Estruturas Criptográficas - TP2-3

PG53721 - Carlos Machado

PG54249 - Tiago Oliveira

Enunciado: Implementação do algoritmo de Boneh e Franklin (BF) com SageMath

In [1]:
```
from collections import namedtuple
import os
import hashlib
```

Definimos aqui funções de hash auxiliares que serão usadas, dando destaque à *hash_to_range*, que transforma um array de bytes para um nº abaixo de um limite.

In [2]:
```
def hash_to_range(s,n,hashfcn):

    # hashlen: the number of octets comprising the output of hashfcn
    hashlen = len(hashfcn(b''))

    v_i = 0
    h_i = b'\x00' * hashlen

    for i in range(1, 3):

        # concatenate h_(i-1) and s to form t_i
        t_i = h_i + s

        # hash of t_i
        h_i = hashfcn(t_i)

        # convert h_i to integer a_i
        a_i = int.from_bytes(h_i, 'big')

        # compute v_i
        v_i = 256 ** hashlen * v_i + a_i

    v = v_i % n

    return v

def sha1(v):
    return hashlib.sha1(v).digest()

def sha224(v):
    return hashlib.sha3_224(v).digest()

def sha256(v):
    return hashlib.sha3_256(v).digest()

def sha384(v):
    return hashlib.sha3_384(v).digest()

def sha512(v):
    return hashlib.sha3_512(v).digest()
```

Definimos aqui os parâmetros de segurança para Boneh-Franklin, tal como estavam no RFC 5091.

In [3]:
```
BFSecParam = namedtuple('SecParam', 'bq bp hashfn')

class BFSecParams:
    n1024 = BFSecParam(160,512,sha1)
    n2048 = BFSecParam(224,1024,sha224)
```

```
    n3072 = BFSecParam(256,1536,sha256)
    n7680 = BFSecParam(384,3840,sha384)
    n15360 = BFSecParam(512,7680,sha512)
```

Definimos aqui a classe que representa o grupo de torção numa curva elíptica que será usado no algoritmo de Boneh-Franklin.

Dados os parâmetros de segurança, é gerado um nº primo *q* e um nº primo *p* múltiplo de 3 tal que *q* é divisor de *p+1*.

São depois gerados dois corpos finitos *Fp* e *Fp^2*, as curvas *E1 = E/Fp* e *E2 = E/Fp^2*, e um grupo de torção *G* de ordem *q* em *E2*.

É também declarado a função de emparelhamento generalizado *tateX*

In [4]:
```
class BFGroup:
    def __init__(self, params):
        bq = params.bq
        bp = params.bp

        q = random_prime(2^bq-1,lbound=2^(bq-1))

        t = q*3*2^(bp - bq)
        while not is_prime(t-1):
            t = t << 1

        p = t - 1

        # Aneis e Corpos
        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)

        # Curvas Elipticas supersingulares em Sagemath

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

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

        self.G = G
        self.bq = params.bq
        self.bp = params.bp
        self.hashfn = params.hashfn
        self.E2 = E2
        self.Fp2 = Fp2
        self.Fp = Fp
        self.q = q
        self.p = p
        self.z = z

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

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

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

In [5]:
```
sg = BFGroup(BFSecParams.n2048)
```

Definimos aqui a classe que implementa o criptosistema Boneh-Franklin.

Ao instanciar a classe, esta gera uma chave privada administrativa e uma chave pública administrativa no grupo recebido, que serão usadas no algoritmo.

Define os métodos:

- **encrypt**: cifra uma mensagem com a chave pública
- **key_extract**: gera uma chave privada com a chave pública
- **decrypt**: decifra um criptograma com a chave privada

In [6]:
```
class BF:

    def __init__(self, subgroup):

        s = hash_to_range(os.urandom(subgroup.bq//8), subgroup.q-1, subgroup.hashfn)
        G_pub = s*subgroup.G

        self.s = s
        self.public = G_pub
        self.sg = subgroup


    def _ID(self, id):
        return int.from_bytes(self.sg.hashfn(id),'little') * self.sg.G

    def _Zr(self):
        return hash_to_range(os.urandom(self.sg.bq//8),self.sg.q-1,self.sg.hashfn)

    def _H(self,v):
        vb = int(v).to_bytes(self.sg.bq//8,'little')
        return hash_to_range(vb,self.sg.q-1,self.sg.hashfn)


    def key_extract(self, id):
        d = self._ID(id)
        return self.s*d

    def encrypt(self,id,x):
        def _in(self,id,x):
            d = self._ID(id)
            v = self._Zr()
            a = self._H(v^^xi)
            u = self.sg.tateX(self.public,d,a)
            return (xi,v,a,u)

        def _out(self,x,v,a,u):
            alpha = a*self.sg.G
            vl = int(v)^^int(self.sg.trace(u))
            xl = x^^self._H(v)
            return (alpha,vl,xl)

        xi = int.from_bytes(x,'little')
        x,v,a,u = _in(self,id,xi)
        return _out(self,x,v,a,u)

    def decrypt(self,key,c):
        def _in(self,key,alpha,vl,xl):
            u = self.sg.tateX(alpha,key,1)
            v = int(vl)^^int(self.sg.trace(u))
            vh = self._H(v)
            x = xl^^vh
            return alpha,v,x

        def _out(self,alpha,v,x):
            a = self._H(v^^x)
            if alpha != a*self.sg.G:
                raise Exception('failed decryption')
            else:
                return x
```

```
        alpha,vl,xl = c
        alpha,v,x = _in(self,key,alpha,vl,xl)
        x = _out(self,alpha,v,x)
        return int(x).to_bytes(len(msg),'little')
```

In [7]:
```
bf = BF(sg)
```

Teste - cifragem e decifragem

In [8]:
```
id = b'sallybad'
msg = b'yepyep'

c = bf.encrypt(id,msg)

s_key = bf.key_extract(id)
xmsg = bf.decrypt(s_key,c)

print('are messages equal?', xmsg == msg)
assert xmsg == msg
```

```
are messages equal? True
```

In [ ]:
```

```