Elliptic curve cryptography

The elliptic curve discrete logarithm problem. Recall that the DLP consists in solving $g^x = h$ in $\mathbb{F}_p^*$ for given g and h. The elliptic curve discrete logarithm problem ECDLP is the analogue changing the multiplicative group operation by the group law in the elliptic curve. It consists in finding $x \in \mathbb{Z}$ such that $xG = H$ where G and H are points on a given elliptic curve over a finite field. We say that x is the discrete logarithm of H to the base G.

In Sage it can be solved with `G.dicrete_log(H)`. For instance

```
E = EllipticCurve(GF(103), [1,1])
G = E([0,1])
H = 20*G
print G.discrete_log(H)
```

prints `20`. If we replace `20` by `100` the result is `13` because the order of G is 87. By the way, the latter value is obtained with `additive_order(G)`.

No algorithm is known to compute discrete logarithms in an elliptic curve over $\mathbb{F}_p$ in less than $\sqrt{p}$ steps. This means that using p with a hundred digits (or even much less) is safe. In the previous listing changing `GF(next_prime(103))` by `GF(next_prime(10^20))` could be too much for Sage running in a usual computer.

Of course in applicatins one looks for G having large order. In Sage the structure of the abelian group of an elliptic curve `E` is given by `E.abelian_group()`. On the other hand, `E.gens()` gives a list with the generators in such a way that the first one has maximal order.

```
E = EllipticCurve(GF(47), [1,1])
print E.abelian_group()
print E.gens()
print 'Element of maximal order =',E.gens()[0]
```

A possible output for this listing is:

```
(Multiplicative Abelian Group isomorphic to C30 x C2, ((44 : 21 : 1),(35 : 0 : 1)))
((44 : 21 : 1), (35 : 0 : 1))
Element of maximal order = (44 : 21 : 1)
```

The format of `E.abelian_group()` can vary from a version of Sage to another. The previous output means that $P = (44, 21)$ and $Q = (35, 0)$ are points of order 30 and 2, respectively and any point on E can be written as $mP + nQ$ with $m, n \in \mathbb{Z}$.

The elliptic curve ElGamal cryptosystem. In principle one can copy the classic ElGamal cryptosystem changing the multiplicative structure of $\mathbb{F}_p^*$ by the group law in an elliptic curve E over $\mathbb{F}_p$ (or a finite field).

A point $G \in E$ of large order and E itself are public information. The private key is an integer k_2 less than the order of G and the public key is $K_1 = k_2 G$. The hardness of ECDLP assures that it is difficult to recover K_1 from k_2.

The set of plaintext messages is the set of points over the given field. The encryption and decryption functions are

$$\begin{aligned} e_{K_1}(M) &= (rG, M + rK_1) \qquad r = \text{ random number} \\ d_{k_2}(C_1, C_2) &= C_2 - k_2 C_1 \end{aligned}$$

A technical problem is how to encode characters into points of an elliptic curve (note that $M \in E$).

There is a variation of the cryptosystem sometimes called MV-ElGamal (MV stands for Menezes and Vanstone) that avoids this technical problem. In this version a message M is divided into two blocks m_1 and m_2 modulo p, i.e. $\mathbb{F}_p \times \mathbb{F}_p$ is the set of plaintext messages (and the encoding is very easy).

The encryption function is given by

$$e_{K_1}(M) = (rG, c_1, c_2) \in E \times \mathbb{F}_p \times \mathbb{F}_p$$

where $c_1 \equiv x m_1 \pmod{p}$, $c_2 \equiv y m_2 \pmod{p}$, with $(x, y) = rK_1$. We assume $x, y \neq 0$, otherwise we choose another random r. The corresponding decryption function is

$$d_{k_2}(C_0, c_1, c_2) = (c_1 x^{-1}, c_2 y^{-1}) \qquad \text{where } (x, y) = k_2 C_0.$$

For instance, if we choose

```
#
# Choose the elliptic curve modulo p = large prime
# and G a point of high order
#
p = next_prime(10^10)
E = EllipticCurve( GF( p ), [2011,1])
G = E([0,1])
print G.additive_order()
```

The output is 3333330247, then the order G is quite large.

The functions e_{K_1} and d_{k_2} introduced before can be coded as:

```
#
# Encryption and decryption functions
#
def encrypt_mv_eg(Kpub,m1,m2):
    x,y = 0,0
    while( (x==0) or (y==0) ):
        r = floor( p*random() )
        x = (r*Kpub)[0]
        y = (r*Kpub)[1]
    return r*G, m1*x, m2*y

def decrypt_mv_eg(kpri,enc):
    x = (kpri*enc[0])[0]
    y = (kpri*enc[0])[1]
    return enc[1]*x^-1, enc[2]*y^-1
```

If it is a valid cryptosystem then $d_{k_2}\big(e_{K_1}(M)\big) = M$

```
#
# Example
#
private_key = 12345
public_key = private_key*G
decrypt_mv_eg(private_key, encrypt_mv_eg(public_key,10101,33333))
```

We recover the original message `(10101,33333)`.

Recall that we can convert strings of characters into integers thanks to the following simple encoding and decoding functions:

```
# text to number
def encoding(text):
    result = 0
    for c in text:
        result = 256*result +ord(c)
    return result

# number to text
def decoding(number):
        number = Integer(number)
        result = ''
        for i in  number.digits(256):
                result = chr(i) + result
        return result
```

Actually in our case we need to divide into an even number of blocks. If we think in a character as a number < 256 (its ASCII code) and we employ $\mathbb{F}_p$ as a field then we can encode at most $\log_{256} p$ characters in each block.

```
#
# TABLE for a long text
# 1st column: Decoded and decrypted text (original message)
# 2nd, 3rd: encoded blocks
# rest: encrypted blocks
#
text = 'This is a long text to be subdivided into blocks'
k = floor( log(p,256) )
key = 12345
for i in range(0, len(text), 2*k):
    m1 = encoding(text[i:i+k])
    m2 = encoding(text[i+k:i+2*k])
    enc = encrypt_mv_eg(key*G,m1,m2)
    d1 = decoding( decrypt_mv_eg(key, enc)[0] )
    d2 = decoding( decrypt_mv_eg(key, enc)[1] )
    print d1+d2, m1,m2, enc
```

This is	1416128883 543781664	((6085741895 : 8254518770 : 1), 7312388880, 5371594140)
a long t	1629514863 1852252276	((8649855487 : 1362971917 : 1), 286631972, 6170749646)
ext to b	1702392864 1953439842	((9600714213 : 1592560103 : 1), 7774722895, 1581078717)
e subdiv	1696625525 1650747766	((6309572051 : 9204716993 : 1), 4678543272, 9009446437)
ided int	1768187236 543780468	((5659728365 : 447382763 : 1), 6143475220, 9237109331)
o blocks	1864393324 1868786547	((434505921 : 4258432774 : 1), 8775161069, 8407264212)