TP1-Grupo6

April 5, 2021

Ana Margarida Campos (A85166), Nuno Pereira (PG42846)

1 Introdução

O presente projeto enquadra-se na unidade curricular de Estruturas Criptográficas, na qual foi proposta a resolução de dois exercícios distintos.

O primeiro exercício centra-se na criação de uma sessão síncrona segura entre dois agentes (*Emitter* e *Receiver*). Para tal são implementados dois protocolos de acordo de chaves: *Diffie-Helman* com autenticação dos agentes através do esquema de assinaturas DSA (*Digital Signature Algorithm*) e Curvas Elípticas *Diffie-Helman* com autenticação dos agentes através de Curvas Elípticas DSA. Este exercício foi desenvolvido com auxílio do package *Criptography*.

No segundo exercício recorreu-se à utilização do *SageMath* de modo a implementar KEM-RSA, a transformação de *Fujisaki-Okamoto* de modo a que a construir um PKE com segurança IND-CCA, uma implementação do DSA e, por último, a implementação do ECDSA.

Nas secções seguintes é descrito detalhadamente o processo de resolução de cada uma das alíneas de ambos os exercícios.

2 Exercício 1

[1]:
```
import multiprocessing
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hmac
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import ec

listanouce = []
```

```
salt = os.urandom(16)  # Salt partilhado

metadados = os.urandom(16)


def kdf(password, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    key = kdf.derive(password.encode('utf8'))
    # verificação
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    kdf.verify(password.encode('utf8'), key)
    return key
```

2.1 a)

Nesta alínea era pedido um gerador aleatório de *nounces*. Para tal, foi desenvolvida a função ***geraNounce*** que, após receber o tamanho pretendido (em bytes), cria uma sequência de bytes aleatórios. De modo a que não existam *nounces* repetidos, todos os *nounces* gerados são armazenados numa lista. Aquando da geração de um novo *nounce*, é verificado se o mesmo pertence a essa lista, ou seja, se já foi usado, e caso seja verdade ocorre a criação de um novo *nounce*.

[2]:
```
# função que gera nounces únicos
def geraNounce(tamNounce):
    nounce = os.urandom(tamNounce)
    if not (nounce in listanouce):
        listanouce.append(nounce)
        return nounce
    else:
        geraNounce(tamNounce)
```

2.2 b)

De modo a desenvolver uma comunicação segura contra ataques aos vetores de inicialização, recorreu-se à cifra simétrica **AES** (*Advanced Encryption Standard*) no modo **GCM** (*Galois Counter*

Mode) para implementar as funções ***cifragem*** e ***decifragem***. Foi escolhido este modo, uma vez que, se utilizado um *nounce* único por cada cifragem (o que acontece na nossa implementação), ataques ao vetor de inicialização são evitados.

Para que ocorra a autenticação de cada criptograma com **HMAC**, foram implementadas duas funções: ***mac*** e ***mac_verify***. A primeira cria uma tag de autenticação a partir da *password* e da chave derivada. Para tal recorre à função de hash hmac. A segunda é utilizada para verificar a autenticidade. Ambas são usadas para garantir autenticidade na partilha de chaves.

[3]:
```
# função que cifra a mensagem
def cifragem(texto, metadados, key):
    texto = texto.encode('utf8') # conversão do texto limpo para bytes
    # utilização do modo de cifragem GCM:
    aesgcm = AESGCM(key)
    nonce = geraNounce(12)
    texto_cifrado = aesgcm.encrypt(nonce, texto, metadados)
    texto_cifrado += nonce # concatenação do nounce com o texto cifrado para␣
 ↪que depois a decifragem seja possível
    return texto_cifrado


# função que decifra a mensagem
def decifragem(texto_cifrado, metadados, key):
    aesgcm = AESGCM(key)
    nonce = texto_cifrado[-12:]  # atribuir os 12 últimos bytes do texto␣
 ↪cifrado ao nonce
    texto_cifrado = texto_cifrado[:-12] # tirar os 12 últimos bytes
    texto_limpo = aesgcm.decrypt(nonce, texto_cifrado, metadados) # decifragem␣
 ↪com recurso a GCM
    return texto_limpo


def mac(key, chave_derivada):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(chave_derivada)
    tag = h.finalize()
    return tag


def mac_verify(key, chave_derivada, tag):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(chave_derivada)
    h.verify(tag)
```

2.3 c)

Nesta alínea era pedida a implementação do protocolo de acordo de chaves *Diffie-Helman* (DH) com verificação da chave e autenticação dos agentes através do esquema de assinaturas *DSA*. Para tal foram implementadas cinco funções:

- ***geraChavesDH:*** função que gera as chaves DH, pública e privada, de ambos os agentes, a partir dos parâmetros *Diffie-Helman*;
- ***geraChavesDSA:*** função que gera as chaves DSA, pública e privada, de ambos os agentes;
- ***verificacaoAssinatura:*** função que a partir das chaves públicas e da assinatura, verifica se a mesma é válida. Caso não seja retorna erro;
- ***derivacaoChave:*** função responsável pela criação da chave partilhada e respetiva derivação;
- ***DHProtocol_DSA:*** função onde se econtra defenido o protocolo de troca de chaves e de autenticação de assinaturas. Recorre a todas as funções anteriormente descritas e, tem como principal objetivo, a transferência das chaves e da assinatura entre o *Emitter* e o *Receiver*, bem como a respetiva verificação e derivação da chave partilhada entre os agentes. É também nesta função que são chamadas as funções *mac* e *mac_verify* para garantir autenticação.

[4]:
```
# geração dos parâmteros DH
parameters = dh.generate_parameters(generator=2, key_size=2048)

# geração das chaves pública e privada DH
def geraChavesDH():
    # geração da chave privada DH
    private_keyDH = parameters.generate_private_key()
    # geração da chave pública DH e passagem para bytes
    public_keyDH = private_keyDH.public_key().
 ↪public_bytes(encoding=serialization.Encoding.PEM,
                                                        format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo)
    return private_keyDH, public_keyDH

# geração das chaves pública e privada DSA
def geraChavesDSA():
    # geração da chave privada DSA
    private_keyDSA = dsa.generate_private_key(key_size=1024)
    # geração da chave pública DSA e passagem para bytes
    public_KeyDSA = private_keyDSA.public_key().
 ↪public_bytes(encoding=serialization.Encoding.PEM,
                                                                ␣
 ↪format=serialization.PublicFormat.SubjectPublicKeyInfo)
    return private_keyDSA, public_KeyDSA

# verificação da assinatura
def verificacaoAssinatura(assinatura, public_keyDH, public_keyDSA, nome):
    try:
```

```
        public_keyDSA.verify(assinatura, public_keyDH, hashes.SHA256())
        print(nome, "Assinatura validada! \n")
    except Exception as err:
        print("Error: " + str(err))

# criação da chave partilhada e respetiva derivação
def derivacaoChave(private_keyDH, public_keyDH):
    shared_key = private_keyDH.exchange(public_keyDH)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b'handshake data'
    ).derive(shared_key)
    return derived_key

# protocolo de troca de chaves e respetiva autenticação
def DHProtocol_DSA(conn, password, nome):
    # criação das chaves
    private_keyDH, public_keyDH = geraChavesDH()
    private_keyDSA, public_keyDSA = geraChavesDSA()

    # assinatura
    signature = private_keyDSA.sign(public_keyDH, hashes.SHA256())

    # envio da informação (chaves + assinatura) para o outro agente
    info = [public_keyDH, public_keyDSA, signature]
    conn.send(info)

    # receção da informação por parte do agente
    info = conn.recv()
    receiverPK_DH_Bytes = info[0]
    receiverPK_DSA_Bytes = info[1]
    receiverSign = info[2]

    # descerialização das chaves (passagem de bytes para DHPublicKey e␣
↪DSAPublicKey)
    receiverPK_DH = load_pem_public_key(receiverPK_DH_Bytes)
    receiverPK_DSA = load_pem_public_key(receiverPK_DSA_Bytes)

    # verificação da assinatura
    verificacaoAssinatura(receiverSign, receiverPK_DH_Bytes, receiverPK_DSA,␣
↪nome)

    # derivação da chaves
    derived_key = derivacaoChave(private_keyDH, receiverPK_DH)
```

```
    # Autenticação HMAC
    tag = mac(password, derived_key)
    conn.send(tag)

    tagRecebida = conn.recv()
    mac_verify(password, derived_key, tagRecebida)
    print(nome, "Processo concluido\n")

    return derived_key
```

2.4 Comunicação síncrona entre o *Emitter* e o *Receiver*

Para assegurar a comunicação entre os dois agentes é criada uma ligação entre ambos através de um pipe. Desta forma, torna-se possível o acordo de chaves entre os agente e posterior troca de mensagens entre os mesmos.

Modo de comunicação: Para inciar a comunicação é necessário receber a *password* tanto do *Emitter* como do *Receiver*. Ambas são derivadas a partir da função de derivação de *passwords* **KDF**. Posteriormente é incializado o protocolo de acordo de chaves e respetiva autenticação dos agentes através das assinaturas. Se tudo correr conforme esperado, isto é, as *passwords* serem as mesmas e não ocorrer erros na verificação e autenticação, a chave é partilhada entre ambos os agentes e é estabelecida a comunicação. Após o estabelecimento da comunicação, ocorre a troca de mensagens entre ambos. Para tal as mensagens são cifradas no lado do *Emitter* e decifradas no lado do *Receiver*.

[5]:
```
# comunicação
# troca de chaves e cifragem das mensagens a serem enviadas
def emitter(conn, msgs, password):
    shared_key = DHProtocol_DSA(conn, password, "[Emitter]")
    for msg in msgs:
        texto_cifrado = cifragem(msg, metadados, shared_key)
        print("[Emitter] Mensagem enviada!\n")
        conn.send(texto_cifrado)

    conn.close()

# troca de chaves e decifragem das mensagens
def receiver(conn,password):
    shared_key = DHProtocol_DSA(conn, password, "[Receiver]")
    try:
        texto_cifrado = conn.recv()
        texto_limpo = decifragem(texto_cifrado, metadados, shared_key)
        print("[Receiver] A mensagem recebida foi: " + texto_limpo.
 ↪decode('utf8') + "\n")
    except Exception as err:
        print("Error: " + str(err))
        return 1
```

[6]:
```
if __name__ == '__main__':
    passEm = input("[Emitter] Introduza a password: ")
    chave = kdf(passEm, salt)

    passRc = input("[Receiver] Introduza a password: ")
    chave2 = kdf(passRc, salt)
    m = ""
    while m != "sair":
        parent_conn, child_conn = multiprocessing.Pipe()
        msgs = []
        m = input("Mensagem:")

        # escrever "sair" para terminar a comunicação
        if m != 'sair':
            msgs.append(m)
            p1 = multiprocessing.Process(target=emitter, args=(parent_conn,␣
 ↪msgs,chave))
            p2 = multiprocessing.Process(target=receiver,␣
 ↪args=(child_conn,chave2))


            # running processes
            p1.start()
            p2.start()

            # wait until processes finish
            p1.join()
            p2.join()
        else:
            p1.terminate()
            p2.terminate()
```

```
[Receiver][Emitter] Assinatura validada!
 Assinatura validada!


[Emitter][Receiver]  Processo concluido
Processo concluido

[Emitter] Mensagem enviada!


[Receiver] A mensagem recebida foi: mensagem ultra secreta
```

2.5 d)

Nesta alínea era pedida a re-implementação do esquema anterior, mas utilizando protocolos baseados em Curvas Elípticas, mais especificamente **ECDH** (*Elliptic-Curve Diffie-Helman*) e **ECDSA** (*Elliptic-Cuve Digital Signature Algorithm*).

A implementação destes protocolos foi bastante semelhante ao esquema anteriormente apresentado mas com algumas diferenças:

- neste caso, apenas existe uma função para gerar as chaves tanto para o DH como para o DSA uma vez que as mesmas são geradas de igual forma. É notável que a *performance* desta implementação foi superior à anterior dado que, não existe a geração de parâmetros que demoravam um tempo considerável a ser criados;
- as restantes diferenças apenas se centram na modificação dos protocolos anteriores para ECDH e ECDSA.

[7]:
```
# gera chaves tanto de ECDH como de ECDSA - são de feitas de igual modo
def geraChaves():
    # geração da chave privada DH
    private_key = ec.generate_private_key(ec.SECP384R1())
    # geração da chave pública DH e passagem para bytes
    public_key = private_key.public_key().public_bytes(encoding=serialization.
 ↪Encoding.PEM,
                                                       format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo)
    return private_key, public_key


def verificacaoAssinaturaEC(assinatura, public_keyDH, public_keyDSA, nome):
    try:
        public_keyDSA.verify(assinatura, public_keyDH, ec.ECDSA(hashes.
 ↪SHA256()))
        print(nome, "Assinatura validada! \n")
    except Exception as err:
        print("Error: " + str(err))


def derivacaoChaveEC(private_keyDH, public_keyDH):
    shared_key = private_keyDH.exchange(ec.ECDH(), public_keyDH)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b'handshake data'
    ).derive(shared_key)
    return derived_key
```

```
def ECDHProtocol_ECDSA(conn, password, nome):
    # criação das chaves
    private_keyECDH, public_keyECDH = geraChaves()
    private_keyECDSA, public_keyECDSA = geraChaves()

    # assinatura
    signature = private_keyECDSA.sign(public_keyECDH, ec.ECDSA(hashes.SHA256()))

    info = [public_keyECDH, public_keyECDSA, signature]
    conn.send(info)

    info = conn.recv()
    receiverPK_ECDH_Bytes = info[0]
    receiverPK_ECDSA_Bytes = info[1]
    receiverSign = info[2]

    receiverPK_DH = load_pem_public_key(receiverPK_ECDH_Bytes)
    receiverPK_DSA = load_pem_public_key(receiverPK_ECDSA_Bytes)


    verificacaoAssinaturaEC(receiverSign, receiverPK_ECDH_Bytes,␣
↪receiverPK_DSA, nome)

    derived_key = derivacaoChaveEC(private_keyECDH, receiverPK_DH)

    tag = mac(password, derived_key)
    conn.send(tag)

    tagRecebida = conn.recv()
    mac_verify(password, derived_key, tagRecebida)
    print(nome, "Processo concluido\n")

    return derived_key
```

2.6 Comunicação síncrona entre o *Emitter* e o *Receiver*

A comunicação é feita do mesmo modo que nos protocolos anteriores, apenas com a diferença da troca de chaves entre ambos os agentes ser realizada com os protocolos ECDH e ECDSA.

[8]:
```
# comunicação
def emitter(conn, msgs, password):
    shared_key = ECDHProtocol_ECDSA(conn, password, "[Emitter]")
    for msg in msgs:
        texto_cifrado = cifragem(msg, metadados, shared_key)
        conn.send(texto_cifrado)
        print("[Emitter] Mensagem enviada!\n ")
```

```
    conn.close()


def receiver(conn,password):
    shared_key = ECDHProtocol_ECDSA(conn, password, "[Receiver]")
    try:
        texto_cifrado = conn.recv()
        texto_limpo = decifragem(texto_cifrado, metadados, shared_key)
        print("[Receiver] A mensagem recebida foi: " + texto_limpo.
 ↪decode('utf8'))
    except Exception as err:
        print("Error: " + str(err))
        return 1


if __name__ == '__main__':
    passEm = input("[Emitter] Introduza a password: ")
    chave = kdf(passEm, salt)

    passRc = input("[Receiver] Introduza a password: ")
    chave2 = kdf(passRc, salt)
    m = ""
    while m != "sair":
        parent_conn, child_conn = multiprocessing.Pipe()
        msgs = []
        m = input("Mensagem:")

        # escrever "sair" para terminar a comunicação
        if m != 'sair':
            msgs.append(m)
            p1 = multiprocessing.Process(target=emitter, args=(parent_conn,␣
 ↪msgs,chave))
            p2 = multiprocessing.Process(target=receiver,␣
 ↪args=(child_conn,chave2))


            # running processes
            p1.start()
            p2.start()

            # wait until processes finish
            p1.join()
            p2.join()
        else:
            p1.terminate()
            p2.terminate()
```

```
[Receiver] Assinatura validada!

[Emitter] Assinatura validada!

[Receiver] [Emitter]Processo concluido
 Processo concluido


[Emitter] Mensagem enviada!

[Receiver] A mensagem recebida foi: mensagem ultra secreta
```

3 Exercício 2

3.1 a)

Nesta alínea era pedida a construção de uma classe Python que implemente um **KEM-RSA**. Numa fase inicial, é necessária a geração das chaves pública e privada a partir de um parâmetro de segurança. Para tal foi desenvolvida a função ***gera_chaves*** que com base no algritmo RSA, gera dois números primos aleatórios, q e p. O n, módulo para as chaves pública e privada, é gerado a partir da multiplicação desses números primos. Posteriormente é criado o *phi* e encontrado um número aleatório e de modo a que, este e o *phi(n)* sejam relativamente primos entre si. Por fim, são retornados os tuplos *(d,p,q)* e *(e,n)* que correspondem à chave privada e pública respetivamente.

Como funções auxiliares à implementação do **KEM-RSA** foram também implementadas as funções de cifragem (***cifrarRSA***) e decifram (***decifrarRSA***) baseadas no algoritmo RSA.

De forma a encapsular os dados corretamente, foi necessária a combinação de dois mecanismos: o **DEM** (*data encapsulation mechanism*) que tem como objetivo ofuscar os dados e o **KEM** (*key encapsulation mechanism*) que tem como função gerar, comunicar e ofuscar a chave privada requerida pelo **DEM**. Com este fim, foram criadas as seguintes funções: - ***KEM***: semelhante a um gerador aleatório que produz um par de resultados: a chave usada pelo **DEM** e o encapsulamento dessa chave. De forma a gerar o primeiro resultado, foi cifrado com resurso à função ***cifrarRSA*** um número pseudo-aleatório r. Posteriormente, para chegar ao segundo resultado, é feito o encapsulamento do r a partir de uma função de *hash*;

- ***KREV***: permite a revelação da chave anteriormente ofuscada. Para tal utiliza como auxílio a função ***decifrarRSA***;
- ***DEM***: permite o encapsulamento da mensagem a partir da operação *XOR* entre a chave e a mensagem;
- ***DREV***: possibilita a decifragem de modo a obter a mensagem original. Desta forma, recorre à função ***KREV*** para obter a chave e, posteriormente, realiza a operação *XOR* entre a criptograma e a chave obtida.

[2]:
```
import hashlib
import binascii
from binascii import unhexlify, hexlify
```

```
class KEM_RSA:

    def __init__(self,s):
        self.s=s

    #gera chaves pública e privada
    def gera_chaves(self):
        p = random_prime(2^self.s-1,True,2^(self.s-1))
        q = random_prime(2^self.s-1,True,2^(self.s-1))
        n = p*q
        ##  Função totiente de Phi
        phi = (p-1)*(q-1)
        #ecolha um inteiro que seja relativamente primo com o phi de n

        e = ZZ.random_element(phi)
        # descoberta de um número "e" que tenha de ser primo entre si
        while gcd(phi, e) != 1:
            #public key, de cifragem
            e = ZZ.random_element(phi)
        #private key, chave de decifragem
        d = inverse_mod(e,phi)
        return (d,p,q), (e,n)

    # cifragem com base no algortimo RSA
    def cifrarRSA(self,mensagem, e, n):
        cifra= pow(mensagem,e,n)
        return cifra

    # decifragem com base no algortimo RSA
    def decifrarRSA(self,cifrado_rsa, d, n):
        decifrar= pow(cifrado_rsa, d, n)
        return decifrar

    # operação XOR
    def bxor(self, a, b):
        return bytes([ x^^y for (x,y) in zip(a, b)])


    # geração da chave e encapusulamento
    def KEM(self,chave_publica):
        e, n = chave_publica
        r = ZZ.random_element(0,n - 1)
        cifra = kem_rsa.cifrarRSA(r,e,n)
        key = hash(r)
        return (cifra,key)
```

```
    # Algoritmo de revelação da chave
    def KRev(self, cifra, chave_privada, chave_publica):
        d,p,q = chave_privada
        e, n = chave_publica
        r = kem_rsa.decifrarRSA(cifra,d,n)
        key = hash(r)
        return key


    # data encapsulation mechanism: encapsulamento da mensagem a partor do XOR
    def DEM(self, mensagem, key):
        m = binascii.hexlify(mensagem.encode('utf-8'))
        k = binascii.hexlify(str(key).encode('utf-8'))
        criptograma = kem_rsa.bxor(m,k)
        return criptograma

    # Revelação da mensagem original
    def DRev(self, criptograma, cifra, chave_privada, chave_publica):
        key = self.KRev(cifra, chave_privada, chave_publica)
        k = binascii.hexlify(str(key).encode('utf-8'))
        texto_limpo = kem_rsa.bxor(criptograma,k)
        texto_limpo = binascii.unhexlify(texto_limpo.decode('utf-8'))
        texto_limpo = texto_limpo.decode('utf-8')
        return texto_limpo
```

[2]:
```
kem_rsa = KEM_RSA(512)
chave_privada, chave_publica = kem_rsa.gera_chaves()

cifra, keyEnc = kem_rsa.KEM(chave_publica)
criptograma = kem_rsa.DEM("mensagem secreta", keyEnc)
print("Criptograma: ",criptograma)


texto_limpo = kem_rsa.DRev(criptograma,cifra, chave_privada, chave_publica)
print("Texto limpo: ", texto_limpo)
```

```
Criptograma:  b'\x05P\x05\x06\x05W\x04\x01\x05\x04\x05\x0f\x05\x01\x05\\\x01\t\x
04\x06\x05\x02\x05\x06\x04\x03\x05\x05\x04\x07\x05\t'
Texto limpo:  mensagem secreta
```

3.2 b)

Nesta alínea era pedida a construção de um **PKE** que seja **IND-CCA** seguro, a partir da transformação *Fujisaki-Okamoto*. Com vista a este fim, foram desenvolvidas duas funções:

- ***tof***: tem como objetivo cifrar a mensagem e segue os seguintes passos:

  1 - criar r que é o resultado de uma função de *hash* aplicada a um número pseudo-aleatório;

2 - calcular g a partir de *hash(r)*;
3 - gerar y, que corresponde à ofuscação da mensagem, a partir da operação *XOR* entre a mensagem de texto-limpo e o g anterior;
4 - concatenar y com r (y||r) e cifrar esta concatenação a partir do algoritmo RSA, obtendo assim o encapsulamento da chave;
5 - calcular a *key* a partir de uma função de hash aplicada à concatenação anterior;
6 - por último, ocorre a ofusção da chave que é o resultado da operação *XOR* entre a *key* e o r.

- ***tof_decifra***: tem como objetivo decifrar a mensagem, obetendo o texto limpo. Segue os passos:

  1 - revelação da chave (*key*) a partir da decifragem RSA e cálculo do hash corresponde;
  2 - obtenção do r através do XOR do c com a *key* e cálculo do g;
  2 - fazer novamente a concatenação do y com o r obtido e cifrar com RSA. Caso o resultado da cifragem dê o mesmo valor que o anterior, passo 3 senão é retornado erro;
  3 - operação *XOR* entre o y e o g, para obter como resultado o texto limpo (mensagem original).

[37]:
```
# Transformação de Fujisaki-Okamoto
kem_rsa = KEM_RSA(512)
chave_privada, chave_publica = kem_rsa.gera_chaves()

def tof(chave_publica, mensagem):
    e, n = chave_publica
    # número aleatório r
    r = hash(ZZ.random_element(0,n - 1))
    # calcular hash(r)
    g = hash(str(r))
    # XOR da mensagem com o hash(r)
    m = binascii.hexlify(mensagem.encode('utf-8'))
    g = binascii.hexlify(str(g).encode('utf-8'))
    y = kem_rsa.bxor(m,g) # ofuscação da mensagem
    # concatenação do y com o r
    y2 = int.from_bytes(y, "big")
    concatencao_y_r = str(y2) + str(r)
    # cifragem da concatenação com RSA
    cifra = kem_rsa.cifrarRSA(int(concatencao_y_r),e,n) # encapsulamento da␣
 ↪chave
    # calcular o hash da concatenação (corresponde à key)
    key = hash(concatencao_y_r)
    k = binascii.hexlify(str(key).encode('utf-8'))
    # XOR da key com o r
    r_bytes = binascii.hexlify(str(r).encode('utf-8'))
    c = kem_rsa.bxor(r_bytes,k) # ofuscação da chave
    return y, cifra, c

def tof_decifra(chave_publica,chave_privada,y, cifra, c):
```

```
    e, n = chave_publica
    d,p,q = chave_privada
    # revelação da chave (KREV)
    decifra = kem_rsa.decifrarRSA(cifra,d,n)
    key = hash(str(decifra))
    k = binascii.hexlify(str(key).encode('utf-8'))
    # XOR entre c e k para descobrir o r
    r = kem_rsa.bxor(c,k)
    r = binascii.unhexlify(r.decode('utf-8'))
    g = hash(r)
    g = binascii.hexlify(str(g).encode('utf-8'))
    # verificação
    y2 = int.from_bytes(y, "big")
    concatencao_y_r = str(y2) + str(int(r))
    cifra2 = kem_rsa.cifrarRSA(decifra,e,n)
    if cifra!=cifra2:
        print("ERRO")
        return
    else:
        # XOR do y com g(r) -> hash(r)
        texto_limpo = kem_rsa.bxor(y,g)
        texto_limpo = binascii.unhexlify(texto_limpo.decode('utf-8'))
        texto_limpo = texto_limpo.decode('utf-8')
        return texto_limpo




mensagem = "mensagem secreta"
y, cifra, c = tof(chave_publica, mensagem)
print("Mensagem:", mensagem, "\n" )
print("Ofuscação da mensagem", y, "\n")
print("Encapsulamento da chave", cifra, "\n")
print("Ofuscação da chave", c, "\n")
texto_limpo = tof_decifra(chave_publica, chave_privada, y, cifra, c)
print("Texto limpo:",texto_limpo)
```

```
Mensagem: mensagem secreta

Ofuscação da mensagem b'\x05W\x05\x04\x05T\x04\x0b\x05\x00\x05\x05\x05\x04\x05S\
x01\x00\x04\x05\x05\x05\x05\x03\x04\x05\x05\x03\x04\x0c\x05\x06'

Encapsulamento da chave 341310794132168390291249753770500539053451431008256527073555204282324326789492405540056656312516912655140673173313870907447714448077719825570712288655965156568805029005336827746286814097433571558123068512806320676729617032260643025974036282528375869298680668536796380306308995120709071084038200803493279159 57
```

```
Ofuscação da chave b'\x01W\x00\x03\x00\x0f\x00\x08\x00\x03\x00\x07\x00\x03\x00\x
00\x00\x07\x00\x02\x00\x06\x00\x0f\x00\x03\x00\x06\x00\x01\x00\x05\x00\x03\x00\r
'

Texto limpo: mensagem secreta
```

3.3 c)

Existem 4 passos importantes na implementação do algoritmo *DSA*. São eles: a geração dos números primos necessários, a geração das chaves, a assinatura e a verificação. Cada um destes passos deu origem às seguintes funções:

- ***numerosPrimos***: recebe como parâmetros o tamanho dos primos q e p e, com estes valores, são gerados estes números primos através de uma função pseudo-aleatória. Existe a particularidade do número *p-1* ter de ser múltiplo do número q. Posteriormente é gerado o inteiro h e calculado o valor de g que é calculado a partir de $g := h(p-1)/q \ mod \ p$;
- ***gera_chaves***: gera a chaves pública e privada a partir dos parâmetros retornados pela função ***numerosPrimos***. A chave privada é um inteiro pseudo-aleatório entre 1 e q-1 e a chave pública é gerada da forma: $y := gx \ mod \ p$;
- ***assinatura***: é realizada a assinatura digital da mensagem passada como argumento. A assinatura é um par de valores r e s que são calculados a partir da chave privada, dos números primos e de um inteiro aleatório k que vai de 1 até q-1: $r := gk \ mod \ p$ e $s := (k\text{-}1(H(m) + xr)) \ mod \ q$;
- ***verificação***: ocorre a verificação da assinatura na mensagem. O primeiro passo desta fase centrou-se na verificação da seguinte condição: $0<r<q$ e $0<s<q$, seguido do cálculo das seguintes variáveis:
  - $w := s\text{-}1 \ mod \ q$
  - $u1 := H(m) \times w \ mod \ q$
  - $u2 := r \times w \ mod \ q$
  - $v := (g\hat{}u1 \ g\hat{}u2 \ mod \ p) \ mod \ q$ Caso v seja igual à assinatura passada como argumento, então a assinatura é válida. Senão a assinatura não é validada e é imprimida uma mensagem de erro no ecrã.

[9]:
```
class DSA:
    # tamanhos recomendados:  (1024, 160), (2048, 224), (2048, 256), ou (3072,␣
 ↪256)
    def __init__(self, tamanhoP, tamanhoQ):
        self.tamanhoP = tamanhoP
        self.tamanhoQ = tamanhoQ

    # geração dos números primos necessários
    def numerosPrimos(self):
        q = random_prime(2^self.tamanhoQ-1,True,2^(self.tamanhoQ-1))
        p = random_prime(2^self.tamanhoP-1,True,2^(self.tamanhoP-1))
```

```
        # (p-1) tem de ser múltiplo de q
        while (p-1)%q !=0:
            p = random_prime(2^self.tamanhoP-1,True,2^(self.tamanhoP-1))
        h = ZZ.random_element(2,p - 2)
        g = pow(h, (p-1)//q, p)
        while g == 1:
            h = ZZ.random_element(2,p - 2)
            g = pow(h, (p-1)/q, p)
        return (p, q, g)

    # geração das chaves pública e privada
    def gera_chaves(self, p, q, g):
        # chave privada
        x = ZZ.random_element(1,q - 1)
        # chave pública
        y = pow(g, x, p)
        return x,y

    # assinatura digital da mensagem
    def assinatura(self, p, q, g, mensagem, x):
        k = ZZ.random_element(1,q - 1)
        r = mod(pow(g, k, p), q)
        # caso r = 0 começa de novo com um k diferente
        while r==0:
            k = ZZ.random_element(1,q - 1)
            r = pow(g, k, p)
        # Fermat's little theorem (+ rapido usando este teorema)
        k_inv = pow(k, q-2, q)
        h_x_r = mod(hash(mensagem) + (x * r), q)
        # propriedade: (A * B) mod C = (A mod C * B mod C) mod C
        s = mod(mod(k_inv,q) * mod(h_x_r,q), q)
        # caso s = 0 começa de novo com um k diferente
        while s==0:
            k = ZZ.random_element(1,q - 1)
            s = ((k^(-1))*(hash(mensagem) + (x * r))) % q
        return (r,s)

    # verificação da assinatura na mensagem
    def verificacao(self, assinatura, mensagem, chave_publica, q, p, g):
        r, s = assinatura
        if int(r) > 0 and int(r) < int(q) and int(s) > 0 and int(s) < int(q):
            # Fermat's little theorem
            w = pow(s, q-2, q)
            u1 = mod((mod(hash(mensagem),q) * mod(w,q)), q)
            # propriedade: (A * B) mod C = (A mod C * B mod C) mod C
            u2 = mod((mod(r,q) * mod(w,q)),q)
            v = mod(mod((pow(g, u1, p) * pow(chave_publica, u2, p)), p),q)
```

```
            if v == r:
                print("Assinatura validada!")
            else:
                print("ERRO! Assinatura não validada!")
        else:
            print("ERRO! Assinatura não validada!")
```

[10]:
```
dsa = DSA(24, 16)
mensagem = "mensagem ultra secreta"

p, q, g = dsa.numerosPrimos()
x, y = dsa.gera_chaves(p, q, g)
print("chave privada:", x,"\n")
print("chave pública:", y, "\n")
assinatura = dsa.assinatura(p,q,g,mensagem, x)
print("assinatura:", assinatura,"\n")
dsa.verificacao(assinatura, mensagem, y, q, p, g)
```

```
chave privada: 23687

chave pública: 7697409

assinatura: (16069, 36683)

Assinatura validada!
```

3.4 d)

Nesta alínea era pedido a construção de uma classe Python que implemente o *ECDSA* usando uma das curvas elípticas primas definidas no **FIPS186-4**. Com vista a este fim, foi, incialmente, construída uma classe auxiliar, *Curva*, que é responsável por armazenar todos os valores da curva escolhida, neste caso, da curva *P-192*. Nesta classe existe uma função (***G***) que é responsável por desenvolver a equação da curva e retorna G, o ponto gerador da curva elíptica definido a partir de (Gx,Gy). Na classe *ECDSA*, são seguidos os 3 passos necessários para a implementação deste algoritmo: geração de chaves, assinatura e verificação. Deste modo foram desenvolvidas 3 funções:

- ***gera_chaves***: gera as chaves pública e privada com base na curva passada como argumento. A chave privada corresponde a um inteiro pseudo-aleatório no intevalo [1,n-1]. A chaves pública é um ponto da curva que é calculado a partir de $Q = d$ G*.
- ***assinatura***: é realizada a assinatura digital da mensagem passada como argumento. A assinatura neste caso também retorna um par s e r que são calculados da seguinte forma: 1- Calcular e = hash(mensagem);
  2- Calcular z como sendo os *bits* mais à esquerda de e. Para tal foi feito o cálculo $z := e \ mod \ p$;
  3- Gerar um número pseudo-aleatório k entre [1, n-1];
  4- Calcular o ponto da curva $(x1,y1) = K \times G$;

5- Calcular $r = x1 \ mod \ n$, caso seja igual a 0 volta ao passo 3;
6- Calcular $s = K^{1}(z + rd) \ mod \ n$, caso seja igual a 0 volta ao passo 3;

- ***verificacao***:ocorre a verificação da assinatura na mensagem. O primeiro passo desta fase centrou-se na verificação da seguinte condição: $0<r<q$ e $0<s<q$, seguido do cálculo das seguintes variáveis:

  -e = hash(mensagem)
  -$z:= e \ mod \ p$;
  -$w:= s\text{-}1 \ mod \ q$
  -$u1:= zw \ mod \ q$
  -$u2:= rw \ mod \ q$
  -calcular o ponto da curva $(x1,y1,o) = u1 \times G + u2 \times Q$. Caso $o = 0$ a assinatura é inválida. Caso r seja igual a $x1 \ mod \ n$, então a assinatura é válida. Senão a assinatura não é validada e é imprimida uma mensagem de erro no ecrã.

[11]:
```
class Curva:
     def __init__(self, p, n, seed, c, b, Gx, Gy):
            self.p = p
            self.n = n
            self.seed = seed
            self.c = c
            self.b = b
            self.Gx = Gx
            self.Gy = Gy
            self.G = self.G()

     # retorna G, o ponto gerador da curva
     def G(self):
            b = ZZ(self.b, 16)
            Gx = ZZ(self.Gx, 16)
            Gy = ZZ(self.Gy, 16)
            # E : y^2  x^3 - 3x +b (mod p)
            E = EllipticCurve(GF(self.p), [-3,b])
            G = E(Gx, Gy)
            return G


class ECDSA:
    def __init__(self, curva):
        self.curva = curva

    # gera as chaves pública e privada
    def gera_chaves(self):
        # chave privada
        d = ZZ.random_element(1,curva.n - 1)
        # chave pública
        Q = d * curva.G
```

```
        return d, Q

    # assinatura digital da mensagem com curvas elípticas
    def assinatura(self, mensagem, d):
        e = hash(mensagem)
        z = mod(e,curva.p)
        k = ZZ.random_element(1,curva.n - 1)
        x1, y1, _= k * curva.G
        r = mod(x1, curva.n)
        # se r=0 repetir
        while r==0:
            k = ZZ.random_element(1,curva.n - 1)
            x1, y1, _= k * curva.G
            r = mod(x1, curva.n)
        # propriedade: (A * B) mod C = (A mod C * B mod C) mod C
        s = mod((pow(k, curva.n-2, curva.n)) * mod(int(z) + (int(r)*d), curva.
↪n) ,curva.n)
        while s==0:
            k = ZZ.random_element(1,curva.n - 1)
            x1, y1, _= k * curva.G
            r = mod(x1, curva.n)
            s = mod((pow(k, curva.n-2, curva.n)) * mod(int(z) + (int(r)*d),␣
↪curva.n) ,curva.n)
        return r, s

    # verificação da assinatura na mensagem
    def verificacao(self, assinatura, mensagem, Q):
        r, s = assinatura
        if int(r) > 0 and int(r) < int(curva.n) and int(s) > 0 and int(s) <␣
↪int(curva.n):
            e = hash(mensagem)
            z = mod(e,curva.p)
            w = mod(s^(-1), curva.n)
            u1 = mod(mod(z, curva.n) * mod(w, curva.n), curva.n)
            u2 = mod(mod(r, curva.n) * mod(w, curva.n), curva.n)
            x1, y1, o=  int(u1) * curva.G + int(u2) * Q
            if o == 0:
                print("ERRO! Assinatura não validada!")
            if r == mod(x1,n):
                print("Assinatura validada!")
            else:
                print("ERRO! Assinatura não validada!")
        else:
            print("ERRO! Assinatura não validada!")
```

[12]:
```
# curva P-192
p = 6277101735386680763835789423207666416083908700390324961279
n = 6277101735386680763835789423176059013767194773182842284081
seed = '3045ae6fc8422f64ed579528d38120eae12196d5'
c = '3099d2bbbfcb2538542dcd5fb078b6ef5f3d6fe2c745de65'
b = '64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1'
Gx = '188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012'
Gy = '07192b95ffc8da78631011ed6b24cdd573f977a11e794811'
curva = Curva(p, n, seed, c, b, Gx, Gy)

mensagem = "mensagem ultra secreta"
print("Mensagem:", mensagem,"\n")
ecdsa = ECDSA(curva)
d, Q = ecdsa.gera_chaves()
print("chave pública:",Q,"\n")
print("chave privada:",d,"\n")
assinatura = ecdsa.assinatura(mensagem, d)
print("assinatura:", assinatura,"\n")
ecdsa.verificacao(assinatura, mensagem, Q)
```

```
Mensagem: mensagem ultra secreta

chave pública: (232695490375533362893156539610076204544959197162522015323 4 :
5043200025129654750258998912696412377063069310658871679719 : 1)

chave privada: 365664799967777562816690782055678464664184545052068812548 2

assinatura: (849923441469980626054524602952621544066882734943091444019,
412371211292815695018869556322889562471582113266924132235 7)

Assinatura validada!
```