Tutorial #8

João Freitas up202100373
Rui Gonçalves up202103077

O presente relatório tem como objetivo descrever o processo seguido para a resolução da ficha `Tutorial #8`, disponibilizada no âmbito da disciplina de Criptografia Aplicada. As seções numeradas em baixo representam cada um dos exercícios resolvidos.

1) Problema do Logaritmo Discreto e Curvas Elípticas

A temática do presente problema devolve-se no uso de sistemas criptográficos, desenhados para campos finitos de inteiros (*integer finite field*). É descrito que duas entidades, Alice e Bob, trocam mensagens com o conhecimento de um inteiro primo *p*, curva elíptica *E* descrita sobre o campo *Fp* e um ponto *P* que pertence à curva (`P ∈ E(Fp)`). É também referido que a Alice escolhe um valor inteiro *nA*, que serve como valor multiplicador do ponto P (nA*P) sendo o resultado desta operação o ponto Qa (`Qa = nA*p`), que representa a chave pública da Alice.

Para a mensagem *M* que pertence à curva E a ser transmitida, o Bob escolhe um valor *k* como a sua chave única e calcula os pontos *C1* e *C2*, que representam a mensagem cifrada, sendo estes dados pelas seguintes expressões:

`C1 = k*P` `C2 = M + k*Qa`

O Bob envia estes dois pontos à Alice, que os utiliza para recuperar a mensagem M:

`M = C2 - nA*C1 = (M + k*Qa) - nA*(K*P) = M + k*(nA*P) - nA*(k*p) = M + k*nA*P - k*nA*P = M`

Este processo é conhecido como a cifração **ElGamal**. Contudo, existe um problema que consiste na representação da mensagem em pontos da curva elitica. Para resolver o mesmo, pode-se utilizar uma variante do El Gamal, o *Menezes-Vanstone*, que adicionalmente ao El Gamal, propôem que:

- A mensagem a enviar pelo Bob é codificada como dois inteiros, *m1* e *m2*, sobre um campo finito *Zp*, sendo que a chave unica k pertence também a este campo;
- O Bob calcula os pontos *R* e *S*, de forma similar aos pontos C1 e C2 (`R = k*P`) e (`S = (xS, yS) = k*Qa`), e de seguida refere que os pontos da mensagem cifrada *c1* e *c2*, são congruentes sobre o módulo de p (`c1 = xS*m1 mod(p); c2 = yS*m2 mod(p)`), enviando à Alice (R, c1, c2);
- A Alice calcula *T*, *m'1* e *m'2*, obtendo assim a mensagem original.

```
T = (xT, yT) = nA*R
m'1 ≡ (xT^-1) * c1 mod(p)
m'2 ≡ (yT^-1) * c2 mod(p)
```

a) Prove que `(m1, m2) = (m'1, m'2)`

É possivel provar a expressão ao expandir os termos com base no que é conhecido do sistema criptográfico:

```
(m1, m2) = (m'1, m'2)
= (c1/xS, c2/yS) = ((xT^-1) * c1, (yT^-1) * c2) //1.
= ((xS^-1) * c1, (yS^-1 * c2)) = ((xT^-1) * c1, (yT^-1) * c2) //2.
= (xS^-1, yS^-1) = (xT^-1), (yT^-1)
= (k*Qa)^-1 = (nA*R)^-1 //3.
= (k*nA*P)^-1 = (nA*k*P)^-1 //4.
```

Dado que:

1. `c1, c2, m1, m2` são congruentes ao modulo de *p*;
2. Uma divisão entre dois elementos *a* e *b* (a/b), pode ser representado como o produto da exponencial negativa com *a* (a/b = (b^-1) * a);
3. `(xS, yS) = k*Qa` e `(xT, yT) = nA*R`;
4. `Qa = nA*P` e `R = k*P`.

b) Implemente em Sage/Python, três métodos que para uma curva Eliptica

1. Produzam uma chave pública `Qa` e privada `nA`, com a assinatura `GenPubKey(A, B, p, xP, yP)`;
2. Produzam um texto cifrado `(R, c1, c2)`, com a assinatura `Encript(A, B, p, xP, yP, Qa, m1, m2)`;
3. Recuperam a mensagem original `(m1, m2)`, com a assinatura `Decript(A, B, p, xP, yP, R, c1, c2)`.

Estes métodos são facilmente implementados, pois apenas é necessário traduzir as expressões da curva elíptica e da variante previamente analisados. Para a curva elíptica recorreu-se ao código disponibilizado, de forma a estabelecer a curva `P-384`.

```
p = 2**384 - 2**128 - 2**96 + 2**32 - 1
Fp = FiniteField(p)
A = -3
B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef

Fp = FiniteField(p)
E = EllipticCurve(Fp,[A,B])
```

Nos restantes métodos foi necessário traduzir as expressões e escolher pontos aleatórios dos campos finitos *Fp* e *Zp*, como demonstrado nos seguintes *code snippets*:

```
def GenPubKey(A, B, p, xP, yP):
    Fp = FiniteField(p)
    E = EllipticCurve(Fp,[A,B])
    P = (xP, yP)
    nA = ZZ(Fp.random_element())

    Qa = tuple([nA*x for x in P]) # nA * P

    return Qa, nA

def Encript(A, B, p, xP, yP, Qa, m1, m2):
    Fp = FiniteField(p)
    E = EllipticCurve(Fp,[A,B])
    P = (xP, yP)
    k = ZZ(Fp.random_element())
    S =  tuple([k*x for x in Qa]) # k * Qa
    xS = S[0]
    yS = S[1]

    R = tuple([k*x for x in P]) # k * P
    c1 = (xS*m1) % p
    c2 = (yS*m2) % p

    return R, c1, c2

def Decript(A, B, p, xP, yP, R, c1, c2):
    P = (xP, yP)
    T = tuple([nA*x for x in R]) # nA * R

    xT = T[0]
    yT = T[1]

    m1 = ((xT ^ -1) * c1) % p
    m2 = ((yT ^ -1) * c2) % p

    return m1, m2
```

No método `Decript` optou-se por calcular *m'1* e *m'2*, dado que foi previamente provado que estes dois pontos são iguais a *m1* e *m2*. É também possível confirmar que a implementação dos métodos ao verificar que as o output do método `Decript` é igual a *m1* e *m2*.

```
xP, yP, m1, m2 = 3, 4
Qa, nA = GenPubKey(A, B, p, xP, yP)
R, c1, c2 = Encript(A, B, p, xP, yP, Qa, m1, m2)
rm1, rm2 = Decript(A, B, p, xP, yP, R, c1, c
assert((m1, m2) == (rm1, rm2))
```