

ECC (Elliptic Curve Cryptography)

Let p be a prime > 3 . An **elliptic curve** over the finite field \mathbb{F}_p is a plane curve given by the equation

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

where a, b are integers modulo p and $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$, together with a point at infinity \mathbf{O} .

The set of points of the curve is

$$E(\mathbb{F}_p) = \{(x, y) \mid x, y \text{ integers modulo } p \text{ that satisfy the equation}\} \cup \{\mathbf{O}\}$$

In terms of implementation, the point \mathbf{O} can be represented with three coordinates, the last of which being a 0.

In this set, addition is defined using the following condition: the sum of three points is \mathbf{O} if and only if the points lie on a line. More concretely, the **sum of points** is given in the following way: if $P = (x_1, y_1)$ is a point of the curve, set $-P = (x_1, -y_1 \pmod{p})$. Then,

- $P + \mathbf{O} = P$
- $P + (-P) = \mathbf{O}$
- if $Q = (x_2, y_2) \neq -P$, then $P + Q = (x', \lambda(x_1 - x') - y_1 \pmod{p})$, where $x' = \lambda^2 - x_1 - x_2 \pmod{p}$ and

$$\lambda = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} \pmod{p} & \text{si } Q \neq P, \\ (3x_1^2 + a)(2y_1)^{-1} \pmod{p} & \text{si } Q = P. \end{cases}$$

With this operation, the set of points of the elliptic curve has the structure of an abelian group: the sum is associative and commutative, has an identity element (namely \mathbf{O}) and every element P has an inverse $-P$ (the inverse of \mathbf{O} is itself).

In this group, the *exponentiation* kP can be done with the **algorithm of “successive squares”**: first the binary expression of k is computed, that is, $k = b_r 2^r + \dots + b_1 2 + b_0$, where $b_i \in \{0, 1\}$ and $b_r = 1$; then

```

R ← O, i ← r
while i ≥ 0 do
    R ← R + R
    if b_i = 1 do R ← R + P
    i ← i - 1
output R
    
```

In the elliptic curves that are used in Cryptography, the group of points $E(\mathbb{F}_p)$ has size $2^m n$, where n is a *prime* number and the cofactor $h = 2^m$ has exponent $0 \leq m \leq 16$. Under these conditions, there is always a point G of order n , that is, $nG = \mathbf{O}$ but all the previous multiples $G, 2G, 3G, \dots, (n-1)G$ are $\neq \mathbf{O}$.

Given the parameters p, a, b, n, G of a cryptographic system, the **private key** of a user is a random integer r , $1 < r < n - 1$, and his/her **public key** is the point $P = rG$.

Diffie-Hellman key exchange. If the private and public keys of a pair of users are $(r_1, P_1 = r_1G)$ and $(r_2, P_2 = r_2G)$, then the first user can compute r_1P_2 and the second one can compute r_2P_1 , so that both of them are actually computing the same point: $r_1r_2G = (x, y)$

This can be used to obtain a 256-bit secret key, which they can, for instance, feed to AES:

$$K = \text{SHA256}(s||x)$$

where s is a random number that they can interchange publicly and $||$ indicates the concatenation of binary words.

ECDSA (Elliptic Curve Digital Signature Algorithm). With the same parameters p, a, b, n, G as before, the signature of a message M by the user whose private key is r and whose public key is P works as follows:

- $kG = (x_1, y_1)$ with $1 < k < n - 1$ random
- $f_1 = x_1 \bmod n$
- $f_2 = k^{-1} (H(M) + f_1r) \bmod n$
- **Send the signature** (f_1, f_2)

If $f_1 = 0$ or $f_2 = 0$, one must go back to the first step and generate a new value of k . Notice that if the prime number p has length ℓ , then the signature has length $\leq 2\ell$.

The receiver verifies the signature in the following way:

- $w_1 = H(M)f_2^{-1} \bmod n$
- $w_2 = f_1f_2^{-1} \bmod n$
- $w_1G + w_2P = (x_0, y_0)$
- **Accept** if $x_0 \bmod n = f_1$

In this algorithm, $H(M)$ denotes a hash of the message M . In the implementation we will consider that the prime number p has 256 bits and that the hash function is *SHA256*. In particular, the signature will be expressed with 64 bytes.

Implementation: signatures. Define the class `ecc` including the following methods:

```
public static BigInteger [] invers( BigInteger [] P, BigInteger[] ParametrosCorba)
```

input: `P` point of the curve given by 3 coordinates (x, y, z) , (if $z = 0$, it's the point at infinity), `ParametrosCorba` = $\{a, b, p\}$, are the parameters of the curve $y^2 = x^3 + ax + b \bmod p$;
output: a list $\{R_x, R_y, R_z\}$ representing the inverse of P , $R = -P$ (if $R_z = 0$, it's the point at infinity).

```
public static BigInteger [] suma( BigInteger [] P, BigInteger [] Q, BigInteger[] ParametrosCorba)
```

input: `P` and `Q` points of the curve given by 3 coordinates (x, y, z) , (if $z = 0$, it's the point at infinity), `ParametrosCorba` = $\{a, b, p\}$ are the parameters of the curve $y^2 = x^3 + ax + b \bmod p$;
output: a list $\{R_x, R_y, R_z\}$ representing the point $R = P + Q$ (if $R_z = 0$, it's the point at infinity).

```
public static BigInteger [] multiple(BigInteger k, BigInteger [] P, BigInteger[] ParametresCorba)
```

input: k integer,
P point of the curve given by 3 coordinates (x, y, z) , (if $z = 0$, it's the point at infinity),
ParametresCorba= $\{a, b, p\}$ are the parameters of the curve $y^2 = x^3 + ax + b \pmod p$;
output: a list $\{R_x, R_y, R_z\}$ representing the point $R = P + \dots + P = k \cdot P$ (if $R_z = 0$, it's the point at infinity).

```
public static BigInteger[] clausECC(BigInteger[] parametresECC)
```

input: parametresECC= $\{n, G_x, G_y, a, b, p\}$, $G = (G_x, G_y)$ is a point of order n in the curve $y^2 = x^3 + ax + b \pmod p$ (obviously, G is not the point at infinity);
output: a list $\{r, P_x, P_y\}$, r is the private key, and (P_x, P_y) is the point (different from the point at infinity) that is the public key.

```
public static BigInteger ECCDHKT(byte[] bytesAleatoris, BigInteger clauPrivadaECC,  
                                BigInteger[] clauPublicaECC, BigInteger[] parametresECC)
```

input: bytesAleatoris is a list of random bytes,
clauPrivadaECC is an integer,
clauPublicaECC= $\{P_x, P_y\}$ (different of the point at infinity)
parametresECC= $\{n, G_x, G_y, a, b, p\}$, $G = (G_x, G_y)$ is a point of order n of the curve $y^2 = x^3 + ax + b \pmod p$ (obviously, G is not the point at infinity);
output: a 256-bit secret key to be used in AES: With clauPrivadaECC and clauPublicaECC a key DH with components (x, y) is computed. The secret key k is given by $k = \text{sha256}(\text{bytesAleatoris} || x)$, x in bytes (without two's complement).

```
public static byte[] firmarECCDSA(byte[] M, BigInteger clauFirma, BigInteger[] parametresECC)
```

input: M is an arbitrarily long array of bytes, the message to be signed,
clauFirma is an integer, the private key of the sender
parametresECC= $\{n, G_x, G_y, a, b, p\}$, $G = (G_x, G_y)$ is a point of order n in the curve $y^2 = x^3 + ax + b \pmod p$ (obviously, G is not the point at infinity);
output: an array of bytes, the signed message; it is the concatenation of the array M with an array of exactly 64 bytes representing the signature.

```
public static boolean verificarECCDSA(byte[] MS, BigInteger[] clauVer, BigInteger[] parametresECC)
```

input: MS is a message (allegedly) signed by the system ECCDSA with parameters parametresECC using the private key corresponding to the public key clauVer;
output: a boolean indicating whether the signature is authentic or not.