Exercício 2 - Trabalho Prático 1

Grupo 6:

Ruben Silva - pg57900

Luís Costa - pg55970

Problema:

1. Use o "package" cryptography para
   A. 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-128.
   B. Use esta cifra para construir um canal privado de informação assíncrona com acordo de chaves feito com `X25519 key exchange` e `Ed25519 Signing&Verification` para autenticação dos agentes. Deve incluir a confirmação da chave acordada.

Implementação do Problema

Parte I

Import

1. Instalar/importar as funcionalidades necessárias do `crypography`
2. Instalar/importar o `asyncio` para ser possível a criação do cliente-servidor assíncrono
3. Importar librarias internas do sistema como `sys` e `os`

In [72]:
```
%pip install cryptography asyncio
import os
import asyncio
import sys

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import x25519, ed25519
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
```

```
Requirement already satisfied: cryptography in c:\users\ruben\desktop\minho\mei\c
si\ec\.venv\lib\site-packages (44.0.1)
Requirement already satisfied: asyncio in c:\users\ruben\desktop\minho\mei\csi\ec
\.venv\lib\site-packages (3.4.3)
Requirement already satisfied: cffi>=1.12 in c:\users\ruben\desktop\minho\mei\csi
\ec\.venv\lib\site-packages (from cryptography) (1.17.1)
Requirement already satisfied: pycparser in c:\users\ruben\desktop\minho\mei\csi
\ec\.venv\lib\site-packages (from cffi>=1.12->cryptography) (2.22)
Note: you may need to restart the kernel to use updated packages.
```

Inicialmente é necessário criar uma variável global pra garantir que os nounces gerados são únicos e não se repetem

In [73]:
```
nounce_list = []
```

Funções Importantes Part I

1. É definido a função **xor_bytes** com return da cifragem

Esta função gera um keystream aplicando `XOR` entre os parametros. Isto é de extrema relevância pois permite facilmente reverter a cifragem

Fonte - XOR Keystream

In [74]:
```
def xor_bytes(a, b):
    return bytes(x ^ y for x, y in zip(a, b))
```

1. É definido a função "shake256XOF" com return do hash final
   A. Verificação se a strig passada está em bytes
   B. Executar o código que implementa o SHAKE256XOF

Esta função iniciliza o modelo sponge, sendo seguido do `absorve` e do `squeeze`, o que implica que é `XOF`

Fonte - Documentação de "cryptography"

In [75]:
```
def shake256XOF(text , length=32):
    if isinstance(text, str):
        text = text.encode('utf-8')
    elif not isinstance(text, bytes):
        raise TypeError("Input must be string or bytes")


    digest = hashes.Hash(hashes.SHAKE256(length),  backend=default_backend()) #s
    digest.update(text) #Absorve
    return digest.finalize() #Squeeze
```

1. É definido a função **contruct_tweak** que retorna o `tweak` (chave de curta direção)
   A. Obter um `nounce` **único** evita ataques de repitação para os mesmos *inputs* dando origem a um keystream **único**

   B. A função é colocada em big-endian concactenando o `nounce` no inicio e o `tweak` no final tendo em conta se qual o propósito desta última, `\x01` para autentição e `\x00` para cifragem. Neste caso, o `\x01` é colocado no último chunk de `16` bytes e o `\x00` ao longo da construção do `tweak`

Fonte - Documentação de "cryptography"

In [76]:
```
def construct_tweak(nounce, index):
    b_half = len(nounce)
    tweak = nounce + index.to_bytes(b_half, 'big') + b'\x00'
    return tweak
```

1. É definido a função **tweakable_encrypt** que retorna o *ciphertext* e a *tag*
   A. A usa `AES` no modo ECB como primitiva básica, o que está alinhado com a construção de *Tweakable Block Ciphers*.
   B. O *plaintext* é dividido em blocos de `16 bytes` (tamanho padrão do AES).
   C. A função *construct_tweak* gera um tweak para cada bloco com base no nounce e no índice do bloco
   D. Cada bloco é `cifrado pela primeira vez` com AES. Em seguida, o resultado é `XORed` com o tweak, garantindo que cada bloco tenha uma transformação única.
   E. O valor modificado pelo tweak é `cifrado novamente` com AES, garantindo que mesmo se um bloco se repetir, seu resultado seja diferente devido à introdução do tweak
   F. Após cifrar todos os blocos, uma `tag` de autenticação é gerada usando *SHAKE-256* no modo *XOF*. A escolha desta metodologia foi um ato experimental nosso para tentar convergir os dois exercícios e explorando as capacidades e conhecimentos lecionados, não saindo do escopo do exercício.

Com esta função, conseguimos a finalização da implementação `cifra AEAD baseada em Tweakable Block Ciphers`

Fonte - Documentação de "cryptography"

In [77]:
```
def tweakable_encrypt(plaintext, key, nounce):
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())

    ciphertext_blocks = []
    for i in range(0, len(plaintext), 16):
        block = plaintext[i:i+16].ljust(16, b'\x00')
        tweak = construct_tweak(nounce, i // 16)

        encryptor = cipher.encryptor()
        first_pass = encryptor.update(block) + encryptor.finalize()
        xored = xor_bytes(first_pass, tweak)

        encryptor = cipher.encryptor()
        ciphertext_blocks.append(encryptor.update(xored) + encryptor.finalize())

    ciphertext = b''.join(ciphertext_blocks)
```

```
    tagInput = key + nounce + ciphertext +  int(1).to_bytes(1, 'big')

    tag = shake256XOF(tagInput, len(tagInput))

    return ciphertext, tag
```

1. É definido a função **tweakable_decrypt** que retorna o `tweak` o *plaintext* final
   A. `TAG` - É aplicado *shake256XOF* com o a concatenação entre *key*, *nounce*, *ciphertext* e *\x01* para garantir que é uma cifra `AEAD` e assim evitar ataques de modificação obtendo assim conhecimento se existir uma `violação da integridade` da mensagem.
   B. Verificar se efetivamente a mensagem sofreu `violação de integrdidade`
   C. Se não sofreu, é necessário voltar a fazer o processo `reverso` do **tweakable_decrypt** voltando assim ao *plaintext* original

Com esta função, conseguimos a finalização da implementação desincriptação `cifra AEAD baseada em Tweakable Block Ciphers`

Fonte - Documentação de "cryptography"

In [78]:
```
def tweakable_decrypt(ciphertext, key, nounce, tag):
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())

    tagInput = key + nounce + ciphertext +  int(1).to_bytes(1, 'big')

    exceptedTag = shake256XOF(tagInput, len(tagInput))

    if exceptedTag != tag:
        raise ValueError("Authentication Failed!")

    plaintext_blocks = []
    for i in range(0, len(ciphertext), 16):
        block = ciphertext[i:i+16]
        tweak = construct_tweak(nounce, i // 16)

        decryptor = cipher.decryptor()
        first_pass = decryptor.update(block) + decryptor.finalize()
        xored = xor_bytes(first_pass, tweak)

        decryptor = cipher.decryptor()
        plaintext_blocks.append(decryptor.update(xored) + decryptor.finalize())

    plaintext = b''.join(plaintext_blocks).rstrip(b'\x00')
    return plaintext
```

Run Part I

As funções previamente criadas irão ser `sequencialmente` executadas de modo a validar & testar a nossa implementação desta parte do exercício

In [79]:
```
global nounce_list

key = os.urandom(16)
nounce = os.urandom(8)
```

```
while nounce in nounce_list:  #Garantir unicidade do nounce
    nounce = os.urandom(8)

plaintext = b"Hello, World! Cipher is cooL!"

ciphertext, tag = tweakable_encrypt(plaintext, key, nounce)
decrypted = tweakable_decrypt(ciphertext, key, nounce, tag)

print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext.hex())
print("Decrypted:", decrypted)
```

```
Plaintext: b'Hello, World! Cipher is cooL!'
Ciphertext: 72a145eb94b7b2519d05f572dbce2d071a6e5098b86eba4dac95dd857398ce49
Decrypted: b'Hello, World! Cipher is cooL!'
```

Parte II

Funções Importantes Parte II

A função assíncrona **generate_keys** gera dois pares de chaves criptográficas:

1. `Ed25519` (Assinatura Digital)
    A. *priv_ed_key*: Chave privada para assinar mensagens.
    B. *pub_ed_key*: Chave pública correspondente para verificar assinaturas.
2. `X25519` (Troca de Chaves - Key Exchange)
    A. *priv_x_key*: Chave privada usada para derivar uma chave secreta partilhada.
    B. *pub_x_key*: Chave pública correspondente para troca de chaves.

A função retorna todas essas chaves para serem usadas em operações de **autenticação** e **estabelecimento de chave segura**.

In [80]:
```
async def generate_keys():
    priv_ed_key = ed25519.Ed25519PrivateKey.generate()
    pub_ed_key = priv_ed_key.public_key()

    priv_x_key = x25519.X25519PrivateKey.generate()
    pub_x_key = priv_x_key.public_key()

    return priv_ed_key, pub_ed_key, priv_x_key, pub_x_key
```

A função **share_keys** envia chaves e suas assinaturas através de uma *queue* assíncrona, permitindo que outro agente receba e verifique a autenticidade das chaves. O processo utiliza os seguintes algoritmos criptográficos:

1. `Envio da Chave Pública Ed25519` (Assinatura Digital)
    A. A chave pública *Ed25519* é colocada na fila.
    B. Em seguida, a função gera uma *assinatura digital* dessa chave usando a *chave privada Ed25519*.
    C. A chave pública e a assinatura são enviadas na fila.

2. `Envio da Chave Pública X25519` (Troca de Chaves)

    A. A chave pública *X25519* é colocada na fila.
    B. Para garantir sua autenticidade, a função assina essa chave usando a *chave privada Ed25519*.
    C. A chave pública e a assinatura são enviadas na fila.

In [81]:
```
async def share_keys(queue, priv_ed_key, pub_ed_key, pub_x_key):
    await queue.put(pub_ed_key)
    sig_ed = priv_ed_key.sign(pub_ed_key.public_bytes(
        encoding=serialization.Encoding.Raw,
        format=serialization.PublicFormat.Raw
    ))
    await queue.put(sig_ed)

    await queue.put(pub_x_key)
    sig_x = priv_ed_key.sign(pub_x_key.public_bytes(
        encoding=serialization.Encoding.Raw,
        format=serialization.PublicFormat.Raw
    ))
    await queue.put(sig_x)
```

A função **receive_keys** recebe e verifica chaves e suas assinaturas a partir de uma fila assíncrona e, em seguida, realiza o estabelecimento de uma chave compartilhada. O processo utiliza os seguintes algoritmos:

1. Usa Ed25519 para verificar se a chave pública recebida foi realmente assinada pelo remetente, caso esta falha, a execução lançará uma exceção, prevenindo assim ataques de chave falsa.

2. Recebe e Verifica a Chave Pública X25519 autêntica, usando a chave Ed25519 do remetente, impedindo assim que um atacante envie uma chave falsa para manipular a troca de chaves.

3. Estabelece a Chave Compartilhada

    A. Utiliza a chave privada X25519 local (`priv_x_key`) e a chave pública X25519 do par para realizar a troca de chaves (ECDH), gerando um **segredo compartilhado**.
    B. Esse segredo é processado com o algoritmo **HKDF (HMAC-based Key Derivation Function)**, usando SHA256, para derivar uma chave simétrica de 16 bytes.
    C. Essa chave derivada (`agreed_key`) será utilizada para a comunicação cifrada via AEAD.

In [82]:
```
async def receive_keys(queue, priv_x_key):
    peer_pub_ed_key = await queue.get()
    peer_sig_ed = await queue.get()
    peer_pub_ed_key.verify(
        peer_sig_ed,
        peer_pub_ed_key.public_bytes(
            encoding=serialization.Encoding.Raw,
            format=serialization.PublicFormat.Raw
```

```
        )
    )
    peer_verify_key = peer_pub_ed_key

    peer_pub_x_key = await queue.get()
    peer_sig_x = await queue.get()
    peer_verify_key.verify(
        peer_sig_x,
        peer_pub_x_key.public_bytes(
            encoding=serialization.Encoding.Raw,
            format=serialization.PublicFormat.Raw
        )
    )

    shared_secret = priv_x_key.exchange(peer_pub_x_key)
    agreed_key = HKDF(
        algorithm=hashes.SHA256(),
        length=16,
        salt=None,
        info=b'key agreement',
    ).derive(shared_secret)

    return peer_verify_key, agreed_key
```

A função **send_message** envia uma mensagem criptografada junto com assinaturas que garantem a integridade e autenticidade dos componentes da mensagem. O processo é o seguinte:

1. Geração do Nonce:
    A. Um valor aleatório de 8 bytes (`nounce`) é gerado com `os.urandom(8)` .
    B. Esse nonce é usado para garantir que a criptografia seja única para cada mensagem.
2. Criptografia com Tweakable Encryption:
    A. A função `tweakable_encrypt` é chamada com o `plaintext` , a chave acordada (`agreed_key`) e o `nounce` .
    B. Essa função retorna o `ciphertext` (texto cifrado) e um `tag` de autenticação, que serve para verificar a integridade da mensagem.
3. Assinatura e Envio dos Dados:
    A. O `ciphertext` é assinado com a chave privada Ed25519 (`priv_ed_key`) para garantir que ele não foi alterado.
    B. Em seguida, a assinatura do `ciphertext` é enviada pela fila.
    C. O `ciphertext` propriamente dito é colocado na fila.
    D. Da mesma forma, o `nounce` é assinado e sua assinatura é enviada.
    E. O próprio `nounce` é enviado logo após sua assinatura.
    F. Por fim, o `tag` de autenticação é enviado, permitindo ao receptor confirmar que o `ciphertext` não foi modificado.

In [83]:
```
async def send_message(queue, plaintext, agreed_key, priv_ed_key):
    nounce = os.urandom(8)
    ciphertext, tag = tweakable_encrypt(plaintext, agreed_key, nounce)
    print(f"Encrypted: {ciphertext.hex()}")
```

```
    await queue.put(priv_ed_key.sign(ciphertext))
    await queue.put(ciphertext)
    await queue.put(priv_ed_key.sign(nounce))
    await queue.put(nounce)
    await queue.put(tag)
```

A função **receive_message** recebe uma mensagem criptografada e verifica sua autenticidade antes de decifrá-la. O processo segue estes passos:

1. A assinatura do *ciphertext* (sig_ct) é recebida da fila.
2. O *ciphertext* propriamente dito é recebido.
3. O *peer_verify_key* (chave pública do remetente) é usado para verificar a assinatura. Isso garante que o ciphertext não foi alterado e foi realmente enviado pelo remetente legítimo.
4. A assinatura do *nounce* (sig_nounce) é recebida da fila.
5. O *nounce* é recebido.
6. O *peer_verify_key* é usado para verificar a assinatura do nounce. Isso evita ataques de repetição e garante que o nonce foi realmente enviado pelo remetente legítimo.
7. A tag de autenticação (*tag*) gerada durante a cifragem é recebida da fila. Essa tag permite verificar se a mensagem foi alterada durante a transmissão.

In [84]:
```
async def receive_message(queue, peer_verify_key, agreed_key):
    sig_ct = await queue.get()
    ciphertext = await queue.get()
    peer_verify_key.verify(sig_ct, ciphertext)

    sig_nounce = await queue.get()
    nounce = await queue.get()
    peer_verify_key.verify(sig_nounce, nounce)
    tag = await queue.get()

    plaintext = tweakable_decrypt(ciphertext, agreed_key, nounce, tag)
    print(f"Decrypted: {plaintext}")
```

Run Parte II

In [85]:
```
async def main():
    queue = asyncio.Queue()

    sender_priv_ed, sender_pub_ed, sender_priv_x, sender_pub_x = await generate_
    receiver_priv_ed, receiver_pub_ed, receiver_priv_x, receiver_pub_x = await g

    await share_keys(queue, sender_priv_ed, sender_pub_ed, sender_pub_x)
    receiver_verify_key, receiver_agreed_key = await receive_keys(queue, receive

    await share_keys(queue, receiver_priv_ed, receiver_pub_ed, receiver_pub_x)
    _, sender_agreed_key = await receive_keys(queue, sender_priv_x)
```

```
    if sender_agreed_key == receiver_agreed_key:
        print("Key exchange successful!")
    else:
        sys.exit("Key agreement failed!")

    message = b"Criptografia e Seguranca da Internet :D"
    print(f"Message: {message}")
    await send_message(queue, message, sender_agreed_key, sender_priv_ed)

    await receive_message(queue, receiver_verify_key, receiver_agreed_key)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    if loop.is_running():
        asyncio.create_task(main())
    else:
        loop.run_until_complete(main())
```

```
Key exchange successful!
Message: b'Criptografia e Seguranca da Internet :D'
Encrypted: fd081fedf2196aa125c1ba7221a370dc0416940a43985fb814c5a4be7d8e64ac6d275d
5bd121a92159e911b05944611f
Decrypted: b'Criptografia e Seguranca da Internet :D'
```