EXERCICIO 2

Neste exercicio é proposto que usemos o “package” Cryptography para:

1. Implementar uma AEAD com “Tweakable Block Ciphers” conforme está descrito na última secção do texto +Capítulo 1: Primitivas Criptográficas Básicas. A cifra por blocos primitiva, usada para gerar a “tweakable block cipher”, é o AES-256 ou o ChaCha20.
2. Use esta cifra para construir um canal privado de informação assíncrona com acordo de chaves feito com “X448 key exchange” e “Ed448 Signing&Verification” para autenticação dos agentes. Deve incluir uma fase de confirmação da chave acordada.

Para realizarmos este exercicio utilizamos as bibliotecas Asyncio para criar os servidores de emissao e resposta, utilizamos o cryptography para fazer a cifragem e decifragem dos textos.

In [1]:
```
import asyncio
import secrets
from ascon import _ascon
import os
import random
from pickle import dumps, loads
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.asymmetric import x448, ed448
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448Priv

import nest_asyncio

nest_asyncio.apply()
```

Para a geração das chaves privadas e publicas, usadas para que se possa criar uma chave partilhada e permitir o compartilhamento das chaves publicas, foi utilizado a curva448.

A curva448 é uma curva eliptica Diffie-Hellman, usada para permitir que dois agentes criem uma chave partilhada segura para ser usada em um canal inseguro

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

    # Generate public key thorugh private key
    peer_public_key = private_key.public_key()

    return private_key, peer_public_key
```

Após termos as chaves de ambos os agentes, podemos criar o acordo. A Shared Key é criada e passa por uma função de derivação para se tornar mais segura.

In [3]:
```
def generateShared(private_key, peer_public_key):

    peer_cipher_key = x448.X448PublicKey.from_public_bytes(peer_publi

    # Gerar uma chave partilha para cifra
    shared_key = private_key.exchange(peer_cipher_key)

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

    return derived_key
```

Para a assinatura de uma mensagem, utilizamos a Ed448, que utiliza uma curva Edwards-curve Digital Signature Algorithm (EdDSA). Foi projetada para ser mais rapida forma de assinatura digital, sem sacrificar a segurança.

É necessario um novo par de chaves para o participante que deseja assinar uma mensagem.

In [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
```

In [5]:
```
def signMsg(prv_key, msg):

    signature = prv_key.sign(msg)

    return signature
```

Para facilitar a implementação do código, foi feita a função de iniciar agentes, nesta função, são geradas as chaves publicas e privadas e chaves publicas e privadas de assinatura.

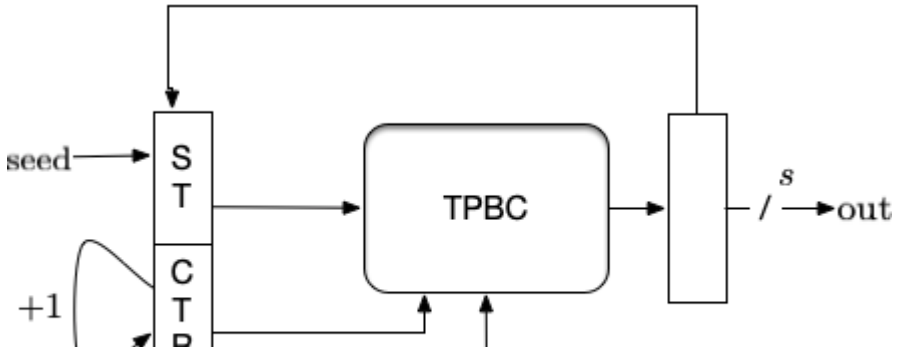
In [6]:
```
def init_agents():
    private_cipher_key, public_cipher_key = generateKeys()
    private_sign_key, public_sign_key = generateSignKeys()

    msg_to_sign = public_cipher_key.public_bytes(encoding=serializati
                                                 format=serialization.PublicFo
                                                )

    signed_message = signMsg(private_sign_key,msg_to_sign)
    content = {'cipher_key': public_cipher_key.public_bytes(encoding=
                                                 format=serialization.PublicFo
                                                ),
           'sign_key': public_sign_key.public_bytes(encoding=seriali
                                                 format=serialization.PublicFo
                                                ),
           'message': signed_message}
    return private_cipher_key, private_sign_key, content
```

As funções abaixo utilizam o Asyncio para enviar e receber uma mensagem.

In [7]:
```
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
```

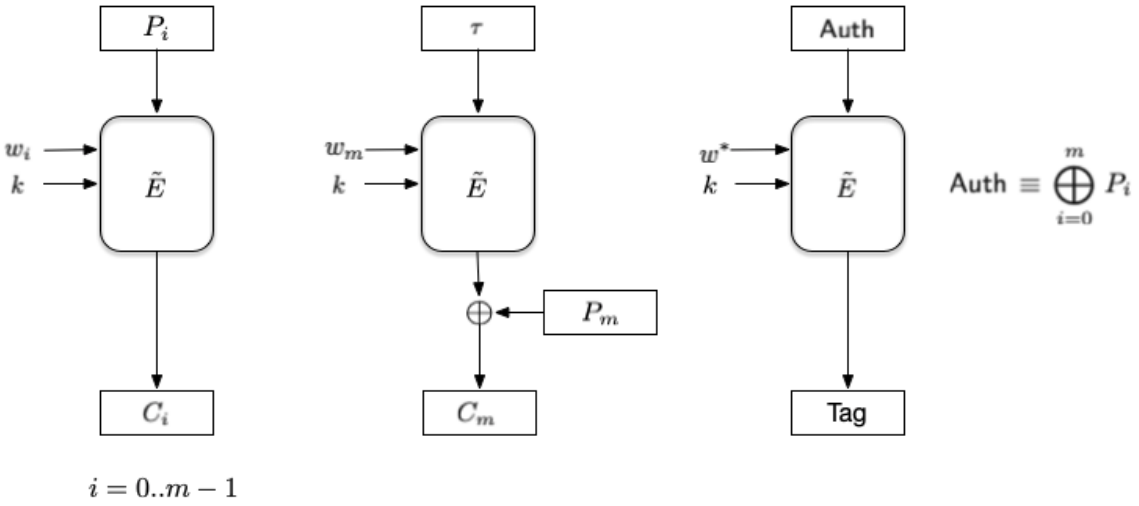
Na função *tweak_blocks_tpbc* é implementada uma função para cifrar blocos de dados usando o TPBC (Tweakable Primitive Block Cipher). A abordagem adotada envolve a expansão da chave de longa duração com um tweak específico para criar uma chave tweakada. O TPBC utiliza a cifra AES-256 no modo CBC (Cipher Block Chaining), com um vetor de inicialização (iv) padrão para maior segurança.

A aplicação do TPBC aos m-1 blocos do plaintext consiste em cifrar cada bloco, exceto o último, usando tweaks diferentes. O tweak é uma combinação de um nonce, um contador (CTR) e um bit de autenticação, resultando em um tamanho de bloco de 16 bytes.



Durante cada iteração, o contador é incrementado para variar o tweak entre os blocos. A criação da tag de autenticação envolve a operação XOR entre a variável "auth" e cada bloco do plaintext, com o bit de autenticação assumindo o valor 0 para codificação e 1 para a criação da tag.

Ao final de cada iteração, a função retorna o contador e a variável "auth" atualizados, juntamente com o criptograma dos m-1 primeiros blocos. O último bloco será processado em uma etapa subsequente. Essa abordagem visa garantir autenticação na recepção da mensagem.



In [8]:
```
def tweak_blocks_tpbc(nounce, counter, plaintext_blocks, key, auth, i
    cyphered_text = b""
    zero = b"\x00"

    for elem in (plaintext_blocks):
        tweak = nounce + counter + zero

        cyphered_block = tpbc(tweak, key, elem, iv)
        cyphered_text += cyphered_block

        counter += 1

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

        auth = aux

    return counter, auth, cyphered_text

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

Na função padding o plaintext é divido em blocos de mesmo tamanho, porém o ultimo bloco

poderá ter o tamanho menor e por isso tem de ser preenchido com 0, o que é feito na função abaixo, com o \x00 e retorna o bloco e o tamanho do bloco.

Já na função unpad é feita a limpeza destes 0's e é retornado o texto limpo.

In [9]:
```
def padding(block, size):
    len_block = len(block)

    for _ in range (len_block, size): # Adds the value 0 until the si
        block += b"\x00"

    return block, len_block

def unpad(last_block, size_block):

    clean_text = last_block[:size_block]

    return clean_text
```

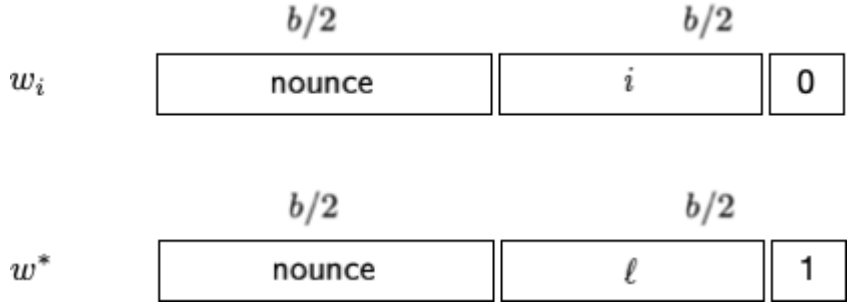
As funções *un_tpbc* e *undo_tweakable_first_blocks* são funções auxiliares para a decifragem da mensagem.

In [10]:
```
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
```

In [11]:
```
def undo_tweakable_first_blocks(nounce, counter, block_ciphertext, ke
    plaintext = b""
    for elem in (block_ciphertext[:-1]):
        tweak = nounce + counter +  b"\x00"

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

        counter +=1

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

        auth = aux

    return counter, auth, plaintext
```

A função enc_txt realiza a cifragem da mensagem, isto é feito utilizando o TPBC. A função recebe uma chave e um texto limpo. O código esta estruturado para utilizar 16 bytes que é o tamanho dos blocos utilizados no AES-256.

Inicialmente são definidos alguns parametros, como o tamanho do bloco e o tamanho do nounce. O nounce é gerado utilizando o hash SHA-256 e reduzido para o tamanho de 8 bytes de maneira a ser posteriormente adicionado ao tweak. Além disso também criamos um contador para ser utilizado no tweak e um vetor de inicialização iv.



O texto original é dividido em blocos de tamanhos iguais para a cifra AES-256, e o ultimo bloco é preenchido com bytes de padding conforme função pad acima mencionada.

O bloco de autenticação é inicializado com bytes nulos e então o código itera sobre os blocos, excluindo o ultimo para realizar a cifragem, utilizando a função *tweak_blocs_tpbc*. Este bloco é atualizado através de uma operação de XOR entre os seus bytes e os bytes do ultimo bloco e o ultimo bloco cifrado é concatenado com o texto cirfrado total.

Por fim é criado um novo tweak para gerar a tag de autenticação que é obtida através da função *tbpc* e aplicada aos dados finais. O resultado da função é um dicionario contendo o texto cifrado, a tag de autenticação, o nonce, o contador inicial, o tamanho do padding no ultimo bloco e o vetor de inicialização.

In [12]:
```
def enc_txt(key, plaintext):
    block_size = 16  # 16-byte blocks for AES256
    nounce_size = 8

    nounce_temp = hashes.Hash(hashes.SHA256()).finalize()
    nounce = nounce_temp[:nounce_size] # 8-byte array for the tweak

    initial_counter = os.urandom(nounce_size-1) # 8-byte array for t
    counter = initial_counter
    iv = os.urandom(16) # initial value for CBC

    plaintext_blocks = divide_into_blocks(plaintext, block_size)#div

    last_block, last_block_size = padding(bytes(plaintext_blocks[-1],

    auth = b""
    for _ in range (block_size): # create the authentication block
        auth += b"\x00"

    counter, auth, encrypt_text = tweak_blocks_tpbc(nounce, counter,

    tweak = nounce + counter +  b"\x00" #create the tweak for the las
    length_block = last_block_size.to_bytes(16, 'big') # Turns the le
    encrypt_mask = tpbc(tweak, key, length_block, iv) # creates the

    encrypt_block = b""
    for x,y in zip(last_block, encrypt_mask): #XOR the last block to
        word = x ^ y
        encrypt_block += word.to_bytes(1, 'big')

    encrypt_text += encrypt_block #concant the last block to the res

    aux = b""
    for x,y in zip(auth, last_block): #XOR the last block with the au
        word = x ^ y
        aux += word.to_bytes(1, 'big')
    auth = aux

    #
    tweak = nounce + counter + b"\x01" # create authentication tag
    tag = tpbc(tweak, key, auth, iv)

    return {"encrypt_text": encrypt_text, "tag": tag, "nounce": nounc
```

O processo de decifragem, com auxilio das suas funções auxiliares trata-se do processo inverso do que foi feito na cifragem.

Primeiro divide-se os blocos cifrados em texto cifrado, tag, nounce, counter, last_block_size e o vetor inicial (iv). Os blocos possuem tamanho de 16 bytes.

No final, a tag é comparada com a tag feita na cifragem, de modo a indicar que a mensagem não foi alterada no meio de transmissão.

In [13]:
```
def dec_txt(key, encrypt_blocks):
    encrypt_text = encrypt_blocks['encrypt_text'] # bytes
    tag_rcv = encrypt_blocks['tag']
    nounce = encrypt_blocks['nounce']
    counter = encrypt_blocks['counter']
    last_block_size = encrypt_blocks['pad']
    iv = encrypt_blocks['iv']

    block_size = 16

    block_ciphertext = divide_into_blocks(encrypt_text, block_size)

    # i= 0 ... m - 1
    auth = b""
    for _ in range (block_size): # array of bytes with size 16 bytes
        auth += b"\x00"
    counter, auth, plaintext = undo_tweakable_first_blocks(nounce, c

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

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

    plaintext += unpad(plain_block, block_size) # ct: bytes

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

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

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

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

    return plaintext, tag_valid
```

Função auxiliar da cifragem e decifragem, para dividir em blocos de mesmo tamanho o texto de entrada.

In [14]:
```
def divide_into_blocks(text, block_size):
    blocks = []
    for i in range(0, len(text), block_size):
        block = text[i:i + block_size]
        blocks.append(block)
    return blocks
```

O emitter é o responsável por:

- Passar pela iniciação dos agentes;
- Enviar as chaves publicas;
- Receber as chaves publicas;
- Criar as chaves partilhadas;
- Cifrar e assinar a mensagem;
- Enviar a mensagem.

O Receiver é o responsável por:

- Criar as chaves publicas e privadas;
- Enviar e receber as chaves publicas;
- Verificar a assinatura da mensagem;
- Decifrar a mensagem;
- Exibir a mensagem.

O main() tem por função definir a mensagem plaintext que passará pelo processo de cifragem e decifragem, e fazer com que os servidores (Emitter e Receiver) executem suas funções de forma assincrona, utilizando a biblioteca Asyncio.

In [15]:
```
async def emitter(plaintext, queue):

    # Emitter's keys
    emitter_private_cipher_key, emitter_private_sign_key, content = i

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

    ## Receber as chaves públicas do peer
    data = await receive(queue)

    print("[E] RECEIVED PEER PUBLIC KEYS")

    pub_peer_cipher = data['cipher_key']
    pub_peer_sign = data['sign_key']
    signature = data['message']
    # 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

        ## 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(emitter_private_cipher_key, pu

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

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

        ## Assinar e enviar a mensagem
        pkg_b = dumps(pkg)
        sig = signMsg(emitter_private_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:
        print("A assinatura não foi verificada com sucesso!")

# Receiver
async def receiver(queue):
    receiver_cipher, receiver_sign, content = init_agents()

    ## 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['message']
    # 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

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

        ## Gerar shared keys
        cipher_shared = generateShared(receiver_cipher, pub_peer_ciph

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

        ## 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['ms
            print("[R] SIGNATURE VALIDATED")

            msg_dict = loads(ciphertext['msg'])

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

            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



async def main():
    loop = asyncio.get_event_loop()
    queue = asyncio.Queue(10)
    asyncio.ensure_future(emitter("Hello World!", queue), loop=loop)
```

```
    loop.run_until_complete(receiver(queue))

if __name__ == "__main__":
    asyncio.run(main())
```

```
[E] SENDING PUBLIC KEYS
[R] SIGNATURE VALIDATED
[R] SEND PUBLIC KEYS
[E] RECEIVED PEER PUBLIC KEYS
[E] SIGNATURE VALIDATED
[E] CIPHER SHARED: b'\xc1\xa7\x1d>i\xb5\xc4Z\x82S\xe3\xac\xd3\xab\x
c6\xf8'
[E] MESSAGE ENCRYPTED
[E] SENDING MESSAGE
[R] AWAIT CIPHER
[R] CIPHER RECEIVED
[R] SIGNATURE VALIDATED
[R] MESSAGE DECRYPTED
[R] PLAINTEXT: Hello World!
[R] END
[E] END
```