Exercício 2 do TP2

In [2]:
```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.asymmetric import x448 , ed448
from cryptography.hazmat.primitives import serialization, hashes
import os
```

_encrypt*chacha20*

Esta função recebe uma chave (`key`) e um texto simples (`plaintext`) como entrada. Ela gera um nonce aleatório de 16 bytes, calcula os tweaks para cada bloco de 64 bytes, e em seguida, cifra o texto usando o algoritmo ChaCha20. O resultado final é o texto cifrado, que inclui o nonce, o comprimento do último bloco e os blocos cifrados.

_decrypt*chacha20*

Esta função recebe a chave (`key`), o nonce e o texto cifrado (`ciphertext`). Ela extrai o comprimento do último bloco e os blocos cifrados do texto. Em seguida, calcula os tweaks, decifra cada bloco usando o ChaCha20 e concatena os blocos decifrados para obter o texto simples original.

In [3]:
```
def encrypt_chacha20(key, plaintext):
    block_size = 64  # Tamanho do bloco
    # Generate a random 16-bit nonce
    nonce = os.urandom(16)

    # Gera tweaks para cada bloco
    tweaks = [nonce + i.to_bytes(16, 'big') for i in range((len(plaintext) /

    block_size = 64  # Tamanho do bloco

    ciphertext = b''

    for i in range(0, len(tweaks)-1):
        cipher = Cipher(algorithms.ChaCha20(bytes(x ^ y for x, y in zip(twea
        encryptor = cipher.encryptor()
        ciphertext += encryptor.update(plaintext[i * block_size: (i + 1) * b

    cipher = Cipher(algorithms.ChaCha20(bytes(x ^ y for x, y in zip(tweaks[i
    encryptor = cipher.encryptor()
    ciphertext += encryptor.update(plaintext[(i + 1) * block_size:])

    ciphertext = nonce + (len(plaintext)% block_size).to_bytes(8, 'big') + c

    # Return the nonce and ciphertext
    return ciphertext

def decrypt_chacha20(key, nonce, ciphertext):
    block_size = 64  # Tamanho do bloco
    last_block_len = int.from_bytes(ciphertext[:8], byteorder='big')
    ciphertext = ciphertext[8:]
```

```
    tweaks = [nonce + i.to_bytes(16, 'big') for i in range((len(ciphertext)

    plaintext = b''

    for i in range(0, len(tweaks)-1):
        cipher = Cipher(algorithms.ChaCha20(bytes(x ^ y for x, y in zip(twea
        decryptor = cipher.decryptor()
        plaintext += decryptor.update(ciphertext[i * block_size: (i + 1) * b

    cipher = Cipher(algorithms.ChaCha20(bytes(x ^ y for x, y in zip(tweaks[i
    decryptor = cipher.decryptor()
    plaintext += decryptor.update(ciphertext[(i + 1) *block_size:])

    return plaintext
```

X448 key exchange e Ed448 Signing&Verification

Usamos X448 para o acordo de chave, o código gera chaves privadas e públicas para Alice e Bob. Em seguida, eles trocam suas chaves públicas e usam o algoritmo X448 para derivar um segredo compartilhado. Este segredo compartilhado é utilizado como base para derivar a chave para o ChaCha20.

Posteriormente, o código gera chaves Ed448 para Alice e Bob. Alice assina uma mensagem com sua chave privada Ed448 e Bob verifica a assinatura utilizando a chave pública correspondente de Alice. Se a verificação for bem-sucedida, a mensagem é considerada autenticada.

_derive*key*

Esta função utiliza o esquema de derivador de chave PBKDF2-HMAC-SHA256 para gerar uma chave derivada de 32 bytes, apropriada para uso com o ChaCha20.

In [4]:
```
def derive_key(password, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        salt=salt,
        iterations=100000,
        length=32,
        backend=default_backend()
    )
    return kdf.derive(password)


# Geração das chaves para Alice e Bob usando X448
alice_private_key_X448 = x448.X448PrivateKey.generate()
alice_public_key_X448 = alice_private_key_X448.public_key()

bob_private_key_X448 = x448.X448PrivateKey.generate()
bob_public_key_X448 = bob_private_key_X448.public_key()
# Acordo da chave usando X448
shared_secret_alice = alice_private_key_X448.exchange(bob_public_key_X448)
shared_secret_bob = bob_private_key_X448.exchange(alice_public_key_X448)
# Derivação da chave para ChaCha20 usando o segredo compartilhado
password = shared_secret_alice + shared_secret_bob
salt = os.urandom(16)
key = derive_key(password, salt) # 32 bytes
```

```
# Geração de chaves Ed448 para Alice e Bob usando a chave derivada

alice_private_key_ed448 = ed448.Ed448PrivateKey.generate()
alice_public_key_ed448 = alice_private_key_ed448.public_key()

bob_private_key_ed448 = ed448.Ed448PrivateKey.generate()
bob_public_key_ed448 = bob_private_key_ed448.public_key()

# Assinatura da mensagem por Alice
alice_signature = alice_private_key_ed448.sign(b"message")

# Verificação da assinatura por Bob usando a chave pública de Alice
try:
    alice_public_key_ed448.verify(alice_signature, b"message")
    print("Verificação da assinatura realizada e bem sucedida")
except Exception as e:
    # Handle any other unexpected exceptions
    print("An unexpected error occurred:", e)


# Example usage

plaintext = b"Hello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm kjbfwkulq

# Encrypt
encrypted_data = encrypt_chacha20(key, plaintext)


# Decrypt
nonce = encrypted_data[:16]
ciphertext = encrypted_data[16:]

decrypted_text = decrypt_chacha20(key, nonce, ciphertext)
print("Plaintext:", plaintext)
print("\nDecrypted:", decrypted_text)
```

```
Verificação da assinatura realizada e bem sucedida
Plaintext: b'Hello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm kjbfwkulq
ulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm
kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjwhdionjnjoq
enqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjw
hdionjnjoqenqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encrypt
ion!kjbwjwhdionjnjoqenqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbkqyvgcoefgpaw
iugiuoqwegrioug\nHello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm kjbfw
kulqulehbfqcwlku wlkguqlcrbkqyvgcoefgpawiugiuoqwegrioug'

Decrypted: b'Hello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm kjbfwkulq
ulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm
kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjwhdionjnjoq
enqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encryption!kjbwjw
hdionjnjoqenqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbHello, ChaCha20 encrypt
ion!kjbwjwhdionjnjoqenqin  nm kjbfwkulqulehbfqcwlku wlkguqlcrbkqyvgcoefgpaw
iugiuoqwegrioug\nHello, ChaCha20 encryption!kjbwjwhdionjnjoqenqin  nm kjbfw
kulqulehbfqcwlku wlkguqlcrbkqyvgcoefgpawiugiuoqwegrioug'
```