Ex3

April 2, 2024

1 Trabalho Prático 2

André Freitas PG54707

Bruna Macieira PG54467

1.1 Exercício 3

O algoritmo de Boneh e Franklin (BF) discutido no +Capítulo 5b: Curvas Elípticas e sua Aritmética é uma técnica 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.

[ ]:
```
%pip install pycrypto
%pip install sagemath-standard
import math
from sage.all import *
from Crypto.Hash import SHA512
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pycrypto in
/home/fura/.sage/local/lib/python3.10/site-packages (2.6.1)
Note: you may need to restart the kernel to use updated packages.
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: sagemath-standard in /usr/lib/python3/dist-
packages (9.5)
Requirement already satisfied: cysignals>=1.10.2 in
/home/fura/.sage/local/lib/python3.10/site-packages (from sagemath-standard)
(1.11.4)
Note: you may need to restart the kernel to use updated packages.
```

Esta classe contém métodos para implementar o esquema de criptografia IBE proposto por Boneh e Franklin, que permite a criptografia e descriptografia de mensagens utilizando identificadores de utilizadores, sem a necessidade de uma infraestrutura de chaves públicas tradicional.

1. Geração de chaves: o administrador do sistema gera um par de chaves pública e privada mestre. A chave pública mestre é usada para criptografar mensagens, enquanto a chave privada mestre é usada para derivar chaves privadas para utilizadores específicos.

2. Extração de Chave Privada: para um utilizador específico, a sua chave privada é extraída do par de chaves mestre utilizando o seu identificador único (por exemplo, endereço de e-mail, nome, entre outros). Esta extração de chave privada é feita de forma segura e eficiente, garantindo que apenas o utilizador autorizado possa aceder à sua chave privada.

3. Criptografia: para enviar uma mensagem para um utilizador específico, o remetente utiliza o identificador do destinatário para derivar uma chave pública correspondente. A mensagem é criptografada utilizando a chave pública do destinatário.

4. Descriptografia: o destinatário recebe a mensagem criptografada e utiliza a sua chave privada correspondente para descriptografá-la e recuperar o conteúdo original. A descriptografia é realizada de forma segura, garantindo que apenas o destinatário autorizado pode aceder à mensagem original.

5. Segurança: o esquema de Boneh e Franklin é projetado para ser seguro contra ataques criptográficos, como a obtenção não autorizada de chaves privadas ou a violação do sistema de criptografia. Aproveita propriedades matemáticas avançadas, como as curvas elípticas e os emparelhamentos de Weil, para garantir a segurança dos dados.

[ ]:
```
class BonehFranklinIBE:

    # recebe uma entrada e calcula o hash SHA-512 dessa entrada, retornando o␣
↪resultado como um número inteiro
    def Hash(hash_input):
        hash = SHA512.new()
        str_val = str(hash_input)
        byte_val = str_val.encode()
        hash.update(byte_val)
        hexadecimal = hash.hexdigest()
        return int(hexadecimal, 16)

    #Fast modular exponentiation algorithm (needed for F_{p^2})
    def fastPowerSmall(g,A):
        a = g
        b = 1
        if g == 0 and A == 0:
            return "Undefined"
        else:
            while A > 0:
                if A%2 == 1:
                    b = b*a
                a = a^2
                A = A//2
                print
            return b

    #converte string em inteiro
    def textToInt(w):
        n  = 0
```

```
    counter = 0
    #counter will give us the index of each character in the string
    for i in w:
        n = n + ord(i)*(256**counter)
        counter = counter + 1
    return n

#converte inteiro em string
def intToText(n):
    str = ""
    while n > 0:
        a0 = n%256
        str = str + chr(a0) #chr undoes ord. ord() inputs character and␣
↪outputs integer. str inputs integer between 0 and 255 and outputs character.
        n = n//256 #This is the quotient after dividing n by 256
    return str

#realiza o xor (exclusivo) entre dois inteiros
def Xor(a, b):
    int_a = int(a)
    int_b = int(b)
    return int_a ^^ int_b

# verifica se um elemento está no campo primo especificado
def is_in_prime_field(element, prime):
    return 0 <= element < prime

#This defines a rational function g(x,y) on E whose divisor is div(g) = [P]␣
↪+ [Q] - [P+Q] - [O]

#A, B coefficients of E. Use A,B = E.a_invariants()[3], E.a_invariants()[4]
#P = E([xP, yP])

def g(P,Q,x,y,E):
    positive_infinity = math.inf
    A,B = E.a_invariants()[3], E.a_invariants()[4]
    if P == E(0) or Q == E(0):
        return "no divisor"
    xP,yP = P[0],P[1]
    xQ,yQ = Q[0],Q[1]
    #Calculate slope of line connecting P and Q
    #JUST check if equal
    if yP == -yQ and xP == xQ:
        slope = +positive_infinity #symbol for Infinity
    elif P == Q:
        slope = (3*(xP**2) + A)/(2*yP)
    else:
```

```
            slope = (yQ - yP)/(xQ - xP)
        #return the function on E
        if slope == +positive_infinity:
            return x - xP
        else:
            return (y - yP - slope*(x - xP))/(x + xP + xQ - slope**2)

    #  implementa o algoritmo de Miller para calcular um emparelhamento de Weil
    def MillerAlgorithm(P,m,x,y,E):
        A,B = E.a_invariants()[3], E.a_invariants()[4]
        xP,yP = P[0],P[1]
        binary = m.digits(2) #gives number in binary
        n = len(binary) #trying to find what "n" is.
        T = P
        f = 1
        for i in range(n-2,-1,-1): #Stop once i = -1, so last number is 0...
↪range(start, stop, step)
            f = (f**2)*BonehFranklinIBE.g(T,T,x,y,E)
            T *= 2 #T = 2T
            if binary[i] == 1:
                f = f*BonehFranklinIBE.g(T,P,x,y,E)
                T += P
        return f

    # calcula o emparelhamento de Weil entre dois pontos em uma curva elíptica
    def WeilPairing(P,Q,m,E):
        A,B = E.a_invariants()[3], E.a_invariants()[4]
        S = E.random_element()
        while m*S == E(0):
            S = E.random_element() #Pick point S that is not m-torsion. This␣
↪guarantees that S isn't a linear combination of P and Q.
        xS,yS = S[0],S[1]
        QplusS = Q + S
        f_P_QplusS = BonehFranklinIBE.MillerAlgorithm(P,m,QplusS[0],QplusS[1],E)
        f_P_S = BonehFranklinIBE.MillerAlgorithm(P,m,xS,yS,E)
        num = f_P_QplusS/f_P_S

        PminusS = P - S
        f_Q_PminusS = BonehFranklinIBE.
↪MillerAlgorithm(Q,m,PminusS[0],PminusS[1],E) #This is f_Q(P-S)
        f_Q_minusS = BonehFranklinIBE.MillerAlgorithm(Q,m,xS,-yS,E) #This is␣
↪f_Q(-S)
        denom = f_Q_PminusS/f_Q_minusS

        e_m = num/denom
        return e_m
```

```
    # calcula um emparelhamento de Weil modificado, usado no esquema de␣
↪criptografia
    def ModifiedWeilPairing(P,Q,m,E):
        Fp = GF(p)
        R.<x> = Fp[]
        Fp2.<z> = Fp.extension(x^2+x+1)      #Form F_{p^2} by adjoining z = {a␣
↪nontrivial cube root of 1}
        E_zeta = EllipticCurve(Fp2, [0,1])   #Define E: y^2 = x^3 + 1 over this␣
↪field
        phiQ = E_zeta([z*Q[0],Q[1]])
        A,B = E_zeta.a_invariants()[3], E_zeta.a_invariants()[4]
        P_zeta = E_zeta([P[0],P[1]])
        S = E_zeta.random_element()
        while m*S == E(0):
            S = E_zeta.random_element()
        xS,yS = S[0],S[1]
        QplusS = phiQ + S
        f_P_QplusS = BonehFranklinIBE.
↪MillerAlgorithm(P,m,QplusS[0],QplusS[1],E_zeta)
        f_P_S = BonehFranklinIBE.MillerAlgorithm(P,m,xS,yS,E_zeta)
        num = f_P_QplusS/f_P_S

        PminusS = P_zeta - S #modify
        f_Q_PminusS = BonehFranklinIBE.
↪MillerAlgorithm(phiQ,m,PminusS[0],PminusS[1],E_zeta) #This is f_Q(P-S)
        f_Q_minusS = BonehFranklinIBE.MillerAlgorithm(phiQ,m,xS,-yS,E_zeta)␣
↪#This is f_Q(-S)
        denom = f_Q_PminusS/f_Q_minusS

        e_m = num/denom
        return e_m

    # gera as chaves pública e privada
    def KeyGen(k):
        q = random_prime((2^k) - 1, True, lbound=2^(k-1)) #Generates a random␣
↪k-bit prime. False means using pseudo-primality tests.
        p = q
        l = 1 #need l for `KeyExtract`
        lq = q
        while True:
            l += 1
            lq += q
            p = lq - 1
            if p%3 == 2 and (p+1)%(q^2) != 0 and is_prime(p) == True:
                break
```

```
        E = EllipticCurve(GF(p), [0,1]) #p is public b/c the elliptic curve is␣
↪known
        P = E(0)
        while P == E(0):
            Q = E.random_element()
            while Q == E(0): #make sure P is not 0
                Q = E.random_element()
            h = (p+1)//q #This is to make sure P has order q.
            P = h*Q #Order of P is q1
        s = ZZ.random_element(2,q-1) #s is private master key in Z_q*.
        P_pub = s*P
        params = [p, q, l, E, P, P_pub]
        return params, s

    # extrai a chave privada correspondente a uma identidade
    def KeyExtract(y0, params):
        p, q, l, E, P, P_pub = params
        x0 = pow((y0^2) - 1, ((2*p)-1)//3, p) #`pow` is Python's built-in␣
↪function that does fast power
        Q = E([x0,y0])
        Q_ID = l*Q #l comes from BDHGenerator. It's the integer s.t. p = lq-1
        d_ID = s*Q_ID
        return Q_ID,d_ID

    def Encrypt(M,Q_ID,params):
        params = [p, q, l, E, P, P_pub]
        r = ZZ.random_element(2,q-1)
        U = r*P
        g_ID = BonehFranklinIBE.ModifiedWeilPairing(Q_ID,P_pub,q,E)
        g_ID_to_r = BonehFranklinIBE.fastPowerSmall(g_ID, r)
        V = M^^BonehFranklinIBE.Hash(g_ID_to_r)
        C = [U,V] #This is the ciphertext
        return C

    def Decrypt(C,d_ID,params):
        params = [p, q, l, E, P, P_pub]
        U,V = C
        weil = BonehFranklinIBE.ModifiedWeilPairing(d_ID,U,q,E)
        M = V^^BonehFranklinIBE.Hash(weil)
        return M
```

[ ]:

```
#Step 1: Only need to run once
params,s = BonehFranklinIBE.KeyGen(512)
p, q, l, E, P, P_pub = params
print("p = ", p)
print("q = ", q)
print("E = ", E)
```

```python
print("P = ", P)
print("s = ", s)
print("l = ", l)
print("P_pub = ", P_pub)
```

```
p =  1909008715499957959243823149698239010586755857174200283448979911268511242307978036246963305049312170423922844687402124514517121985451065480095402312248410 8457
```

```
p =  190900871549995795924382314969823901058675585717420028344897991126851124230
797803624696330504931217042392284468740212451451712198545106548009540231224841 08
457
q =  128813003744936434496884153151028273318944389822820531946624825321761892193
520785171859872135581118112275495592942113664947174223039882960870135108788691 69
E =  Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 190900871
549995795924382314969823901058675585717420028344897991126851124230797803624696 33
050493121704239228446874021245145171219854510654800954023122484108457
P =  (179195505693028056777290063550468969203146193962324146661047753804055071 86
268419165313161682125555522517077870039263806232543588624121081257646316010179 68
063 : 114657201623157039034638908242067340183295377889324046007668228819371405 34
134877937577588606143137810155960326668650571955802444992438148492698649869237 54
4767 : 1)
s =  369879485350370971665237527792449212393254359495659097358788888862693639746
03817539770451554113403450897677177560083439726426991879667328444675929065536 63
l =  1482
P_pub =  (181314955701655892846350234427413288820345155763139194668359362920857 3
393399954019037056332598292829298130028912838298518290470258984602896990567394 98
26281269 : 116205292794725095927504699250236319235602355088070615714498510797703
692009197504904197637049207973147045517378167028942248207713974166485692488956 40
304115506 : 1)
```

[ ]:
```python
#Step 2: Hash. Only need to run once for one ID.
your_ID = "user"

y0 = BonehFranklinIBE.Hash(your_ID)
print("y0 = ", y0)
```

```
y0 =  928402724471147686873657471119918533213378296723704414815372759602787506 72
788621655196332115300084529514486444270859303135371338304415246399775148665807 06
```

[ ]:
```python
#Step 3: Only need to run once.
Q_ID,d_ID = BonehFranklinIBE.KeyExtract(y0,params)
print("Q_ID = ", Q_ID)
print("d_ID = ", d_ID)
```

```
Q_ID =  (103101200851230958583351142731741998428972385351419527940309761817203 92
787778222581171411179413073394774873142299651012151382608471863691027655210588 82
7925033 : 522546950969026954345053136725180051920321221538884289712210438534049 9
082832228927765538033176396342654511153860728248629265097269244776339368928531 55
8478162 : 1)
d_ID =  (822407741351757621581069862015256497684943994982582801789471743129406 97
975489485277614493866625680235090915746020967215943645164262280379148067217941 53
```

```
853603 : 672387805765477264232207089823848267114977399617105340961634103258926608
8458354549915844102542187244866013374800475066791614521701669097820378581641241
1056692 : 1)
```

[ ]:
```python
#Step 4: Encrypt
input_message = "Teste"

M = BonehFranklinIBE.textToInt(input_message)
C = BonehFranklinIBE.Encrypt(M,Q_ID,params)
U,V = C
print("M = ", M)
print("U = ", U)
print("V = ", V)
```

```
M =  435745416532
U =  (19079965039666007035187582082681337012220401378160556659210561865024213962
830529489368878217300722565266965869774661129479485412026102404998943951026207883
3354 : 144101147274934932869894678656501348177937932490807612587641832553036046764
9424643617646645202207854118912164636902219465528059800527571853208745954223706438 : 1)
V =  121799022365727769073137513170007797824841784910545578013991191864294497729
28440817284177619561842281182842642114092456406784128284253571873057088658385730
```

[ ]:
```python
#Step 5: Decrypt
M0 = BonehFranklinIBE.Decrypt(C,d_ID,params)
output_message = BonehFranklinIBE.intToText(M0)
print("M0 = ", M0)
print("messsage = ", output_message)
```

```
M0 =  435745416532
messsage =  Teste
```