat secp2561.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
```

Código 6: Generar los parámetros de la curva

Para generar la pareja de clave privada tenemos dos opciones: generarla a través de unos parámetros de la curva preexistentes (Código 7) o simplemente seleccionando el nombre de la curva (Código 8).

```
$ openssl ecparam -in secp256k1.pem -genkey -noout -out secp256k1-key.pem
$ cat secp256k1-key.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIAbXuq9SpxisBn/p4Vk+Ce4aLIJZgJpg9Hp0CfZs5PKAoAcGBSuBBAAK
oUQDQgAEdr/qwSqOx966+SLboZOlalO2QTCFrwbiFaLcBhqPmjk+D8FnWo6HLcrT
nBhoVk4IAUhsFGMgudcVYwgOFuHAXQ==
-----END EC PRIVATE KEY-----
```

Código 7: Crear clave privada, parámetros de curva preexistentes

```
$ openssl ecparam -name secp256k1 -genkey -noout -out secp256k1-key.pem
$ cat secp256k1-key.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIH9lFVGJd8SVrSd47l3MAjx6SiapcAw6j+SxDmBYXzJ6oAcGBSuBBAAK
oUQDQgAEiG8RZTai+T9mjFFoCPcanTvDfsQ2gHSvgHAtvDItnLVlc0TVo+YRiT/+
81r+9k1Fe6sHi8aGNycLHKiyY8Yz9Q==
-----END EC PRIVATE KEY-----
```

Código 8: Crear clave privada a partir del nombre de la curva

Una vez tenemos la clave privada para a partir de ella generar la clave pública requiere de ejecutar el comando mostrado en el Código 9

```
$ openssl ec -in secp256k1-key.pem -pubout -out ecpubkey.pem
read EC key
writing EC key
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEiG8RZTai+T9mjFFoCPcanTvDfsQ2gHSv
gHAtvDItnLVlc0TVo+YRiT/+81r+9k1Fe6sHi8aGNycLHKiyY8Yz9Q==
-----END PUBLIC KEY-----
```

Código 9: Crear la clave pública a partir de la privada

Referencias

Washington, L. (2008, 3 de abril). Elliptic curves: Number theory and cryptography, second edition. (Vol. 20080550, pp. 10, 11). doi:10.1201/9781420071474

Elliptic curve catalog. (2008). Último acceso el 7 de diciembre de 2019, desde https://commons.wikimedia.org/wiki/File:EllipticCurveCatalog.svg#/media/File:EllipticCurveCatalog.svg

Elliptic Curve Cryptography. (s.f.). https://sinhvientot.net/elliptic-curve-cryptography/.

Wikipedia. (s.f.). Schoof's algorithm. Último acceso el 1 de noviembre de 2019, desde https://en.wikipedia.org/wiki/Schoof%27s_algorithm

Pohlig, S. & Hellman, M. (1978). An improved algorithm for computing logarithms over GF (p) and its cryptographic significance (Corresp.) *IEEE Transactions on information Theory*, *24*(1), 106-110.

Elliptic Curve Cryptography: ECDH and ECDSA - Andrea Corbellini. (s.f.). Último acceso el 7 de diciembre de 2019, desde https://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

Secp256k1. (s.f.). https://es.bitcoin.it/wiki/Secp256k1.

Langley, A. & Hamburg, M. (s.f.). Elliptic Curves for Security. Último acceso el 7 de diciembre de 2019, desde https://tools.ietf.org/html/rfc7748#page-4

Curve25519. (s.f.). https://es.wikipedia.org/wiki/Curve25519.

Curva 448. (s.f.). https://es.wikipedia.org/wiki/Curva448.

Command Line Elliptic Curve Operations - OpenSSLWiki. (s.f.). Último acceso el 8 de diciembre de 2019, desde https://wiki.openssl.org/index.php/Command_Line_Elliptic_Curve_Operations

Everyone Loves Curves! But Which Elliptic Curve is the Most Popular? (s.f.). https://malware.news/t/everyone-loves-curves-but-which-elliptic-curve-is-the-most-popular/17657.

Elliptic Curve Cryptography: a gentle introduction - Andrea Corbellini. (s.f.). Último acceso el 8 de diciembre de 2019, desde https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/