TP2 - Ex.3

Abril 1, 2024

Estruturas Criptográficas

PG53886, Ivo Miguel Alves Ribeiro

A95323, Henrique Ribeiro Fernandes

In [ ]:
```
from sage.all import *
from aux_func import *
```

In [ ]:
```
class BonehFranklinCryptosystem:
    def __init__(self):
        pass

    def BFsetup(self, version, n):
        if n not in [1024, 2048, 3072, 7680, 15360]:
            raise ValueError("Invalid security parameter")

        if version != 2:
            raise ValueError("Invalid version number")
        else:
            public_params,s= self.BFsetup1(n)

        return public_params,s

    def BFsetup1(self,n):
        #1
        version = 2

        #2
        if n == 1024:
            n_p, n_q, hashfcn = 512, 160, "1.3.14.3.2.26"  # SHA-1
        elif n == 2048:
            n_p, n_q, hashfcn = 1024, 224, "2.16.840.1.101.3.4.2.4"  # SHA-224
        elif n == 3072:
            n_p, n_q, hashfcn = 1536, 256, "2.16.840.1.101.3.4.2.1"  # SHA-256
        elif n == 7680:
            n_p, n_q, hashfcn = 3840, 384, "2.16.840.1.101.3.4.2.2"  # SHA-384
        elif n == 15360:
            n_p, n_q, hashfcn = 7680, 512, "2.16.840.1.101.3.4.2.3"  # SHA-512

        while True:
            #3
            while True:
                # Escolha aleatória de a e b
                #a = randint(1, n_q-1)
                #b = randint(1, n_q-1)

                #q = 2**a + 2**b + 1
                q = Integer(1393796574908163946345982391759047617413119)
                if q.is_prime() and q < 2**n_q:
                    for r in range(1, (2 ** n_p)//(12*q)):
                        # Calcular p
                        p = 12 * r * q - 1

                        # Verificar se p é primo e menor que 2**n_p
                        if p < 2**n_p and p.is_prime():
```

```
                            break

                    if (12*r*q-1 ==p) and p < 2**n_p and p.is_prime():
                        break
                #4
                F_P = GF(p) # y^2 = x^3 + 1
                a = 0
                b = 1
                E = EllipticCurve(F_P, [a, b])

                point =E.random_element()
                P_prime = (point[0], point[1])
                P = 12 * r * E(P_prime)

                if P!= 0:
                    break

            #5
            s = randint(2, q - 1)
            P_pub = s * P

            return {'version': version,'E': E , 'p': p, 'q': q, 'P': P, 'P_pub': P_pu
        ##direita
        def BFderivePubl(self, id, public_params):
            E, p, q, hashfcn = public_params['E'], public_params['p'], public_params[
            hashfcn = Hashfunc(hashfcn)
            Q_id = HashToPoint(E, p, q, id, hashfcn)
            return Q_id
        ###direita
        def BFextractPriv(self, id, public_params, s):
            Q_id = self.BFderivePubl(id, public_params)
            return s * Q_id

        def BFencrypt(self, m, id, public_params):
            E, p, q, P, P_pub, hashfcn = public_params['E'], public_params['p'], publ

            #1
            hashfcn = Hashfunc(hashfcn)
            hashlen = hashfcn().digest_size
            #2
            Q_id =  self.BFderivePubl(id, public_params)
            #3
            rho = bytes([randint(0, 255) for _ in range(hashlen)])
            #4
            t = hashfcn(m).digest()
            #5
            l = HashToRange(rho+t, q, hashfcn)
            if len(rho+t) != 2*hashlen :
                raise ValueError("The concatenation of rho and t most have (2 * hashl
            if l < 0 or l > q -1:
                raise ValueError("Invalid value l in Encryption:"+str(l))
            #6
            U = l * P
            #7
            theta = pairing(E, p, q, P_pub, Q_id)
            #8
            theta_prime = theta**l
            #9
            z = canonical_encoding(E,p, None, 0,theta_prime)
            #10
            w = hashfcn(z).digest()
            #11
            V = bytes([(a).__xor__(b) for a, b in zip(w, rho)])
            #12
            W = bytes([(a).__xor__(b) for a, b in zip(HashBytes(len(m),rho,hashfcn),
```

```
        return U, V, W

    def BFdecrypt(self, S_id, ciphertext, public_params):
        E, p, q, P, P_pub, hashfcn = public_params['E'], public_params['p'], publ
        U, V, W = ciphertext
        #1
        hashfcn = Hashfunc(hashfcn)
        hashlen = hashfcn().digest_size
        #2
        theta = pairing(E, p, q, U, S_id)
        #3
        z = canonical_encoding(E,p, None, 0,theta)
        #4
        w = hashfcn(z).digest()
        #5
        rho = bytes([(a).__xor__(b) for a, b in zip(w, V)])
        #5
        m = bytes([(a).__xor__(b) for a, b in zip(HashBytes(len(W),rho,hashfcn),
        #7
        t = hashfcn(m).digest()
        #8
        l = HashToRange(rho+t, q, hashfcn)
        #9
        if l * P != U:
            return "Invalid ciphertext"
        return m
```

teste

In [ ]:
```
def create_BF_cryptosystem():
    # Criando uma instância do criptossistema Boneh-Franklin
    bf_crypto = BonehFranklinCryptosystem()

    # Setup do criptossistema
    version = 2
    security_parameter = 1024
    public_params, master_secret = bf_crypto.BFsetup(version, security_parameter)
    # Gerando chaves pública e privada para uma identidade
    identity = "alice@example.com"
    public_key = bf_crypto.BFderivePubl(identity, public_params)
    private_key = bf_crypto.BFextractPriv(identity, public_params, master_secret)

    return bf_crypto,identity, public_params , private_key


bf_crypto,identity, public_params , private_key = create_BF_cryptosystem()
```

In [ ]:
```
def test_BF_cryptosystem(bf_crypto,identity, public_params , private_key):
    # Mensagem a ser criptografada
    message = b"Hello, world!"

    # Criptografando a mensagem para a identidade especificada
    ciphertext = bf_crypto.BFencrypt(message, identity, public_params)

    # Descriptografando a mensagem usando a chave privada correspondente
    decrypted_message = bf_crypto.BFdecrypt(private_key, ciphertext, public_param

    # Verificando se a mensagem descriptografada é igual à mensagem original
    assert decrypted_message == message, "Decryption failed!"

    print("Test passed successfully!")

test_BF_cryptosystem(bf_crypto,identity, public_params , private_key)
```

```
Test passed successfully!
```