MIEI/MI - Estruturas Criptográficas

Trabalho Prático 1

João Alves
a77070@alunos.uminho.pt

Nuno Leite
a70132@alunos.uminho.pt

Universidade do Minho

9 de Abril de 2019

1 Introdução

A resolução deste trabalho prático tem como propósito introduzir o **SageMath**, corpos finitos primos, curvas elípticas sobre esses corpos e esquemas criptográficos baseados nos mesmos. Os principais objetivos passam por:

- Implementar um esquema *RSA* como uma classe **Python**.
- Implementar um esquema *ECDSA* como uma classe **Python**.
- Implementar um esquema *ECDH* como uma classe **Python**.

Além disso, devem ser aplicadas as seguintes particularidades aos esquemas desenvolvidos:

- O esquema *RSA* deve fornecer métodos para cifrar, decifrar, assinar e verificar uma mensagem.
- O esquema *ECDSA* deve utilizar uma das curvas primas definidas no **FIPS186-4**.
- O esquema *ECDH* deve utilizar curvas elíticas binárias.

O relatório está dividido em três partes. Cada uma descreve os objetivos a cumprir e está estruturada de forma a que o texto entre os *snippets* de código seja suficientemente explicativo, sobre a implementação e desenho da solução.

2 Imports e funções comuns

Esta secção tem como objetivo importar as bibliotecas que serão necessárias na definição dos esquemas implementados neste trabalho, bem como a implementação de qualquer função que seja comum a várias secções.

```
In [1]: import hashlib
        from sage.crypto.util import ascii_to_bin, bin_to_ascii

        def hash_message(message):
            digest = hashlib.sha256(message).hexdigest()
            return digest
```

```python
def convert_to_ZZ(message):
    raw = ascii_to_bin(message)
    return ZZ(int(str(raw),2))

def bpf(factors):
    f = 0
    for pair in factors:
        p = pair[0]
        if p > f:
            f = p
    return f
```

3 Esquema RSA

O objetivo desta secção passa por definir a classe **Python** que implementa o algoritmo **RSA**. Esta permitirá inicializar uma intância, fornecendo-lhe o parâmetro de segurança. Após criada a instância, a mesma permitirá:

- Cifrar uma mensagem.
- Decifrar uma mensagem previamente cifrada.
- Assinar digitalmente uma mensagem.
- Verificar uma assinatura digital de uma mensagem, previamente produzida.

3.1 Definição do esquema RSA

O esquema **RSA** definido deve sempre receber como parâmetro de inicialização o parâmetro de segurança (tamanho em bits do módulo **RSA**). As 5 funções presentes na definição deste esquema têm a seguinte funcionalidade:

- `__init(self,l)__` tem como objetivo inicializar a instância **RSA** e guardar o módulo **q**, a chave pública e a chave privada, sendo que todas as variáveis utilizadas na geração das anteriores são descartadas após a inicialização.
- `encrypt(self,plaintext)` tem como objetivo cifrar a mensagem **plaintext** recebida.
- `decrypt(self,ciphertext)` tem como objetivo decifrar a mensagem **ciphertext** recebida.
- `sign(self,message)` tem como objetivo assinar digitalmente a mensagem **message** recebida.
- `verify(self,message,signature)` tem como objetivo verificar que a assinatura **signature** está correta tendo em conta a mensagem **message**.

A inicialização da instância **RSA** segue o seguinte algoritmo:

- Gerar p e r tal que $p > 2r \geqslant 2^{l/2}$.
- $q = pr$.
- $\varphi(q) = (p-1)(r-1)$.
- Gerar k (chave pública) tal que $gcd(k, \varphi(q)) = 1$.
- Gerar s (chave privada) tal que $s = 1/k \mod \varphi(q)$.

A cifragem de uma mensagem segue o seguinte algoritmo:

- Conversão da mensagem recebida para um inteiro **ZZ** utilizando a função `convert_to_ZZ(message)` acima definida.
- Calcular o criptograma em forma de inteiro, como $ciphertext = p^k \mod q$, onde p é a mensagem a cifrar em inteiro e k é a chave pública.
- Transformar o criptograma em forma de inteiro numa mensagem em texto e retorná-lo.

A decifragem de um criptograma segue o seguinte algoritmo:

- Conversão do criptograma recebido para um inteiro **ZZ** utilizando a função `convert_to_ZZ(message)` acima definida.
- Calcular o texto limpo em forma de inteiro, como $plaintext = c^s \mod q$, onde c é o criptograma em inteiro e s é a chave privada.
- Transformar o texto limpo em forma de inteiro na mensagem em texto e retorná-la

A assinatura de uma mensagem segue o seguinte algoritmo:

- Calcular um *hash* da mensagem recebida.
- Transformar o *hash* calculado em inteiro.
- Calcular assinatura desse *hash* como $sig = h^s \mod q$, onde h é o *hash* da mensagem e s a chave privada.

A verificação de uma assinatura segue o seguinte algoritmo:

- Calcular mk como $mk = sig^k \mod q$, onde sig é a assinatura e k é a chave pública.
- Transformar mk que está em inteiro, em texto (caractéres *ASCII*).
- Calcular o hash da mensagem.
- Se o hash calculado for igual ao hash extraído da assinatura, então a assinatura é válida, caso contrário é inválida.

In [2]:
```
class RSA:

    def __init__(self,l):
        while True:
            r = random_prime(2**l-1,True,2**(l-1))
            p = random_prime(2**(l+1),True, 2**l)
            if 2*r >= 2**(l/2) and p > 2*r:
                break
        self.q = p * r
        phi = (p - 1) * (r - 1)

        k = ZZ.random_element(phi)
        while gcd(k, phi) != 1:
             k = ZZ.random_element(phi)
        self.public_key = k

        self.private_key = inverse_mod(self.public_key,phi)

    def encrypt(self,plaintext):
        plaintext = convert_to_ZZ(plaintext)
```

```
        ciphertext_zz = power_mod(plaintext,self.public_key,self.q)
        ciphertext_zz_raw = ciphertext_zz.binary()

        bits_missing = 8 - Mod(len(ciphertext_zz_raw),8)
        raw = ('0' * bits_missing) + ciphertext_zz_raw
        ciphertext = bin_to_ascii(raw)

        return ciphertext

    def decrypt(self,ciphertext):
        ciphertext = convert_to_ZZ(ciphertext)
        plaintext = power_mod(ciphertext,self.private_key,self.q)
        raw_b = plaintext.binary()

        bits_missing = 8 - Mod(len(raw_b),8)
        raw = ('0' * bits_missing) + raw_b

        return bin_to_ascii(raw)

    def sign(self,message):
        msg_hash = hash_message(message)
        message = convert_to_ZZ(msg_hash)
        return power_mod(message,self.private_key,self.q)

    def verify(self,message,signature):
        mk = power_mod(signature,self.public_key,self.q)
        mk_raw = mk.binary()

        bits_missing = 8 - Mod(len(mk_raw),8)
        mk_raw = ('0' * bits_missing) + mk_raw

        message = hash_message(message)
        mk_raw = bin_to_ascii(mk_raw)
        return mk_raw == message
```

3.2 Teste ao Esquema RSA

In [3]:
```
# Iniciar uma instância RSA com parâmetro de segurança de 2048 bits.
rsa = RSA(2048)
msg = "Estruturas Criptográficas - Trabalho 2 - Iniciação SageMath - Esquema RSA"
print('message:')
print(msg)

#cifrar a mensagem com a instância previamente criada
ciphertext = rsa.encrypt(msg)

print('\n ciphertext:')
print(ciphertext)
```

```
        print('\n')

        #Decifrar o criptograma previamente produzido
        plaintext = rsa.decrypt(ciphertext)
        print('decrypted:')
        print(plaintext)

        #Assinar e verificar a assinatura
        print('\n')
        print('signature result:')
        sig = rsa.sign(msg)
        if(rsa.verify(msg,sig)):
            print('OK')
        else: print('Not OK!')
```

```
message:
Estruturas Criptográficas - Trabalho 2 - Iniciação SageMath - Esquema RSA

 ciphertext:
[U+034B][U+FFFD][U+FFFD][U+FFFD][U+FFFD]z[U+FFFD]n$fo[U+FFFD]Y[U+FFFD]![[U+FFFD]<R[U+FFFD]M[U+F
[U+FFFD]w[U+FFFD]i[U+FFFD][U+FFFD]uL[U+FFFD]B[U+FFFD]87[U+FFFD]Ge,]?[U+FFFD]²[U+FFFD][U+FFFD][U
JW[U+BF23C]S_ [U+FFFD][U+FFFD]}[U+FFFD]<?_[U+FFFD][U+FFFD][U+FFFD][U+FFFD]_[U+FFFD][U+02EE][U+F
[U+FFFD]Mĭ[U+FFFD]&[U+FFFD]Q¹C[U+FFFD][U+FFFD]'a[U+FFFD]p[U+FFFD]Wl[U+FFFD][U+FFFD]([U+FFFD][U+


decrypted:
Estruturas Criptográficas - Trabalho 2 - Iniciação SageMath - Esquema RSA


signature result:
OK
```

4 Esquema ECDSA

Nesta secção, definiu-se uma classe **Python** que implementa o algoritmo **ECDSA**. A inicialização desta classe terá como objetivo criar os parâmetros, a partir da curva NIST **P256**, gerando também a curva elítica associada, bem como o seu ponto gerador, uma chave pública e uma chave privada. Após a criação da instância, a mesma permitirá:

- Assinar uma mensagem.
- Verificar a assinatura de uma mensagem.

4.1 Definição da curva *NIST*

```
In [4]: NIST = dict()
        NIST['P-256'] = {
            'p': 115792089210356248762697446949407573530086143415290314195533631308867097853951,
```

```
        'n': 115792089210356248762697446949407573529996955224135760342422259061068512044369,
        'seed': 'c49d360886e704936a6678e1139d26b7819f7e90',
        'c': '7efba1662985be9403cb055c75d4f7e0ce8d84a9c5114abcaf3177680104fa0d',
        'b': '5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b',
        'Gx': '6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296',
        'Gy': '4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5'
    }
```

4.2 Definição do esquema ECDSA

As 3 funções presentes neste esquema são as seguintes:

- A função `__init(self)__` tem como objetivo inicializar os parâmetros necessários para que seja, posteriormente, possível assinar e verificar mensagens.
- A função `sign(self,message)` tem como objetivo assinar digitalmente a mensagem **message**.
- A função `verify(self,message,signature)` tem como objetivo verificar a assinatura **signature** tendo em conta a mensagem **message**.

A inicialização de uma instância deste tipo segue o seguinte algoritmo:

- Extrair os parâmetros presentes na curva *NIST* definida no dicionário NIST com a chave `P-256`.
- Criar a curva elítica associada.
- Criar o ponto gerador da curva a partir de `Gx` e `Gy` extraídos.
- Calcular aleatoriamente uma chave privada **s** tal que $1 \leqslant s < n$.
- Calcular a chave pública k como $k = s * G$, onde s é a chave privada.

A assinatura de uma mensagem segue o seguinte algoritmo:

1. Calcular o *hash* da mensagem e converter esse *hash* num inteiro do tipo **ZZ** utilizando a função `convert_to_ZZ(message)`.
2. Calcular um inteiro k aleatório tal que $1 \leqslant k < n$.
3. Calcular um ponto r_point como $r\_point = k * G$, onde G é o ponto gerador da curva.
4. Calcular um inteiro r como $r = r\_point\_x \mod n$, onde r_point_x é a coordenada x do ponto calculado no passo 3.
5. Se $r == 0$ voltar ao passo 2. Caso contrário, prosseguir.
6. Calcular a inversa do aleatório k como $k\_inverse = 1/k \mod n$.
7. Calcular a assinatura como $sig = k\_inverse * (hash + (r * private\_key)) \mod n$.
8. Se $sig == 0$ retornar ao passo 2. Caso contrário, a assinatura da mensagem é o par `(r,sig)`.

A verificação da assinatura de uma mensagem segue o seguinte algoritmo:

1. Se $1 \leqslant r < n$ e $1 < sig < n$, prosseguir. Caso contrário, assinatura inválida.
2. Calcular o *hash* da mensagem e converter esse *hash* em inteiro **ZZ**.
3. Calcular w como $w = 1/sig \mod n$.
4. Calcular u1 como $u1 = hash * w \mod n$.
5. Calcular u2 como $u2 = r * w \mod n$.
6. Calcular cp como $cp = u1 * G + u2 * public\_key$.

7. Se $(cp\_x \mod n) == (r \mod n)$, onde cp_x é a coordenada x do ponto cp, a assinatura é válida. Caso contrário, a assinatura é inválida.

In [5]:
```
class ECDSA:

    def __init__(self):
        curve = NIST['P-256']
        p = curve['p']
        self.n = curve['n']
        b = ZZ(curve['b'],16)
        Gx = ZZ(curve['Gx'],16)
        Gy = ZZ(curve['Gy'],16)
        self.E = EllipticCurve(GF(p),[-3,b])
        self.G = self.E((Gx,Gy))
        self.private_key = ZZ.random_element(1,self.n)
        self.public_key = self.private_key * self.G


    def sign(self,message):
        digest = hash_message(message)
        digest = convert_to_ZZ(digest)
        back_to_1 = False
        while not back_to_1:
            ok = False
            k = ZZ.random_element(1,self.n)
            r_point = k * self.G
            r = Mod(r_point[0],self.n)
            if r == 0:
                back_to_1 = False
            else :
                while not ok:
                    k_inverse = inverse_mod(k,self.n)
                    temp_calc = k_inverse * (digest + (r*self.private_key))
                    s = ZZ(Mod(temp_calc,self.n))
                    if s == 0:
                        ok = True
                    else:
                        ok = True
                        back_to_1 = True

        return r,s

    def verify(self,message,signature):
        sig_r = signature[0]
        sig_s = signature[1]
        if (sig_r < 1 or sig_r > self.n -1 or sig_s < 1 or sig_s > self.n - 1):
            return False
        else:
```

```
digest = hash_message(message)
digest = convert_to_ZZ(digest)
w = inverse_mod(sig_s,self.n)
u1 = ZZ(Mod(digest*w,self.n))
u2 = ZZ(Mod(sig_r*w,self.n))
cp = u1*self.G + u2*self.public_key
if Mod(cp[0],self.n) == Mod(sig_r,self.n):
    return True
else:
    return False
```

4.3 Teste ao esquema ECDSA

```
In [6]: e = ECDSA()
        msg = "Estruturas Criptográficas - Trabalho 2 - Iniciação SageMath - Esquema ECDSA"
        r,s = e.sign(msg)
        if e.verify(msg,(r,s)):
            print 'OK'
        else:
            print 'Not OK'
```

```
OK
```

5 Esquema ECDH

5.1 Definição do esquema ECDH

As 3 funções presentes neste esquema são as seguintes:

- A função `init(self,n)` tem como objetivo definir os parâmetros da curva, a partir da dimensão do corpo **K** recebida como parâmetro.
- A função `generate_key_pair(self)` tem como objetivo gerar um par de chaves (pública e privada) na instância definida.
- A função `exchange(self,sk,peer_public_key)` tem como objetivo calcular a chave comum a dois agentes, recebendo para o efeito a chave privada de um e a chave pública do outro.

A inicialização de uma instância deste tipo segue o seguinte algoritmo:

1. Calcular o corpo base **K** como $K = GF(2^n)$.
2. Gerar b aleatório a partir de **K**.
3. Criar a curva elítica sobre o corpo **K**, definida pelas raízes em K^2 pelo polinómio $\phi = y^2 + xy + x^3 + x^2 + b$.
4. Calcular a ordem da curva elítica criada.
5. Encontrar o maior factor primo **N** da ordem calculada.
6. Se $N < 2^{n-1}$, voltar ao passo 2. Caso contrário, prosseguir.
7. Calcular um ponto aleatório **P** em **E**.
8. Calcular a ordem de **P** (#P):

- Se $\#P < N$ voltar ao passo 7. Caso contrário, prosseguir.
- Se $\#P > N$, calcular $h = \#P/N$ e temos que $G = hP$. Encontrámos assim os parâmetros N, b e G.
- Caso contrário ($\#P == N$), e $G = P$. Encontrámos assim os parâmetros N, b e G.

A geração do par de chaves segue o seguinte algoritmo:

1. É gerada uma chave privada **sk** tal que $1 \leqslant sk < N$.
2. É calculada a chave pública **pk** como $pk = sk * G$.
3. é retornado o par `(sk,pk)`.

A geração da chave partilhada por ambos os agentes segue o seguinte algoritmo:

1. Gerar a chave partilhada **shared_key** como $shared\_key = sk * peer\_public\_key$, onde **sk** será a chave privada de quem invocou a função e **pk** será a chave pública do agente com o qual ele quer combinar uma chave partilhada.
2. Retornar a chave partilhada.

```
In [7]: class ECDH:

            def __init__(self,n):
                K.<t> = GF(2^n)
                ok = False
                while not ok:
                    b = K.random_element()
                    E = EllipticCurve(K,[1,1,0,0,b])
                    e_order = E.order()
                    F = factor(e_order)
                    N = bpf(list(F))
                    if N < 2^(n-1):
                        pass
                    else:
                        while True:
                            P = E.random_point()
                            P_order = P.order()
                            if P_order < N:
                                pass
                            elif P_order > N:
                                h = ZZ(P_order/N)
                                self.G = h*P
                                self.N = N
                                self.b = b
                                ok = True
                                break
                            else:
                                self.N = N
                                self.b = b
                                self.G = P
                                ok = True
```

```
                    break

        def generate_key_pair(self):
            private_key = ZZ.random_element(1,self.N)
            public_key = private_key * self.G
            return (private_key,public_key)

        def exchange(self,sk,peer_public_key):
            shared_key = sk * peer_public_key
            return shared_key
```

5.2 Teste ao esquema ECDH

```
In [8]: ecdh = ECDH(163)
        alice = ecdh.generate_key_pair() #pair (alice_private_key,alice_public_key)
        bob = ecdh.generate_key_pair() #pair (bob_private_key,bob_public_key)
        alice_shared = ecdh.exchange(alice[0],bob[1])
        bob_shared = ecdh.exchange(bob[0],alice[1])

        if alice_shared == bob_shared:
            print 'OK'
        else:
            print 'Not OK'
```

```
OK
```

6 Conclusão

Os resultados da realização deste trabalho prático são, na nossa opinião, bastante satisfatórios, visto que fomos capazes de cumprir todos os objetivos inicialmente propostos, ou seja, à implementação de um esquema **RSA** com funções de cifragem, decifragem, assinatura e verificação, à implementação de um esquema **ECDSA** a partir de uma curva standard da *NIST* com funções de assinatura e verificação e, finalmente à implementação de um esquema **ECDH** sobre curvas elíticas binárias, fornecendo funções que possibilitem gerar um par de chaves com base nos parâmetros criados e a geração da chave partilhada.

Durante a resolução deste trabalho prático, o grupo deparou-se com uma principal dificuldade, que diz respeito ao grupo 3 deste trabalho (esquema **ECDH**) visto que, inicialmente, não conseguimos entender concretamente o que tínhamos que fazer. Essa dificuldade foi ultrapassada após uma sessão de esclarecimento de dúvidas com o professor.

Em suma, é da nossa opinião que realizámos um trabalho muito bom, principalmente, tendo em conta que este foi o primeiro contacto do grupo na utilização do *sagemath* para implementação de criptosistemas.

7 Referências

1. Worksheets TP2 do professor
2. Worksheets TP3 do professor
3. Elliptic curves over Finite fields - SageMath Doc
4. Notas Manuscritas do professor
5. Trabalho ECDSA - Márcio Aurélio Ribeiro Moreira