TP2_Exercicio3

April 2, 2024

1 TP2 - Exercício 3

1.0.1 Autores

Afonso Ferreira - pg52669
Tiago Rodrigues - pg52705

1.0.2 Enunciado

O algoritmo de Boneh e Franklin (BF) discutido no +Capítulo 5b: Curvas Elípticas e sua Aritmética é uma tecnica fundamental na chamada "Criptografia Orientada à Identidade". Seguindo as orientações definidas nesse texto, pretende-se construir usando Sagemath uma classe Python que implemente este criptosistema.

[ ]:
```
from sage.all import *
from sage.schemes.elliptic_curves import *
```

Começamos por a criação das variáveis necessárias para o problema, conforme visto no RFC5091, como a geração das curvas $E_1 \equiv E/\mathbb{F}_p$ e $E_2 \equiv E/\mathbb{F}_{p^2}$ com a equação $y^2 = x^3 + 1$

[ ]:
```
# Geração dos primos  q, p
bq     = 160                # tamanho em bits do primo "q". Deve ser entre␣
 ↪160-bit e 512-bit
bp     = 512                # tamanho minimo em bits do primo "p". Deve ser␣
 ↪entre 512-bit e 7680-bit

# q - A 160-bit to 512-bit prime that is the order of the cyclic subgroup of␣
 ↪interest in E(F_p).
q = random_prime(2**bq-1,lbound=2**(bq-1))

# tem de se verificar p = 2^t * q * 3 - 1 iterativamente até encontrar um primo
t = q*3*2^(bp - bq)
while not is_prime(t-1):
    t = t << 1

p = t - 1

Fp = GF(p)          # corpo primo com "p" elementos
R.<z> = Fp[]        # anel dos polinomios em "z" de coeficientes em Fp
```

```
f = R(z^2 + z + 1)
Fp2.<z> = GF(p**2, modulus=f)
# extensão de Fp de dimensão 2 cujo módulo é o polinómio "f"
# o polinómio "f"  é irredutivel, tem grau 2 e verifica  z^3 = 1 mod f
# se o ponto (x,y) verificar a equação y^2 = x^3 + 1,
#      então o ponto (z*x,y) verifica a mesma equação

# Função que mapeia Fp2 em Fp
def trace(x):       # função linear que mapeia Fp2  em  Fp
    return x + x^p

# Geração das curvas
E1 = EllipticCurve(Fp, [0,1])

# a curva supersingular sobre Fp  definida pela equação  y^2 = x^3 + a * x + b
E2 = EllipticCurve(Fp2, [0,1])

print(E2.is_supersingular())

# GrupoG = {n * Gerador | 0 < n < q}  # gerador de ordem "q" em E2
# Gerador = cofac * P
# cofac = (p + 1)//q

# ponto arbitrário  de ordem "q" em E2
P = E2.random_point() # E2.random_point() é um ponto arbitrário em E2
cofac = (p + 1)//q # cofactor de E2
G = cofac * P # gerador de ordem "q" em E2

identidade = b"Antonio Silva <asilva@qualquer.sitio> # 2024/12/31 23:59 #␣
↪read,write"
```

```
True
```

KeyGen(λ)

Gera um segredo administrativo s e uma chave pública administrativa β (beta)

[ ]:
```
# emparelhamento e oraculo DDHP

def phi(P):             # a isogenia que mapeia  (x,y)  ->  (z*x,y)
    (x,y) = P.xy()
    return E2(z*x,y)

def TateX(P,Q,l=1):      # o emparelhamento de Tate generalizado
    return P.tate_pairing(phi(Q), q, 2)^l
```

```
def ddhp(P,Q,R):        # o oraculo DDHP  que decide se (P,Q,R) é um triplo de␣
↪DH
    return TateX(P,Q) == TateX(R,G)

def Zr(q):
    s = ZZ.random_element(0, q-1)  # Generate a random integer in Zq (0...q-1)
    return s

def g(n):
    return int(n) * G # Grupo de torção G de ordem q em E2

def KeyGen(q):
    # Generate a secret key s
    s = Zr(q)  # Generate a random integer in Zq (0...q-1)

    # Compute the public key beta
    beta = g(s)  # Compute s * G

    return s, beta
```

KeyExtract(d)

Extração da chave privada *key* associada à chave pública d

[ ]:
```
def h(bytes):
    int_val = int.from_bytes(bytes, "little")
    return int_val

def ID(identidade):
    return g(h(identidade))

def KeyExtract(id):
    return s * id
```

Encrypt(d, x)

Recebe a chave pública d e o "plaintext" $x \in \mathbb{Z}$ e devolve o criptograma *criptograma*.

[ ]:
```
def Xor(a,b):
    int_a = int(a)
    int_b = int(b)
    return int_a ^^ int_b

# função de hash Z -> Zq
def H(int):
    return int % q
```

```python
# função de conversao Fp2 -> Z
def f(x):
    return x[0]

def input_E(d,x):
    v = Zr(q)
    a = H(Xor(v,x))
    u = TateX(beta, d, a)
    return (x,v,a,u)

def output_E(x,v,a,u):
    alfa = g(a)
    v_ = Xor(v,f(u))
    x_ = Xor(x,H(v)) # qual a utilidade do H?
    return (alfa,v_,x_)

def Encrypt(d,x):
    (x,v,a,u) = input_E(d,x)
    (alfa, v_, x_) = output_E(x,v,a,u)
    # build criptograma from alfa, v_, x_
    criptograma = (alfa, v_, x_)
    return criptograma
```

Decrypt

Usa a chave privada key, obtida do algoritmo Extract, e o criptograma $criptograma \equiv \langle \alpha\,,\, v'\,,\, x' \rangle$ para recuperar a mensagem original x.

[ ]:
```python
def input_D(key, alfa, v_, x_):
    u = TateX(alfa,key,1)
    v = Xor(v_, f(u))
    x = Xor(x_, H(v))
    return (alfa,v,x)

def output_D(alfa, v, x):
    a = H(Xor(v,x))
    if alfa != g(a):
        return None
    return x

def Decrypt(key, criptograma):
    (alfa, v_, x_) = criptograma
    (alfa,v,x) = input_D(key, alfa, v_, x_)
    x = output_D(alfa,v,x)
    if x is None:
        print("Decryption failed")
```

```
    return x
```

1.0.3 Teste

```
[ ]: s, beta = KeyGen(bp)
     print("s=", s, " beta=", beta)

     d = ID(identidade)
     print("d=",d)

     key = KeyExtract(d)
     print("key=",key)

     x = 1234
     criptograma = Encrypt(d, x)
     print("criptograma=",criptograma)

     plaintext = Decrypt(key, criptograma)
     print("plaintext=", plaintext)
```

```
s= 280  beta= (14323924494197112528435475819170941666442973444497239882842342672
84454048454970713143518832635497980864592250480904320266844511930278413066128836
22752728935*z + 2361266473384154606418303519892461897970535771827505443667080675
51816829521544085450731509650604699361101338149059127240856669692192328662727701
9771530878 : 1326588506550578635090955424596678740244112763613209102457136369560
73999443796700261513041440414929289917334205159606837609359412658337607881147510
900432972*z + 109351181324348341876387288859149085695230495821733201468513475086
25370106881982262680741954748079456256487385649830942929613898812578345642193952
2789903482 : 1)
d= (3821006035238874104466660430001962597910158546361708335912475987986881364767
9639917159541682415312508643065774268527318481167770856416919153871240068946646*
z + 1460988109016624024575893788104110314260560661680630916861515056204610383363
28187952491944972935102001053483242715817535136910902835266220918194851334289099
: 145650500509787967646276798412585707155035805919373751802976806927839461628509
252054166151924541054765947218534254921956844751849624912645953184415243337957*z
+ 877124811008634558871246044834304970102566087320987381662297442740819326384346
09567262137092023887798182166862543761152674725017790506053470762803989950872 :
1)
key= (36672450298903471885004302119431269059306691901178113071971590158214644901
37528651723269358960413564484044209065683386395201142288302010562743314232600913
8*z + 70205519843125796065651871695386238150280158756633043187862575888982981221
49549706418982190226998380888824067560410002338825705775373798227998432056277294
9 : 1586056846405651397310171526852202341374776725642807026683713439214287396346
76012499825624442028525354701002199012831724786950812008863361054132916964847121
*z + 14617020317754616616980108690033207381065043491568607192051003749129259220
71788541173049010755683850927478044318631727542368809458051883275514167596249956
1 : 1)
criptograma= ((14805456841973938239697389112702773099713437336246276162540389671
```

```
11953400632256861632519408232804886775368511413329459720444719721650707078533162
95574447067*z + 1311927459954098097243050563758439108479782859956721171097273366
31846714566033212651196927009068940876284031576798288937767815979072675723765329
51366984477 : 647121895313932279933006965093342390870087201342418454743220626642
24365091469879917323556597147284822865752489200117625597950758931186392071926606
041709705*z + 137663524850734850692724159652286397836198032145777822681068105324
33163511098721634662900300670201699754279260146923228498063733291449966090652332
0706918540 : 1), 101757702952793885480786013607544513429956007435897818820300204
49718578862077952372790879313440144616869666341874387646478804363034825895319473
081081460983, 558682282055185747099024580299133157270462671080)
plaintext= 1234
```