TP1_Ex3

March 7, 2023

1 Exercício 3

1.1 Descrição do Problema

O objetivo deste exercício é desenvolver uma AEAD com "Tweakable Block Ciphers". Uma cifra AEAD consiste numa cifra por blocos na qual é feita uma autenticação de entidade e texto cifrado. Neste problema, pretendemos implementar um algoritmo que recorra a "Tweakable Block Ciphers". Estes seguirão um comportamento semelhante aos algoritmos de Tweakable Block Ciphers apresentados nas aulas teóricas desta UC. Esta cifra deve recorrer a uma cifra primitiva como AES-256 ou o ChaCha20.

Esta cifra será usada num canal privado de informação assíncrona com acordo de chaves, utilizando o "X448 key exchange", disponivel na biblioteca Cryptography. Para a autenticação dos agentes será utilizado o "Ed448 Signing&Verification". Adicionalmente, este algoritmo deverá também apresentar uma fase de confirmação da chave acordada.

[1]:
```
#Imports necessários
import os
import asyncio
import random
from pickle import dumps, loads
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import x448, ed448
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.exceptions import InvalidSignature

# Para a porção de código associada com as queues (na comunicação entre␣
 ↪participantes)
import nest_asyncio

nest_asyncio.apply()
```

1.2 Criação das chaves privadas/públicas

O acordo de chaves, como indicado no enunciado, deve ser feito com **"X448 key exchange"**.

Para tal é necessário que cada um dos agentes crie as suas próprias chaves (privada-pública), para que posteriormente as chaves públicas possam ser partilhadas e permitir a criação de uma chave partilhada.

Neste caso será usando a **curva448**, como apresentado na API: - https://cryptography.io/en/latest/hazmat/primitives/asymmetric/x448/

A chave resultante do acordo de chaves será usada no decorrer da comunicação assíncrona entre agentes para **cifrar/decifrar** os dados.

[2]:
```
def generateKeys():
    # Generate private key for exchange
    prv_key = x448.X448PrivateKey.generate()

    # Generate public key thorugh private key
    pub_key = prv_key.public_key()

    return prv_key, pub_key
```

1.3 Criação das chaves partilhadas

Após ter as chaves de cada agente (privada/pública), podemos estabelecer o *exchange*, i.e o acordo entre ambos agentes, sobre um segredo partilhado.

A *shared_ key* criada, tal como recomendado na API, deve ser passada por uma função de derivação, no âmbito de a tornar mais segura; adicionando mais informações à chave para destruir qualquer estrutura que possa ser criada.

Será criada uma *shared_ key*, a ser usada no âmbito da cifra.

[3]:
```
def generateShared(prv_key, peer_key):

    peer_cipher_key = x448.X448PublicKey.from_public_bytes(peer_key)

    # Gerar uma chave partilha para cifra
    cipher_key = prv_key.exchange(peer_cipher_key)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=16, #32,
        salt=None,
        info=b'handshake data',
    ).derive(cipher_key)

    return derived_key
```

1.4 Assinar a mensagem

A autenticação dos agentes é realizada com **"Ed448 Signing&Verification"**; sendo este um algoritmo de assinaturas com recurso ao **EdDSA** (*Edwards-curve Digital Signature Algorithm*).

Para tal é necessário um novo par de chaves (privada/pública), do tipo *Ed448*, para um dado participante que pretende assinar uma mensagem. As chaves públicas devem ser partilhadas por ambos, de modo a validar a **assinatura** da mensagem que receberam. Sendo a assinatura realizada através da chave privada do agente, a enviar a mensagem.

Criou-se um método particular para gerar as chaves de autenticação dos agentes, para facilitar e tornar o código mais *readable*.

[4]:
```
def generateSignKeys():

    ## Chave privada para assinar
    private_key = Ed448PrivateKey.generate()

    ## Chave pública para autenticar
    public_key = private_key.public_key()

    return private_key, public_key

def signMsg(prv_key, msg):

    signature = prv_key.sign(msg)

    return signature
```

1.5 Funções auxiliares para cifragem

De seguida iremos apresentar três métodos auxiliares criados, cujos objetivos são adicionar padding ao *plaintext*, executar a cifra TPBC num único bloco e aplicar a cifra TPBC a todos os blocos do plaintext.

1.5.1 Pad

Numa cifra de blocos, o plaintext deverá ser dividido em blocos com o mesmo tamanho. Contudo, poderá ser necessário adicionar alguns bytes ao último bloco, caso este não tenha o tamanho fixo. A função que permite adicionar **padding** ao *plaintext*, sendo dado o tamanho de cada bloco, irá obter o último bloco do *plaintext*, e preenche *bytes* com **0** (neste caso
x00), até que este último tenha o tamanho definido. Esta retorna o *plaintext* atualizado e o tamanho do último bloco antes do padding.

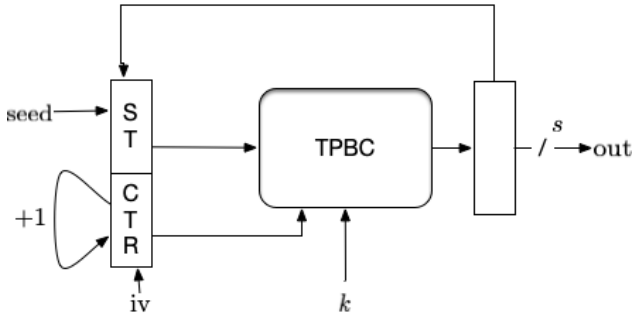
1.5.2 Cifra *TPBC* (*Tweakable Primitive Block Cipher*)

Nesta secção encontra-se também definida a função que permite **executar a cifra *TPBC* (*Tweakable Primitive Block Cipher*)** num bloco. Esta recebe o *tweak*, **chave de cifra**, bloco de *plaintext* e o *initial_value* (*iv*). Na implementação da mesma, optou-se pela abordagem de **expansão** da chave de longa duração com o *tweak*, criando a *tweaked_key* (Concatenação da chave com o *tweak*).

Deste modo, *E(w,k,x) = E(K_ x)*, sendo w o *tweak* criado, k a chave de longa duração, x a mensagem a cifrar e K a chave resultante da expansão ($K = w \,//\, k$).

Cifra-se o bloco do *plaintext*, utilizando a cifra AES-256 num modo *CBC* (*cipher block chaining*), que recebe como argumento o *iv* (sendo um modo *standard* e mais seguros que outros, como p.e. o *ECB*). Retorna-se, por fim, o repetivo **criptograma**.

Sendo assim, o *TPBC* segue o esquema:

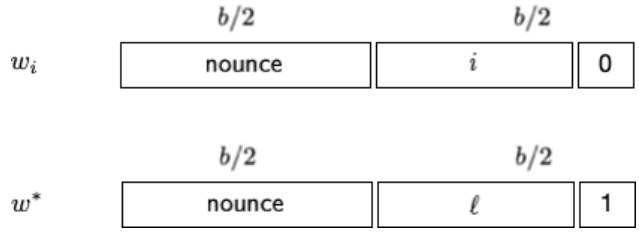


... o *tweak* criado, como é explicado no seguinte método, correponde à combinação da *seed + counter + bit auth.*

1.5.3 Aplicação *TPBC* aos m-1 blocos

Por último,foi implementada a **aplicação do *TPBC* a todos os blocos do *plaintext* (menos o último)**. Todos os blocos, exceto o último, serão cifrados da mesma maneira. Estes blocos serão *input* de o algoritmo *TPBC* juntamente com o tweak e com a chave de longa duração. O *tweak* deve ser diferente para todos os blocos. Este tem o tamanho de um bloco do *plaintext* (16 *bytes*) e será a combinação/concatenação de um *nounce*, um *counter* (CTR) e um bit de autenticação.

Considerando o esquema seguinte, e o tamanho b definido:



... *nounce* - 8 *bytes*, *counter* - 7 *bytes* e *bit* - 1 byte. Este bit de autenticação terá o valor de 0 para a codificação de todos os blocos do plaintext, e o valor de 1 para a criação da tag. Esta tag será usada para autenticação na receção da mensagem.

No final de cada iteração, para o *tweak* variar entre cada bloco, o *counter* é incrementado. Para a criação da *tag* de autenticação executa-se o *XOR*, entre a variável *auth* e cada bloco do *plaintext*. Nesta função, é realizada a operação *XOR* com todos os blocos que serão cifrados, ou seja, todos exceto o último. Este será adicionado numa operação futura.

No final retorna-se o *counter* e a variável *auth* atualizados e o respetivo criptograma (dos *m-1* primeiros blocos).

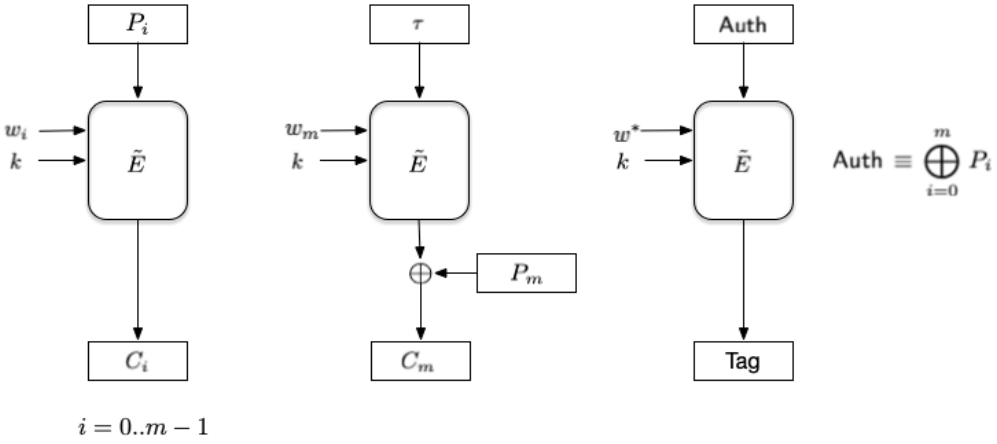
[5]:
```
def pad (block_plaintext, length):
    last_block = block_plaintext[-1] # Get the last block of the list
    len_last_block = len(last_block)

    for _ in range (len_last_block, length): # Adds the value 0 until the size␣
 ↪the last block is 0
        last_block += b"\x00"

    block_plaintext[-1] = last_block

    return block_plaintext, len_last_block

def tpbc (tweak, key, block, iv):
    tweaked_key = tweak + key
    cipher = Cipher(algorithms.AES256(tweaked_key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    ct_block = encryptor.update(block) + encryptor.finalize()
    return ct_block

# nounce: bytes, ctr: bytes, block_plaintext: bytes, key: bytes, auth: bytes
def tweakable_first_blocks(nounce, ctr, block_plaintext, key, auth, iv):
    ct = b""
    zero = b"\x00"

    for elem in (block_plaintext[:-1]):
        tweak = nounce + ctr + zero

        c_i = tpbc(tweak, key, elem, iv)
        ct += c_i

        len_ctr = len(ctr)
        ctr_int = int.from_bytes(ctr, 'big')
        ctr_int += 1
        ctr = ctr_int.to_bytes(len_ctr, 'big')

        aux = b""
        for x,y in zip(auth, elem):
            word = x ^ y
            aux += word.to_bytes(1, 'big')

        auth = aux

    return ctr, auth, ct
```

1.6 Cifrar a mensagem

Como indicado no enunciado, a cifra a criar para o presente problema tem por base *Tweakable Block Ciphers*, nomeadamente o modo **TAE** (*Tweaked Authentication Encryption*), apresentado na última parte do capítulo 1 (lecionado em aula).

Este é um modo de cifra por blocos autenticada, sem o uso de dados associados, que recorre a *TPBC* (*Tweaked Primitive Block Ciphers*) envês de *PBC* (*Primitive Block Ciphers*).

Deste modo, seguiu-se o esquema apresentado:



É notável que existem três processos a fazer no contexto da cifra por blocos, em modo *TAE*, e, considerando um *plaintext* composto por m blocos, de tamanho b cada, tem-se:

1. **Cifrar os primeiros *m - 1* blocos do *plaintext***

- Os primeiro *m-1* blocos (P_i_, t.q. *i = 0..m-1*) serão cifrados a partir da mesma chave de longa duração e de um *tweak* único a cada bloco;
- Estes vão gerar um criptograma, *C_ i*.

2. **Cifrar o último bloco *m***

- O últmo bloco, m, de tamanho n, t.q $n < b$, terá um *padding* adicional no fim ($pad = b - n$), para ter tamanho b, resultando no bloco *P_ m*, de tamanho b, em que os últimos $b - n$ bytes são zeros;
- Para cifrar o bloco é primeiro realizado um processo de cifragem no seu tamanho original n, utilizando a mesma chave de longa duração dos restantes *m - 1* blocos e um *tweak* novo (criando uma máscara);
- Por fim, é realizado um *XOR* da máscara resultante com o último bloco *P_ m*, resultando no criptograma *C_ m*.

3. **Gerar uma *tag* de autenticação a partir da paridade do *plaintext***

- Como se visualiza na imagem, a *tag* é o resultado de cifrar os *bytes* de autenticação.
- Estes *bytes*, *Auth*, são calculados realizando um *XOR* por todos os m blocos.
- Existirá um *byte* de autenticação que terá valor 1, o último, sendo os restantes de valor 0, sendo estes *bytes* os usados para o *tweak*.

1.6.1 Código desenvolvido

Sendo assim, o primeiro passo, após criar o *nounce*, o *counter* e o *initial value* a serem usados na cifragem com *TPBC* (correspondendo aos tamanho inicialmente definidos - $b = 16$ *bytes*), divide-se o *plaintext* em blocos com *NBytes* de tamanho, na variável *block_plaintext*.

Com o método apresentado anteriormente:

```
def pad(block_plaintext, length)
```

...acrescenta-se os restantes *bytes* ao último bloco, retornando o *plaintext* repartido atualizado e o tamanho n do último bloco (*bytes* não nulos da mensagem original).

Cria-se a variável *auth* para ser usada na criação da *tag*. Esta vai ser usada em processos de *XOR*, sendo uma palavra de 16 *bytes*, inicializada com zeros.

Aplica-se a cifra *TPBC* aos primeiros m - *1* blocos, com o método também apresentado e definido anteriormente:

```
def tweakable_first_blocks(nounce, ctr, block_plaintext, key, auth, iv)
```

... obtendo o criptograma *C_i*, a variável de autenticação *auth* atualizada com os *m-1* primeiros blocos e o *counter* resultante.

Cria-se a máscara com o número de *bytes* do último bloco, *lenght_block*, sendo criado o *tweak* do mesmo modo que os outros blocos e, tal como esses, cifrado com o método:

```
def tpbc(tweak, key, block, iv)
```

... posteriormente é realizado o *XOR* com o último bloco, *P_m*, criando o criptograma *C_m*; concatenado, por fim, ao criado inicialmente, C_i_. O criptograma encontra-se pronto.

Antes de terminar o processo, falta realizar o último *XOR*, atualizando a variável *auth* com o último bloco *m*. Termina-se com a criação da *tag*, resultante da cifragem com o método *tpbc* usado nos blocos do *plaintext*, com o *tweak* semelhante (diferindo no último *byte*, de autenticação que agora é 1).

O método resultante

```
def encrypt(msg, key)
```

retorna o criptograma, a *tag*, o *nounce*, o *counter* inicial (antes de ser atualizado), o *pad* (tamanho dos dados do último bloco - n) e o *initial value* usado para a cifra *CBC*.

Concluimos assim uma análise global do algoritmo implementado para que as mensagens introduzidas sejam cifradas com um algoritmo AEAD, que utiliza TPBC.

[6]:
```
def encrypt(msg, key):
    NBytes = 16; size_nounce = 8

    ## Nounce
    digest = hashes.Hash(hashes.SHA3_256())
    nounce_temp = digest.finalize()
    nounce = nounce_temp[:size_nounce] # array of bytes with size 8
```

```
    ## Counter
    ctr_i = os.urandom(size_nounce-1) # array of bytes with size 7
    ctr = ctr_i

    ## Initial value for CBC
    iv = os.urandom(16)

    ## Partitioning plaintext into Nbytes blocks
    block_plaintext = [bytes(msg[i:i+NBytes], 'utf8') for i in range(0,␣
↪len(msg), NBytes)]  # List of block of bytes. Block size 16

    ## Padding last block of plaintext
    block_plaintext, len_last_block = pad(block_plaintext, NBytes) # blocks de␣
↪bytes de tam 16, tamanho original do ultimo bloco

    # i = 0 ... m - 1
    auth = b""
    for _ in range (NBytes): # array of bytes with size 16 bytes
        auth += b"\x00"

    ## Apply TPBC algorithm to all but last block
    ctr, auth, ct = tweakable_first_blocks(nounce, ctr, block_plaintext, key,␣
↪auth, iv) # ctr: bytes; auth: str ; ct: bytes

    # i = m - TPBC on last block
    tweak = nounce + ctr +  b"\x00"
    length_block = len_last_block.to_bytes(16, 'big') # Turns the length of the␣
↪last block into a 16 bytes block
    c_aux = tpbc(tweak, key, length_block, iv) # c_aux: bytes

    c_m = b""
    for x,y in zip(block_plaintext[-1], c_aux):
        word = x ^ y
        c_m += word.to_bytes(1, 'big')

    ## Concatenating last block c_m to the rest of the cipher
    ct += c_m # ct: bytes

    aux = b""
    for x,y in zip(auth, block_plaintext[-1]):
        word = x ^ y
        aux += word.to_bytes(1, 'big')
    auth = aux

    ## Autenticação - Criar a tag para autenticação
    tweak = nounce + ctr + b"\x01"
    tag = tpbc(tweak, key, auth, iv)
```

```
    return {"ct": ct, "tag": tag, "nounce": nounce, "ctr": ctr_i, "pad":␣
↪len_last_block, "iv": iv}
```

1.7 Funções auxiliares para decifragem

As funções criadas para a decifragem da mensagem são o inverso das criadas em cima, para a cifragem.

Neste sentido, tem-se:

1. Eliminação do *bytes* de *padding* do último bloco do *plaintext*;
2. A decifragem (do *CBC*) de um único bloco, utilizando o *tweak* e o *initial value* usados para cifrar;
3. A decifragem dos primeiros (*m - 1*) blocos, utilizando o método em **2.**.

... para o ponto **3.**, ainda tem-se a atualização do valor do *counter* e da var *auth* (exatamente como no processo de cifra), retornando-os.

[7]:
```
# size_block = tau
def unpad(last_block, size_block, NBytes):

    last_block = last_block[:size_block]

    return last_block

def un_tpbc(tweak, key, block, iv):
    tweaked_key = tweak + key
    cipher = Cipher(algorithms.AES256(tweaked_key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plain_block = decryptor.update(block) + decryptor.finalize()
    return plain_block

def undo_tweakable_first_blocks(nounce, ctr, block_ciphertext, key, auth, iv):
    plaintext = b""
    for elem in (block_ciphertext[:-1]):
        tweak = nounce + ctr +  b"\x00"

        c_i = un_tpbc(tweak, key, elem, iv)
        plaintext += c_i

        len_ctr = len(ctr)
        ctr_int = int.from_bytes(ctr, 'big')
        ctr_int += 1
        ctr = ctr_int.to_bytes(len_ctr, 'big')

        aux = b""
        for x,y in zip(auth, c_i):
            word = x ^ y
```

```
            aux += word.to_bytes(1, 'big')

        auth = aux

    return ctr, auth, plaintext
```

1.8 Decifrar mensagem

O processo de decifragem, à semelhança das suas funções auxiliares definidas em cima, trata-se do processo **inverso** do que foi efetuado no âmbito da cifra.

Primeiro, reparte-se o *plaintext* em blocos, com o tamanho originalmente usado - 16 *bytes*. De seguida, decifra-se os primeiros *m-1* blocos e, logo após estes, o último bloco, *m*. Obtém-se o _plaintext (através da concatenação dos *m-1* blocos com o *m*). Por fim, recria-se a autenticação (com a criação da *tag*).

A autenticação é exatamente do mesmo modo que na cifra, para no final comparar as *tags* resultantes. Envia-se um *booleano* para indicar se o processo foi feito com sucesso ou se sofreu repercussões, na forma da variável *tag_ valid*.

[8]:
```
def decrypt(msg, key):
    ct = msg['ct'] # bytes
    tag_rcv = msg['tag']
    nounce = msg['nounce']
    ctr = msg['ctr']
    len_last_block = msg['pad']
    iv = msg['iv']

    NBytes = 16

    block_ciphertext = [ct[i:i+NBytes] for i in range(0, len(ct), NBytes)]  #␣
↪list of block of bytes. Block size 16 bytes

    # i= 0 ... m - 1
    auth = b""
    for _ in range (NBytes): # array of bytes with size 16 bytes
        auth += b"\x00"
    ctr, auth, plaintext = undo_tweakable_first_blocks(nounce, ctr,␣
↪block_ciphertext, key, auth, iv)

    # i = m
    tweak = nounce + ctr + b"\x00"
    length_block = len_last_block.to_bytes(16, 'big')
    c_aux = tpbc(tweak, key, length_block, iv)

    c_m = b""
    for x,y in zip(block_ciphertext[-1], c_aux):
        word = x ^ y
```

```
        c_m += word.to_bytes(1, 'big')

    plaintext += c_m[:len_last_block] # ct: bytes

    aux = b""
    for x,y in zip(auth, c_m):
        word = x ^ y
        aux += word.to_bytes(1, 'big')
    auth = aux

    # Autenticação
    tweak = nounce + ctr + b"\x01"

    tag = tpbc(tweak, key, auth, iv)

    tag_valid = True
    if tag != tag_rcv:
        tag_valid = False

    return plaintext, tag_valid
```

1.9 Funções auxiliares para o canal de comunicação

Encontra-se 3 métodos auxiliares para:

- inicialização dos agentes;
- Envio de mensagens;
- Receção de mensagem.

Os métodos para envio e receção de mensagens têm em consideração o código proporcionado pela equipa docente, em consideração das estruturas *queues*.

O método para **inicialização**:

- Cria os dois pares de chaves pública-privada (para cifrar e assinar);
- Serializa as chaves públicas (para enviar ao *peer*);
- Retorna um tuplo com as chaves e o pacote a enviar ao peer.

...com o nome _init_comm().

É realizada uma **assinatura da chave pública para cifrar** à priori, correpondendo à fase de **confirmação da chave acordada**. Uma vez que esta é uma chave temporária, e usada unicamente para confirmar que a assinatura está correta, não inclui tantas informações para a tornar mais crédula; por exemplo certificados, protocolos, etc...

Sendo assim, a fase de confirmação desta chave de **curta duração** é, essencialmente, a criação de uma assinatura para a avaliar ao ser enviada ao respetivo *peer*.

[9]:
```
def init_comm():
    ## Gerar chaves (cifrar e autenticar)
    prv_cipher_key, pub_cipher_key = generateKeys()

    prv_sign_key, pub_sign_key = generateSignKeys()

    msg_to_sign = pub_cipher_key.public_bytes(encoding=serialization.Encoding.
 ↪Raw,
                                              format=serialization.PublicFormat.Raw
                                             )

    ##Assinar a public cipher key - autenticação da chave acordada
    sign = signMsg(prv_sign_key, msg_to_sign)

    ## Dicionário com a chavess públicas (serializadas)
    msg = {'cipher_key': pub_cipher_key.public_bytes(encoding=serialization.
 ↪Encoding.Raw,
                                              format=serialization.PublicFormat.Raw
                                             ),
           'sign_key': pub_sign_key.public_bytes(encoding=serialization.Encoding.
 ↪Raw,
                                              format=serialization.PublicFormat.Raw
                                             ),
           'sig': sign
            }

    return prv_cipher_key, prv_sign_key, msg

async def send(queue, msg):

    await asyncio.sleep(random.random())

    # put the item in the queue
    await queue.put(msg)

    await asyncio.sleep(random.random())

async def receive(queue):
    item = await queue.get()

    await asyncio.sleep(random.random())
    aux = loads(item)

    return aux
```

1.10 Emitter

O processo do *emitter* passa pelos passos:

- Criar dois pares de chaves privadas/públicas (para **cifrar** e **assinar**);
- Assinar a chave pública de cifragem (com a chave privada de assinar);
- Enviar as chaves públicas (de cifragem e para verificar a assinatura);
- Receber as respetivas chaves públicas do peer (e a assinatura);
- Validar a assinatura;
- Gerar o segredo partilhado para cifragem (chave partilhada - *cipher*)
- Cifrar a mensagem com a chave partilhada (*encrypt*)
- Assinar com a chave privada (para assinar)
- Enviar pacote para o *receiver*

[10]:
```
## Emitter Code
async def emitter(plaintext, queue):

    ## Gerar as chaves (privada e publica) & partilhar com participante
    prv_cipher_key, prv_sign_key, msg = init_comm()

    ## Enviar a chaves públicas para o peer
    print("[E] SENDING PUBLIC KEYS")
    await send(queue, dumps(msg))

    ## Receber as chaves públicas do peer
    msg = await receive(queue)
    print("[E] RECEIVED PEER PUBLIC KEYS")

    pub_peer_cipher = msg['cipher_key']
    pub_peer_sign = msg['sign_key']
    signature = msg['sig']
    # print("[E] Receiver pub_key_cipher: " +str(pub_peer_cipher))
    # print("[E] Receiver pub_key_sign: " +str(pub_peer_sign))
    # print("[E] Receiver signature: " +str(signature))

    try:
        ## Obter a chave pública (Assinatura)
        peer_sign_pubkey = ed448.Ed448PublicKey.from_public_bytes(pub_peer_sign)

        ## Verificar a assinatura da chave pública
        peer_sign_pubkey.verify(signature, pub_peer_cipher)
        print("[E] SIGNATURE VALIDATED")

        ## Criar as chaves partilhadas (cifrar/autenticar)
        cipher_shared = generateShared(prv_cipher_key, pub_peer_cipher)

        print("[E] CIPHER SHARED: "+str(cipher_shared))
```

```
        ## Cifrar a mensagem
        pkg = encrypt(plaintext, cipher_shared)
        print("[E] MESSAGE ENCRYPTED")

        ## Assinar e enviar a mensagem
        pkg_b = dumps(pkg)
        sig = signMsg(prv_sign_key, pkg_b)

        ## a Enviar...
        msg_final = {'sig': sig, 'msg': dumps(pkg)}

        print("[E] SENDING MESSAGE")
        await send(queue, dumps(msg_final))

        print("[E] END")

    except InvalidSignature:
        printf("A assinatura não foi verificada com sucesso!")
```

1.11 Receiver

Os dois agentes, no contexto deste exemplo, são particularmente semelhantes, apenas só alterando a ordem dos eventos. Neste caso, é do *emitter* para o *receiver*.

Sendo assim, os eventos surgem como:

- Criar dois pares de chaves privadas/públicas (para cifrar e assinar);
- Receber as respetivas chaves públicas do peer (e a assinatura);
- Validar a assinatura da chave pública de cifra;
- Gerar o segredo partilhado para cifragem (chave partilhada - *cipher*);
- Enviar as chaves públicas (de cifragem e para verificar a assinatura);
- Receber o criptograma do *emitter*;
- Validar a assinatura do criptograma; - Decifrar o pacote e obter o *plain_text*;
- Imprimir o *plaintext*.

[11]:
```
## Receiver Code
async def receiver(queue):

    ## Gerar as chaves (privada e publica) & partilhar com participante
    prv_cipher_key, prv_sign_key, msg = init_comm()

    ## Receber as chaves publicas do peer
    pub_keys = await receive(queue)

    pub_peer_cipher = pub_keys['cipher_key']
```

```
    pub_peer_sign = pub_keys['sign_key']
    signature = pub_keys['sig']
    # print("[R] Emitter pub_key_cipher: " +str(pub_peer_cipher))
    # print("[R] Emitter pub_key_sign: " +str(pub_peer_sign))
    # print("[R] Receiver signature: " +str(signature))

    try:
        ## Obter a chave pública (Assinatura)
        peer_sign_pubkey = ed448.Ed448PublicKey.from_public_bytes(pub_peer_sign)

        ## Validar a correçaõ da assinatura
        peer_sign_pubkey.verify(signature, pub_peer_cipher)
        print("[R] SIGNATURE VALIDATED")

        ## Gerar shared keys
        cipher_shared = generateShared(prv_cipher_key, pub_peer_cipher)

        ## Enviar as chaves públicas ao peer
        print("[R] SEND PUBLIC KEYS")
        await send(queue, dumps(msg))

        ## Receber criptograma
        print("[R] AWAIT CIPHER")
        ciphertext = await receive(queue)
        print("[R] CIPHER RECEIVED")

        try:
            ## Validar a correção da assinatura
            peer_sign_pubkey.verify(ciphertext['sig'], ciphertext['msg'])
            print("[R] SIGNATURE VALIDATED")

            msg_dict = loads(ciphertext['msg'])

            ## Decifrar essa mensagem
            plain_text, tag_valid = decrypt(msg_dict, cipher_shared)

            if tag_valid == False:
                print("Autenticação falhada!")
                return

            print("[R] MESSAGE DECRYPTED")

            ## Apresentar no terminal
            print("[R] PLAINTEXT: " + plain_text.decode('utf-8'))

            print("[R] END")
```

```
        except InvalidSignature:
            print("The signature wasn't validated correctly! - Cipher")

    except InvalidSignature:
        print("The signature wasn't validated correctly! - Cipher key")
```

1.12 Teste de execução

Encontra-se na célula abaixo um exemplo de execução da comunicação entre um *emitter* e um *receiver*, acompanhada das respetivas autenticações do agentes, da mensagem, bem como a cifragem/decifragem da última.

O método:

```
def ex3(msg)
```

... funciona como um método *main* para a execução; recebendo como argumento a **mensagem**.

[12]:
```
def ex3(msg):
    loop = asyncio.get_event_loop()
    queue = asyncio.Queue(10)
    asyncio.ensure_future(emitter(msg, queue), loop=loop)
    loop.run_until_complete(receiver(queue))
```

[13]:
```
ex3("Hello World!")
```

```
[E] SENDING PUBLIC KEYS
[R] SIGNATURE VALIDATED
[R] SEND PUBLIC KEYS
[E] RECEIVED PEER PUBLIC KEYS
[E] SIGNATURE VALIDATED
[E] CIPHER SHARED: b'W\x9e\x1e\x93\xd6\x01\x9e\x95\x90\xf0\x97\xdec\x00\xc2\xc8'
[E] MESSAGE ENCRYPTED
[E] SENDING MESSAGE
[E] END
[R] AWAIT CIPHER
[R] CIPHER RECEIVED
[R] SIGNATURE VALIDATED
[R] MESSAGE DECRYPTED
[R] PLAINTEXT: Hello World!
[R] END
```

[14]:
```
ex3("HELLO WORLD! THIS IS JUST A TEST OF A AEAD ALGORITHM.")
```

```
[E] SENDING PUBLIC KEYS
[R] SIGNATURE VALIDATED
[R] SEND PUBLIC KEYS
[R] AWAIT CIPHER
[E] RECEIVED PEER PUBLIC KEYS
[E] SIGNATURE VALIDATED
```

```
[E] CIPHER SHARED: b'\x1dY\x1d\xb2X:a\x87\x0bGG\xf13}\xa3@'
[E] MESSAGE ENCRYPTED
[E] SENDING MESSAGE
[R] CIPHER RECEIVED
[R] SIGNATURE VALIDATED
[R] MESSAGE DECRYPTED
[R] PLAINTEXT: HELLO WORLD! THIS IS JUST A TEST OF A AEAD ALGORITHM.
[R] END
```