In [229…

```
from ast import Try
from hmac import digest
from inspect import signature
import re
from select import select
import time
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import cryptography.exceptions
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import x448
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateK
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, mo
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.serialization import load_der_private
from cryptography.hazmat.primitives.serialization import load_pem_private
from cryptography.hazmat.primitives.serialization import load_pem_public_
from cryptography.hazmat.primitives.serialization import load_der_public_
```

Introdução

Para este trabalho foi-nos pedido uma AEAD com tweakable block chipers. O grupo reaproveitou as classes do trabalho anterior e fez as respetivas modificações

Emitter

No emitter são criadas cinco chaves três delas que usam a X488 e as restantes são para autenticação e verificação. As chaves privadas são criadas com os respetivos metodos de geração e as chaves publicas através destas.

Para a shared key utiliza-se uma derivação da chave publica do outro interviente. Antes de proceder-se à codificação da mensagem autentica-se a chave partilhada entre ambos para saber que a mensagem só o recetor desta a pode descodificar, após isto prodece-se a codificação. Esta consiste em gerar blocos de 16 bits, 8 são respetivos ao tweak e outros 8 a mensagem em si. Aplica-se o padding e após isto a cada bloco de texto vai ser concatenado um tweak que é composto por um nounce, um contador e uma tag como é referido na documentação.

Aplica-se a cifra usando este tweak e a chave partilhada e adiciona-se ao texto cifrado. Na documentação tambem referia que o ultimo bloco era tratado de forma diferente sendo a tag igual a 1 e em fez de se aplicar a cifra era aplicada uma mascara xor, mascara esta que é composta pelo tweak respetivo ao bloco e a chave partilhada.

A seguir utiliza-se o Ed448 para autenticar a assinatura gerada do HMAC e concatena-se tudo numa mensagem separada por um separador definido pelo grupo e envia-se ao recetor

In [230…

```
class Alice:
    def __init__(self):
        self.private_key=None
        self.public_key=None
        self.shared_key=None
        self.private_key_ED=None
        self.public_key_ED=None

    #gera-se a chave publica
    def generate_key(self):
        self.private_key = X448PrivateKey.generate()


    def generate_key_Ed(self):
        self.private_key_ED=Ed448PrivateKey.generate()

    def generate_public_key_ED(self):
        self.public_key_ED=self.private_key_ED.public_key()


    def generate_public_key(self):
       self.public_key=self.private_key.public_key()

    def set_public_key(self,key):
        self.public_key=key


    #gera-se a chave partilhada entre ambos para o protocolo DH
    def generate_derived_key(self,bob_public_key):
        derived_key=self.private_key.exchange(bob_public_key)
        self.shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=64,
            salt=None,
            info=b'handshake data',
).derive(derived_key)

    #gera-se a signature para autenticação da mensagem através do HMAC
    def authenticate(self):
        h=hmac.HMAC(self.shared_key,hashes.SHA256())
        h.update(self.shared_key)
        signature=h.finalize()
        return signature

    def authenticateEd(self,message):
        return self.private_key_ED.sign(message)


    def encrypt1(self,message):
        tamanho = len(message)
        padder = padding.PKCS7(64).padder()
        padded = padder.update(message) + padder.finalize()
        cipher_text = b''
        contador = 0
        for a in range(0,len(padded),16):
            block=padded[a:a+16]
            if (a+16+1 > len(padded)):
                tweak = self.generate_tweak(tamanho,1)
                cipher_text = cipher_text+ tweak
                bloco_com_mascara = b''
                for index, byte in enumerate(block):
                    mascara = self.shared_key + tweak
                    bloco_com_mascara += bytes([byte ^ mascara[0:16][0]])
```

```
                cipher_text = cipher_text+ tweak + ct
            contador += 1



        return cipher_text



    #gera-se a signature através do HMAC e contatena-se a mensagem com o
    # a associated_data que não é encriptada e data que é encriptada
    def send_message(self,data):
        signatureHMAC=self.authenticate()
        ct= self.encrypt1(data)
        signatureEd=self.authenticateEd(signatureHMAC)
        return signatureEd+separador+signatureHMAC+separador+ct

    def generate_tweak(self,contador,tag):
        nounce= os.urandom(8)
        return nounce + contador.to_bytes(7,byteorder = 'big') + tag.to_b
```

Reciever

A classe reciever é analoga a emitter, tirando a parte de descodificação e a parte de autenticação que agora apenas se faz a verificação das assinaturas enviadas.

Antes de proceder a descodificação da mensagem verificamos se as assinaturas estão corretas, tanto a do HMAC como a ED448.

Na descodifição acontece o contrário da codificação onde sabemos que em cada bloco de 32 bits 16 vão ser respetivos ao tweak e os outros 16 ao texto.

Aplicamos um parser ao tweak e retiramos as suas componentes, assim temos acesso a tag e ao contador que vão ser importantes para saber se tudo correu bem.

Sendo assim percorremos o texto codificado de 32 em 32 bits e aplicamos a descodificação com a cifra gerada com o tweak do respetivo bloco e descoficamos o texto.

Como o ultimo bloco foi aplicado uma mascara, quando a tag é 1, ou seja, quando é o bloco a ser tratado de forma diferente aplicamos o xor com a mascara que é na mesma a soma do tweak com a chave partilhada.

In [231…

```
class Bob:
    def __init__(self):
        self.private_key=None
        self.public_key=None
        self.shared_key=None
        self.private_key_ED=None
        self.public_key_ED=None


    #gera.-se a chave privada
    def generate_key(self):
        self.private_key = X448PrivateKey.generate()

    def generate_public_key(self):
        self.public_key=self.private_key.public_key()

    def generate_key_Ed(self):
        self.private_key_ED=Ed448PrivateKey.generate()

    def generate_public_key_ED(self,public_key):
        self.public_key_ED=public_key


    #gera-se a chave partilhada entre ambos
    def generate_derived_key(self,alice_public_key):
        derived_key=self.private_key.exchange(alice_public_key)
        self.shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=64,
            salt=None,
            info=b'handshake data',
).derive(derived_key)



    #utiliza-se o HMAC para autenticar a mensagem recebida
    def verify(self,signature):
        h=hmac.HMAC(self.shared_key,hashes.SHA256())
        h.update(self.shared_key)
        try:
            h.verify(signature)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False




    def decrypt1 (self,ciphertext):
        pt=b''
        for blocks in range (0,len(ciphertext),32):
            first_tweak= ciphertext[blocks:blocks+16]
            first_block=ciphertext[blocks+16:blocks+32]
            nonce,contador,tag=parser_tweak(first_tweak)
            if tag != 0:

                msg_sem_mascara=b''
                for index, byte in enumerate(first_block):
                   mascara = self.shared_key + first_tweak
                   msg_sem_mascara += bytes([byte ^ mascara[0:16][0]])
                pt= pt + msg_sem_mascara
            else:
                cipher=Cipher(algorithms.AES(self.shared_key),modes.XTS(
```

```
        if (len(unpadded_text.decode("utf-8")) == contador):
            print("Esta correto")
            return unpadded_text.decode("utf-8")
        else:
            print("Esta incorreto")




    def verify_ED(self,signature,message) :
       return self.public_key_ED.verify(signature, message)

 #realiza-se o parsing da mensagem recebida,
 #verifica-se se a signature corresponde a signature que o bob tem
 #realiza-se a descodificação da mensagem
    def recieve_message(self,cpr):
        (signatureEd,signature,ct)=parser(cpr)
        autenticacao=self.verify(signature)
        autenticacaoED=self.verify_ED(signatureEd,signature)
        if autenticacao ==False:
            print("erro")
        else:
            if autenticacaoED ==False:
                print("erro")
            else:
                text_plain=self.decrypt1(ct)
                print(text_plain)
```

In [232…

```
#parser da mensagem enviada pela alice ao bob
def parser(msg):
    msg_splitted = msg.split(sep=b"\r\n\r\n")
    signatureEd=msg_splitted[0]
    signature=msg_splitted[1]
    ct = msg_splitted[2]
    return (signatureEd,signature,ct)

separador = b"\r\n\r\n"

def parser_tweak(tweak):
        nonce=tweak[0:8]
        contador=int.from_bytes(tweak[8:15], byteorder = 'big')
        tag=tweak[15]
        return nonce,contador,tag
```

In [233…

```
def generate_public_key():
        return X448PrivateKey.generate().public_key()
```

A comunicação é realizada na main onde se simula a geração e troca de chaves entre os intervinientes.

Confirma-se que as chaves estão iguais e procede-se a comunicação.

In [235…

```
def main() :

    #gera-se os parametros para a criação da chave privada
    alice = Alice()
    bob =Bob()
    #gera-se a chave privada para a alice e o bob
    start=time.time()
    alice.generate_key()
    alice.generate_public_key()
    alice.generate_key_Ed()
    alice.generate_public_key_ED()




    bob.generate_key()
    bob.generate_public_key()
    bob.generate_public_key_ED(alice.public_key_ED)
    bob.generate_key_Ed()



    #utiliza-se a chave publica para gerar a chave partilhada entre o bob
    alice.generate_derived_key(bob.public_key)
    bob.generate_derived_key(alice.public_key)

    #envio de mensagem
    msg=b"Ola Daniel E Joao a estudar"
    #se a chave coincidir procede-se ao envio e receção da mensagem auten




    if(alice.shared_key==bob.shared_key) == True:
        ct=alice.send_message(msg)
        bob.recieve_message(ct)
    else:
        "erro"
        print("men")


    end=time.time()
    #print(end-start)



if __name__== "__main__":
    main()
```

```
Esta correto
Ola Daniel E Joao a estudar
```