Exercício 3 - BF

April 2, 2024

1 Estruturas Criptográficas - Criptografia e Segurança da Informação

Grupo 03

(PG54177) Ricardo Alves Oliveira

(PG54236) Simão Oliveira Alvim Barroso

1.1 TP3 - Exercício 3

3. 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.

Ao longo deste trabalho, utilizamos como fonte para o desenvolvimento o `RFC5091` e as seguintes secções dos apontamentos da UC:

- Criptosistema de Boneh-Franklin
- IBE : "Pairing Based Encryption"
- Emparelhamentos

1.2 Resolução

Para resolver este exercício começamos por importar a biblioteca `sagemath`.

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

# variaveis globais (explicadas em baixo)
q = None
p = None
f = None
G = None
z = None
s = None
beta = None
E2 = None
nonce = 54345264
```

Um emparelhamento tate definido num grupo de torção $\mathbb{G}$ de ordem prima q como vimos na secção de emparelhamentos. O grupo $\mathbb{G}$ tem o gerador G.

1.2.1 Emparelhamentos e "Pairing Based Encryption"

De seguida definimos as funções auxiliares do criptosistema com recurso ao sagemath seguindo as orientações abaixo (nome das funções, domínios e contradomínios):

$$\mathsf{Zr} : \mathbb{N} \to \mathbb{Z}_q$$

$$\begin{aligned} f &: \mathbb{F}_{p^2} \to \mathbb{Z} \\ h &: \mathsf{Bytes} \to \mathbb{Z} \\ H &: \mathbb{Z} \to \mathbb{Z}_q \\ g &: \mathbb{Z} \to \mathbb{G} \\ \mathsf{ID} &: \mathsf{Bytes} \to \mathbb{G} \end{aligned}$$

A função `Zr` é um gerador pseudo-aleatório de inteiros no intervalo $[0, q-1]$. Utiliza a função `set_random_seed` do sagemath com esse intuito(gerador pseudo-aleatório (PRG) que toma como argumento um "nounce" n e produz um inteiro no intervalo $\mathbb{Z}_q$).

A função `f_hash` e a função `h` são funções de hash que convertem os seus tipos de dados do dominio em inteiros, utilizando a função hash do sagemath.

Já a função de Hash `H` tem como dominio os inteiros e como contradomínio os inteiros módulo q.

A função `g` tem definida como $g : n \mapsto n * G$, assim como descrita nos apontamentos da UC.

A função `ID` está feita como descrita nos apontamentos da UC definida como $\mathsf{ID}(m) \equiv g(h(m))$

A função `phi` e a função `TateX` são utilizadas para o emparelhamento de tate. Esta função realizará, no que toca ao criptosistema, o papel de `ex` e é responsável pelo emparelhamento generalizado.

[ ]:
```
# Gerador pseudo-aleatório
def Zr():
    set_random_seed(nonce)
    random_number = randint(0, q - 1)
    return random_number

# Função de hash f
def f_hash(data):
    return hash(data)

# Função de hash h
def h(bytes_seq):
    return hash(bytes_seq)

# Função de hash H
def H(integer):
```

```
    return hash(integer)%q

# Função g
def g(n):
    return n * G

# Função ID
def ID(m):
    return g(h(m))

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
```

1.2.2 Criptosistema de Boneh-Franklin

A função $\mathsf{KeyGen}(\lambda)$, em baixo codificada e explicada, foi a primeira função que desenvolvemos das necessárias para a implementação do criptosistema de Boneh-Franklin.

Aqui em baixo está o descrito no apontamento da UC, que fizemos:

Este algoritmo gera um segredo administrativo s e uma chave pública administrativa β a partir de um parâmetro de segurança λ.

```
- Gera os elementos comuns com q 2 .
- Gera :
```

$$\begin{cases} s & \leftarrow & \mathsf{Zr} \\ \beta & \leftarrow & g(s) \end{cases}$$

Em tudo o que se segue é público o representante da identidade do receptor $id \in \mathsf{Bytes}$ e equivalentemente a sua descrição em $\mathbb{G}$ calculada como $d \leftarrow \mathsf{ID}(id)$.

É de relembrar que para além disso utilizamos curvas eliticas para calcular G (grupo de torção).

[ ]:
```
def KeyGen(lbd):
    global q, p, f, G, z, s, beta, E2
    lbd = 8
    bq = 2^(lbd-1)
    bp = 2^lbd-1
    q = random_prime(bp,lbound=bq)

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

    p = t - 1
```

```
    Fp      = GF(p)
    R.<z>   = Fp[]
    f      = R(z^2 + z + 1)
    Fp2.<z> = GF(p^2, modulus=f)


    E2 = EllipticCurve(Fp2, [0,1])

    cofac = (p + 1)//q
    G = cofac * E2.random_point()

    s = Zr()
    beta = g(s)

    return (s, beta)
```

Abaixo encontramos alguns exemplos do funcionamento normal de cada uma das funções definidas, incluindo a função `KeyGen` do criptosistema.

[ ]:
```
print("KeyGen(8):", KeyGen(8))
print("Zr:", Zr())
print("f(120):", f_hash(120))
print("h(b'120'):", h(b'120'))
print("H(1209):", H(1209))
print("g(7879804275796629580):", g(7879804275796629580))
print("ID(b'120'):", ID(b'120'))
```

```
KeyGen(8): (43, (2273509137437079069073044363616197667795552*z +
2586539837687654800395197499527599141963365 :
2601031599955598367050358798398131368305268*z +
3666844326123734211351882443400913088361704 : 1))
Zr: 43
f(120): 120
h(b'120'): -6704737720330644928
H(1209): 44
g(7879804275796629580): (447862481297099378476202810527519997145296*z +
1461048300905744039827616491204872797300801 :
3250663916209159572589715430952517634635430*z +
813324317439946188892337006353278162771996 : 1)
ID(b'120'): (1838468207785596571188274807642605336280829*z +
1015814752453841911732129789248543680971279 :
692066221863991911169549172785863290907791*z +
1266632313281915923121514064029698030698563 : 1))
```

Relativame ao keyExtract (algoritmo usa a informação de administração para extrair a chave privada *key* associada à chave pública *id*, é implementado como está enunciado no diagrama:

$$key \leftarrow (\vartheta\, d \leftarrow \mathsf{ID}(id)\,.\, s * d\,)$$

[ ]:
```
def KeyExtract(id):
    d = ID(id)
    key = s * d
    return key

key = KeyExtract(b'120')

print("KeyExtract(b'120'):",key)
```

```
KeyExtract(b'120'): (242328491312070938471230498263647530060281*z +
1372628945229091290119271217719971001182061 :
313249654328383345859482642652467216406237*z +
2195553208011767821144680133372618889417073 : 1)
```

No in_encrypt temos a seguinte definição:

$$\mathsf{in\_encrypt}(id, x) \;\equiv$$
$$\vartheta\, d \leftarrow \mathsf{ID}(id)\,.\,\vartheta\, v \leftarrow \mathsf{Zr}\,.\,\vartheta\, a \leftarrow H(v \oplus x)\,.\,\vartheta\, \mu \leftarrow \mathbf{ex}(\beta, d, a)\,.\,\langle x, v, a, \mu\rangle$$

[ ]:
```
def in_encrypt(id, x):
    d = ID(id)
    v = Zr()
    a = H(v ^^ x)
    mu = TateX(beta, d, a)
    return v, a, mu

v, a, mu = in_encrypt(b'120', 120)

# print para debug
print("in_encrypt(b'120', 120):", (v, a, mu))
```

```
in_encrypt(b'120', 120): (43, 83, 2629630024904420427856237800297019839180494*z
+ 1884864036726402927662965387476718460785243)
```

No out_encrypt temos de mencionar extra a utilização do ^^ para fazer xor, com recurso ao sagemath. Segue o diagrama apresentado em baixo:

$$\mathsf{out\_encrypt}(x, v, a, \mu) \;\equiv$$
$$\vartheta\, \alpha \leftarrow g(a)\,.\,\vartheta\, v' \leftarrow v \oplus f(\mu)\,.\, x' \leftarrow x \oplus H(v)\,.\,\langle \alpha, v', x'\rangle$$

[ ]:
```
def out_encrypt(x, v, a, mu):
    alpha = g(a)
    v_prime = v ^^ f_hash(mu)
    x_prime = x ^^ H(v)
```

```
    return (alpha, v_prime, x_prime)

# print para debug
print("out_encrypt(120,v,a,mu):", out_encrypt(120, v, a, mu))
```

```
out_encrypt(120,v,a,mu): ((847233263721477088871220453940206233687146*z +
2595127466026680811584227734898508292904506 :
2658283201752406190710119028625335768515877*z +
1052028975402982538101411525669900895040728 : 1), 7697624891466661929, 83)
```

Por sua vez, a função de Encrypt é uma composição das funções `in_encrypt` e `out_encrypt`:

$$\mathsf{Encrypt}(id, x) \;\equiv$$

$$\vartheta\, x, v, a, \mu \leftarrow \mathsf{in}(id, x) \centerdot \mathsf{out}(x, v, a, \mu)$$

```
[ ]: def Encrypt(id, x):
         v, a, mu = in_encrypt(id, x)
         alpha, v_prime, x_prime = out_encrypt(x, v, a, mu)
         return (alpha, v_prime, x_prime)

     alpha, v_prime, x_prime = Encrypt(b'120', 120)

     # print para debug
     print("Encrypt(b'120', 120):", (alpha, v_prime, x_prime))
```

```
Encrypt(b'120', 120): ((847233263721477088871220453940206233687146*z +
2595127466026680811584227734898508292904506 :
2658283201752406190710119028625335768515877*z +
1052028975402982538101411525669900895040728 : 1), 7697624891466661929, 83)
```

Na implementação do `in_decrypt` utilizamos surge a necessidade de utilizar a função `ex` para realizar o emparelhamento generalizado que, no nosso caso, recorre à função `TateX`:

$$\mathsf{in\_decrypt}(key, \alpha, v', x') \;\equiv$$
$$\vartheta\, \mu \leftarrow \mathbf{ex}(\alpha, key, 1) \centerdot \vartheta\, v \leftarrow v' \oplus f(\mu) \centerdot x \leftarrow x' \oplus H(v) \centerdot \langle \alpha, v, x \rangle$$

```
[ ]: def in_decrypt(key, alpha, v_prime, x_prime):
         mu = TateX(alpha,key,1)
         v = v_prime ^^ f_hash(mu)
         x = x_prime ^^ H(v)
         return (alpha, v, x)

     alpha, v, x = in_decrypt(key,alpha,v_prime,x_prime)
```

```
print("in_decrypt(key,alpha,v_prime,x_prime):", (alpha, v, x))
```

```
in_decrypt(key,alpha,v_prime,x_prime):
((8472332637214770888712204539402062336871 46*z +
```

A função `out_decrypt` é a função que realiza a desencriptação final do texto cifrado e a sua verificação, devolve o texto em limpo no caso de sucesso e `Decryption failed` no caso de falha.

$$\text{out\_decrypt}(\alpha, v, x) \equiv$$
$$\vartheta\, a \leftarrow H(v \oplus x) \,\centerdot\, \text{if}\ \ \alpha \neq g(a)\ \ \text{then}\ \ \texttt{fails}\ \ \text{else}\ \ x$$

[ ]:
```
def out_decrypt(alpha, v, x):
    a = H(v ^^ x)
    if g(a) != alpha:
        return "Decryption failed"
    return x

print("out_decrypt(alpha, v, x):", out_decrypt(alpha, v, x))
```

```
out_decrypt(alpha, v, x): 120
```

O `Decript`, mais uma vez, é uma composição das funções `in_decrypt` e `out_decrypt`:

$$\text{Decrypt}(\mathit{key}, c) \equiv$$
$$\vartheta\, \alpha, v, x \leftarrow \text{in}(\mathit{key}, c) \,\centerdot\, \text{out}(\alpha, v, x)$$

[ ]:
```
def Decrypt(key, alpha, v_prime, x_prime):
    alpha, v, x = in_decrypt(key, alpha, v_prime, x_prime)
    return out_decrypt(alpha, v, x)

print("Decrypt(key, alpha, v_prime, x_prime):", Decrypt(key, alpha, v_prime,␣
 ↪x_prime))
```

```
Decrypt(key, alpha, v_prime, x_prime): 120
```

Desde já, com os testes individuais de cada função, podemos concluir que o criptosistema de Boneh-Franklin está a funcionar corretamente e que as funções estão a ser executadas de acordo com o esperado. Isto pode ser verificado nos outputs de cada função do criptosistema e, de forma completa, na secção de testes que se segue.

1.2.3 Testes

Nesta secção temos alguns testes que fizemos para demonstrar o bom funcionamento do código.

Bem sucedido Primeiramente temos um teste bem sucedido, onde tudo acontece como deve. Começamos por criar um lambda, um id (decididimos um email exemplo) e uma mensagem, neste caso e para facilitar um numero inteiro.

Depois utilizamos o lamba para gerar um segredo administrativo e uma chave publica. Extraimos a chave priva pelo id. Ciframos e deciframos e mensagens deram iguais como era de esperar.

[ ]:
```
lmbda = 128
id = "sender@test.com"
x = 7655465

s, beta = KeyGen(lmbda)
key = KeyExtract(id)
alpha, v_prime, x_prime = Encrypt(id, x)
decrypted_x = Decrypt(key, alpha, v_prime, x_prime)
print("Original message:", x)
print("Decrypted message:", decrypted_x)
```

```
Original message: 7655465
Decrypted message: 7655465
```

Teste Falhado - Mudança do id Fizemos este teste para verificar que ao mudar o id e gerar uma nova key com esse id. Vemos que a decifração falha, como era de esperar.

[ ]:
```
lmbda = 128
id = "sender@test.com"

x = 7655465

s, beta = KeyGen(lmbda)
key = KeyExtract(id)
alpha, v_prime, x_prime = Encrypt(id, x)
id = "mudamos_o_id@vamosVerSeMuda.pt"
key = KeyExtract(id)
decrypted_x = Decrypt(key, alpha, v_prime, x_prime)
print("Original message:", x)
print("Decrypted message:", decrypted_x)
```

```
Original message: 7655465
Decrypted message: Decryption failed
```