Exercicio 1

O mecanismo *AEAD* procura garantir a autenticação tanto do texto cifrar como o texto a ser cifrado. As *tweakable primitive block ciphers* têm dois parâmetros de controlo, a chave em si e um *tweak*, que vai variando.

O primeiro passo para a resolução deste exercicio passou por desenvolvermos o ambiente de comunicação assíncrona.

> De modo a se obter um contexto de envio e receção de mensagens, desenvolvemos:
>
> - Um Emitter
> - Um Receiver

Cada um destes foi desenvolvido tendo em conta a necessidade da implementação dos diferentes acordos de chaves.

In [ ]:
```
%pip install cryptography
```

In [ ]:
```
import os
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives import hmac, hashes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
from cryptography.hazmat.backends import default_backend
```

O Emitter, sendo o interveniente que envia a mensagem, vai ter diferentes atributos correlacionados com a cifragem por blocos usando *tweaks* mas também com as diferentes chaves e assinaturas usadas na autenticação.

In [ ]:

```
BLOCK = 8

class emitter:

    # ATTRIBUTES

    tweak=b"" # the tweak
    message = b"Bom dia" # the message to be transmited
    mac = b""

    X448_private_key = b""
    X448_public_key = b""
    X448_shared_key = b""

    assinatura = b"Signing Message" # assinatura
    Ed448_private_key = b""
    Ed448_public_key = b""
    signature = b"" # the encoded signature after mixing it up with the pr


    ck = b"" # full key aka the one who will cipher
```

- Geração das chaves privadas e públicas e da assinatura utilizadas no esquema de assinatura da curva elíptica de Edwards - ED448

In [ ]:

```
    # geracao da chave privada
    def ed448privateKeygen(self):
        self.Ed448_private_key = Ed448PrivateKey.generate()

    # geracao da chave publica a partir da chave privada
    def ed448publicKeygen(self):
        self.Ed448_public_key = self.Ed448_private_key.public_key()

 # assinatura é gerada a partir mensagem de assinatura sendo depois defini
    def ed448signatureGen(self):
        self.signature = self.Ed448_private_key.sign(self.assinatura)
```

- Geração da chave privada, pública e partilhada utilizada na autenticação por troca de chaves X448.
- A chave partilhada é gerada com o auxílio de uma KDF!

In [ ]:

```
    # geracao da chave privada
    def privateKeyGenX448(self):
        self.X448_private_key = X448PrivateKey.generate()

    # geracao da chave publica do emitter
    def publicKeyGenX448(self):
        self.X448_public_key = self.X448_private_key.public_key()

    # geracao da chave partilhada a partir da derivaçao da chave publica d
    def sharedKeyGenX448(self, peerPublickey): # esta public key é referen
        key = self.X448_private_key.exchange(peerPublickey)

        self.X448_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake assinatura',
        ).derive(key)
```

De modo a verificar o acordo entre chaves é necessário cifrar a chave de forma a que a mesma não possa ser verificada por terceiros, apenas pelo **receiver**. Para isso definiu-se o **key_agree** que é cifrada usando o algoritmo *ChaCha20* ao qual se adiciona um nonce de 16bits

In [ ]:

```
    def key_agree(self):

        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cifrador = Cipher(algorithm, mode=None).encryptor()
        ct = nonce + cifrador.update(self.X448_shared_key)
        return ct
```

Após isso resta apenas definir o processo de padding e o respetivo processo de cifragem por blocos, extremamente similar ao que foi feito no anterior trabalho prático.

In [ ]:

```
    def pad_divide(self,message):
        x = []
        for i in range (0,len(message), BLOCK):
            x.append(message[i:i+BLOCK])
        return x


    def cipher(self):
        ciphertext = b''

        padder = padding.PKCS7(64).padder()
        padded = padder.update(self.message) + padder.finalize()

        p = self.pad_divide(padded)
        for x in range (len(p)): # Percorre blocos do texto limpo
            for bloco, byte in enumerate(p[x]): # Percorre bytes do bloco
                ciphertext += bytes([byte ^ self.ck[x:(x+1)*BLOCK][bloco]]



        h = hmac.HMAC(self.ck, hashes.SHA256())
        h.update(ciphertext)
        self.mac = h.finalize()


        complete_ct = self.mac + ciphertext

        return complete_ct
```

Apresenta-se de seguida o Receiver, que vai receber a mensagem enviada pelo Emitter.

In [ ]:

```
class receiver:

    X448_private_key = b""
    X448_public_key = b""
    X448_shared_key = b""

    tweak = b""
    ck = b""

    assinatura = b"Signing Message"
```

- Geração da chave privada, pública e partilhada utilizada na autenticação por troca de chaves X448.
- A chave partilhada é gerada com o auxílio de uma KDF!

In [ ]:

```
    def privateKeyGenX448(self):
        self.X448_private_key = X448PrivateKey.generate()

    def publicKeyGenX448(self):
        self.X448_public_key = self.X448_private_key.public_key()

    def sharedKeyGenX448(self, X448_emitter_public_key):
        key = self.X448_private_key.exchange(X448_emitter_public_key)

        self.X448_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake assinatura',
        ).derive(key)
```

O receiver recebe a chave pública do emitter e a mensagem de assinatura e a partir dela deve confirmar que corresponde à assinatura que

In [ ]:

```
    def check_Ed448_signature(self, assinatura, public_key):
        try:
            public_key.verify(assinatura, self.assinatura)
            print("Sucesso na autenticação da assinatura da curva eliptica
        except cryptography.exceptions.InvalidSignature:
            print("Erro na autenticação da assinatura da curva eliptica de
```

De forma a verificar a chave cifrada da key exchange X448 produzida pelo Emitter definiu-se a seguinte funcao que indica se as chaves realmente correspondem.

In [ ]:

```
    def confirm_key_agreement(self, ct):

        nonce = ct[:16]
        key = ct[16:]

        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)

        decifrador = cipher.decryptor()
        key = decifrador.update(key)

        if key == self.X448_shared_key:
            print("CONFIRMAÇÃO POSITIVA DA VERIFICAÇÃO DO X448 KEY EXCHANGI
        else:
            print("verificacao negativa DO X448 KEY EXCHANGE")

        return
```

O processo de decifragem é analogo ao de cifragem. Será explicado em detalhe mais à frente

In [ ]:

```
    def verify_Auth(self, ck,message, signature):
        h = hmac.HMAC(ck, hashes.SHA256())

        h.update(message)

        try:
            h.verify(signature)
            return True
        except:
            return False

    def decipher(self, ciphertext,receiver):

        complete_key = receiver.ck
        signature = ciphertext[:32]
        ct = ciphertext[32:]


        if self.verify_Auth(complete_key, ct, signature):
            print("Autenticação do criptograma")
        else:
            print("Falha na autenticação do criptograma!")


        clear_text = b''

        # XOR METODO
        for x in range (0,len(ct),8):
            b = ct[x:x+8]
            for index, byte in enumerate(b):
                clear_text += bytes([byte ^ self.ck[x*8:(x+1)*8][index]])

        # Algoritmo para retirar padding para decifragem
        unpadder = padding.PKCS7(64).unpadder()

        # Retira bytes adicionados
        unpadded_message = unpadder.update(clear_text) + unpadder.finalize

        print(unpadded_message.decode("UTF-8"))
```

De forma a se poder compilar a exemplificação, fica aqui todo o código disposto.

In [ ]:

```
BLOCK = 8

class emitter:

    # ATTRIBUTES

    tweak=b"" # the tweak
    message = b"Bom dia" # the message to be transmited
    mac = b""

    X448_private_key = b""
    X448_public_key = b""
    X448_shared_key = b""

    assinatura = b"Signing Message" # assinatura
    Ed448_private_key = b""
```

```
    Ed448_public_key = b""
    signature = b"" # the encoded signature after mixing it up with the pr


    ck = b"" # full key aka the one who will cipher


    # assinatura é gerada a partir mensagem de assinatura sendo depois def
    def ed448signatureGen(self):
        self.signature = self.Ed448_private_key.sign(self.assinatura)

    # geracao da chave privada
    def ed448privateKeygen(self):
        self.Ed448_private_key = Ed448PrivateKey.generate()

    # geracao da chave publica a partir da chave privada
    def ed448publicKeygen(self):
        self.Ed448_public_key = self.Ed448_private_key.public_key()


    # x448

    # geracao da chave privada
    def privateKeyGenX448(self):
        # Generate a private key for use in the exchange.
        self.X448_private_key = X448PrivateKey.generate()

    # geracao da chave publica do emitter
    def publicKeyGenX448(self):
        self.X448_public_key = self.X448_private_key.public_key()

    # geracao da chave partilhada a partir da derivaçao da chave publica d
    def sharedKeyGenX448(self, peerPublickey): # esta public key é referen
        key = self.X448_private_key.exchange(peerPublickey)

        self.X448_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake assinatura',
        ).derive(key)


    def key_agree(self):

        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cifrador = Cipher(algorithm, mode=None).encryptor()
        ct = nonce + cifrador.update(self.X448_shared_key)
        return ct


    def pad_divide(self,message):
        x = []
        for i in range (0,len(message), BLOCK):
            x.append(message[i:i+BLOCK])
        return x


    def cipher(self):
        ciphertext = b''
```

```
        padder = padding.PKCS7(64).padder()
        padded = padder.update(self.message) + padder.finalize()

        p = self.pad_divide(padded)
        for x in range (len(p)): # Percorre blocos do texto limpo
            for bloco, byte in enumerate(p[x]): # Percorre bytes do bloco
                ciphertext += bytes([byte ^ self.ck[x:(x+1)*BLOCK][bloco]]



        h = hmac.HMAC(self.ck, hashes.SHA256())
        h.update(ciphertext)
        self.mac = h.finalize()

        return  self.mac + ciphertext
```

In [ ]:

```
class receiver:

    X448_private_key = b""
    X448_public_key = b""
    X448_shared_key = b""

    tweak = b""
    ck = b""  # complete key aka the one who will cipher

    assinatura = b"Signing Message"

    #acordo de chaves feito com "X448 key exchange" e "Ed448 Signing&Verif


    def check_Ed448_signature(self, signature, public_key):
        try:
            public_key.verify(signature, self.assinatura)
            print("Sucesso na autenticação ed448")
        except cryptography.exceptions.InvalidSignature:
            print("Erro na autenticação da assinatura ed448")


    # Generate a private key for use in the exchange.
    def privateKeyGenX448(self):
        self.X448_private_key = X448PrivateKey.generate()

    def publicKeyGenX448(self):
        self.X448_public_key = self.X448_private_key.public_key()

    def sharedKeyGenX448(self, X448_emitter_public_key):
        key = self.X448_private_key.exchange(X448_emitter_public_key)

        self.X448_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake assinatura',
        ).derive(key)


    def create_ck(self):
```

```
        print("A criar a chave completa.")
        self.ck = self.X448_shared_key + self.tweak

    def confirm_key_agreement(self, ct):

        nonce = ct[:16]
        key = ct[16:]

        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)

        decryptor = cipher.decryptor()
        d_key = decryptor.update(key)

        if d_key == self.X448_shared_key:
            print("CONFIRMAÇÃO POSITIVA DA VERIFICAÇÃO DO X448 KEY EXCHANGE
        else:
            print("verificacao negativa DO X448 KEY EXCHANGE")

        return



    def verify_Auth(self, ck,message, signature):
        h = hmac.HMAC(ck, hashes.SHA256())

        h.update(message)

        try:
            h.verify(signature)
            return True
        except:
            return False

    def decipher(self, ciphertext,receiver):

        complete_key = receiver.ck
        signature = ciphertext[:32]
        ct = ciphertext[32:]


        if self.verify_Auth(complete_key, ct, signature):
            print("Autenticação do criptograma")
        else:
            print("Falha na autenticação do criptograma!")


        clear_text = b''

        # XOR METODO
        for x in range (0,len(ct),8):
            b = ct[x:x+8]
            for index, byte in enumerate(b):
                clear_text += bytes([byte ^ self.ck[x*8:(x+1)*8][index]])

        # Algoritmo para retirar padding para decifragem
        unpadder = padding.PKCS7(64).unpadder()

        # Retira bytes adicionados
        unpadded_message = unpadder.update(clear_text) + unpadder.finalize
```

```
        return unpadded_message.decode("UTF-8")
```

Fica aqui todo o processo de autenticação, verificação de assinaturas, e definição da cifragem do texto por via de tweakable block ciphers

In [ ]:

```
def ed448_setup(emitter):
    emitter.ed448privateKeygen()
    emitter.ed448signatureGen()
    emitter.ed448publicKeygen()


def x448keys(emitter,receiver):
    emitter.privateKeyGenX448()
    receiver.privateKeyGenX448()
    # chaves privadas geradas 1o porque as publicas sao geradas a partir d
    emitter.publicKeyGenX448()
    receiver.publicKeyGenX448()

def sharedkeygen(emitter,receiver):
    emitter.sharedKeyGenX448(receiver.X448_public_key)
    receiver.sharedKeyGenX448(emitter.X448_public_key)


def tweakgen(key):

    nounce = os.urandom(16)
    algorithm = algorithms.ChaCha20(key, nounce)
    cipher = Cipher(algorithm, mode=None)
    ct = cipher.encryptor()
    tweak = ct.update(b"Tweakable")
    return tweak

emitter = emitter()
receiver = receiver()
ed448_setup(emitter)
# Verificação da assinatura
receiver.check_Ed448_signature(emitter.signature, emitter.Ed448_public_key
x448keys(emitter,receiver)
sharedkeygen(emitter,receiver)

# Verificação de se as chaves foram bem acordadas X448
keyCT = emitter.key_agree() # key cipher for 2 side agreement
receiver.confirm_key_agreement(keyCT)

# GERACAO DE TWEAK
tweak = tweakgen(emitter.X448_shared_key)
emitter.tweak = tweak
receiver.tweak = tweak


emitter.ck = emitter.X448_shared_key + emitter.tweak
receiver.ck = receiver.X448_shared_key + receiver.tweak


ciphertext = emitter.cipher()
plaintext = receiver.decipher(ciphertext,receiver)

print("CT: " , ciphertext)
print("PT: ", plaintext)
```