Exercício 2 - AEAD TPBC

February 27, 2024

1 TP1

1.1 Estruturas Criptográficas - Criptografia e Segurança da Informação

Grupo 03

(PG54177) Ricardo Alves Oliveira

(PG54236) Simão Oliveira Alvim Barroso

1.1.1 Exercício 2

2. Use 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.

Instalar packages necessários

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

Resumo Este 1º trabalho prático da Unidade Curricular de Estruturas Criptográficas tem como objetivo a implementação de uma AEAD com "Tweakable Block Ciphers" e a construção de um canal privado de informação assíncrona com acordo de chaves feito com "X448 key exchange" e "Ed448 Signing&Verification" para autenticação dos agentes, incluindo uma fase de confirmação da chave acordada.

A ordem deste notebook será a seguinte:

Numa primeira secção é demonstrado o código desenvolvido bem como a sua explicação, divididos em secções. De seguida é implementada uma série de testes ao que foi desenvolvido.

Imports Sobre os imports é necessário mencionar o seguinte :

- Tal como imposto no enunciado, foi utilizado o package "cryptography" para a implementação de todas as funcionalidades pedidas.
- Utilização do asyncio para simular a comunicação entre duas entidades.

- Biblioteca OS e utilização do os.urandom para gerar valores aleatórios, uma vez que é uma melhor forma para gerar números aleatórios do que a função random do python.
- Import do datetime para a obtenção do tempo atual, para a geração da `associated data`

[ ]:
```
import os
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from datetime import datetime
import sys
import asyncio
import nest_asyncio

nest_asyncio.apply()
```

Modo TAE ("tweaked authentication encryption") Para a resolução deste trabalho, optamos por duas abordagens diferentes, uma para o AES256 e outra para o ChaCha20. Estas abordagens têm também diferença se implementam o modo TAE ou não (assim como descrito na última secção do texto Capítulo 1: Primitivas Criptográficas Básicas).

Uma cifra AEAD com "Tweakable Block Ciphers" utiliza um input adicional designado de *tweak*. Estes funcionam como chaves únicas de cada bloco, enquanto que a chave propriamente dita é a mesma em todos os blocos, tornando a cifra mais resistente a ataques.

Optamos por utilizar o Chacha20Poly1305 com o modo TAE e o AES256 sem o modo TAE.

Esta opção é ativada através de um parâmetro boleano que é passado à classe `channel`. Se obter o valor de `True` então o modo TAE está ativado e é utilizado o **Chacha20Poly1305**, caso contrário é utilizado o **AES256**.

Relativamente ao AES256 utilizamos com o modo CTR uma vez que este não precisa de padding. Uma vez que AES256 no modo CTR é uma cifra por blocos, utilizamos o aprendido nas aulas teóricas (como construir uma TPBC $\tilde{E}(w,k,x)$ a partir de uma PBC $E(k,x)$) para implementar o pedido.

Relativamente ao modo TAE é especificada a divisão dos blocos e todo o seu tratamento mais à frente, sendo o algoritmo utilizado apenas para cifrar e decifrar, não tendo o própio algoritmo de dividir a mensagem como é o caso do AES256.

[ ]:
```
mode_TAE = True
```

Funções auxiliares sem modo TAE Estas funções servem de auxiliares quando o **modo TAE** toma o valor de **falso**. Neste caso e de modo a apoiar a expressão $ Ẽ(w,k,x) = E(k,w E(k,x)) $,

é utilizado uma função de padding. Esta função primeiramente aumenta o tamanho do tweak para o tamanho da mensagem, preenchendo o tweak de bits de valor 0 e de seguida chama a função `xor` para fazer entre cada bit da primeira cifra e do tweak.

[ ]:
```
def padding(b1,b2):
    lb1 = len(b1)
    lb2 = len(b2)
    if lb1 < lb2:
        b1 += b"\x00" * (lb2 - lb1)
    return xor(b1, b2)

def xor(b1, b2):
    result = b''
    result += bytes([bt1 ^ bt2 for bt1, bt2 in zip(b1,b2) ])
    return result
```

Funções auxiliares com modo TAE De modo a aplicar o modo TAE, é preciso primeiro uma função auxiliar que gera os tweaks a aplicar, assim como a figura demonstrada nas aulas.

Esta função recebe como input o número de blocos, tamanho da mensagem, e um "name only used once" (nonce). Esta função vai gerar os vários tweaks gerados por esta função serão utilizados no mecanismo geral da cifra que vai ser explicado mais à frente (função encrypt_TAE e decrypt_TAE). O n-1 primeiros blocos, onde n é o número total de blocos, são utilizados para a cifra dos blocos da mensagem e último é utilizado para a autenticação do criptograma (para gerar o tag).

Decidimos que cada bloco vai ter 32 bytes, sendo igual ao tamanhos dos blocos das mensagens.

Tal como descrito na imagem acima mencionada nos n-1 primeiros blocos são gerados da seguinte maneira:

- A primeira metade é ocupada pelo nounce (16 bytes).
- A segunda metade é ocupada pelo contador (número do bloco que está a ser cifrado)
- Por último, um bit a 0.

$w_i = [nonce|i|0]$, i = 0.. n-1

Já o último bloco é gerado de forma diferente:

- A primeira metade é ocupada pelo nounce (16 bytes).
- O comprimento da mensagem (sem padding)
- Um bit a 1

$w^* = [nonce|length(plaintext)|1]$

[ ]:
```
def gen_tweaks(number_blocks, plaintext_length, nonce):

    # criação do array que vai conter os n-1 tweaks
    ctweaks = []
    #  geração dos i..n-1 tweaks : [nonce|counter|0]
    for i in range(0, number_blocks):
```

```
        # um tweak é composto em metade por um nounce, e a outra metade por um␣
↪counter
        tweak = nonce + int(i).to_bytes(16, byteorder='big')
        # o último byte do tweak é 1
        tweak = tweak[:-1] + int(1).to_bytes(1, byteorder='big')
        ctweaks.append(tweak)

    # o tweak w* : [nonce |plaintext_length |0]
    # é composto pela metade com um nounce, e a outra metade com o tamanho do␣
↪plaintext
    authtweak = nonce + plaintext_length.to_bytes(16, byteorder='big')
    # o último byte do tweak é 0
    authtweak = authtweak[:-1] + int(0).to_bytes(1, byteorder='big')

    return ctweaks, authtweak
```

Esta função é utilizada para cifrar cada bloco da mensagem. A mensagem é cifrada de acordo com a abordagem acima descrita e ensinada na aula: $ Ẽ(w,k,x) = E(k,w E(k,x)) $.

Esta função devolve Ẽ(w,k,x). Utilizamos o algoritmo `ChaCha20Poly1305` para cifrar.

[ ]:
```
def cipher(ckey, nonce, tweak, plaintext, ad):

    chacha = ChaCha20Poly1305(ckey)
    ciphertext = chacha.encrypt(nonce, plaintext, ad)

    xored = b''

    for (a,b) in zip(tweak, ciphertext):
        xored += bytes([a^b])

    return xored
```

Esta função funciona de maneira análoga à função de cifrar descrita posteriormente.

Recebe-se como argumentos para a função o criptograma, a chave , os dados associados, o tag e o nounce utilizado para gerar o tweak, .

Primeiramente, dividimos o criptograma em blocos de 32 bytes. Vamos depois gerar os tweaks com o número de blocos, o tamanho da mensagem (plaintext) e o nounce.

De seguida vamos decifrar cada bloco. Neste ao contrário do que é preciso fazer na função de cifrar não precisamos de fazer padding em gerar Cm (representado na figura), uma vez que assumimos que o texto já vem em blocos todos com 32 bytes cada um.

Assim passamos para a última fase que é a geração do Tag.

Esta função segue o descrito no diagrama das aulas.

[ ]:
```
def decrypt_TAE(ciphertext, ckey, ad, tag, nonce, ntweak):
    # dividir o ciphertext em blocos de 32 bytes
    blocks = []
    for i in range(0, len(ciphertext), 32):
        blocks.append(ciphertext[i:i+32])

    # vamos obter dados dos blocos
    number_blocks = len(blocks)
    n = len(ciphertext)
    r = len(tag)

    # como está dividido em blocos de 32 bytes, o tamanho do plaintext é n - (32 - r)
    length_plaintext = n - (32 - r)

    # gerar os tweaks para depois aplicar
    ctweaks, authtweak = gen_tweaks(number_blocks, length_plaintext, ntweak)

    # criamos uma lista para colocar os blocos decifrados
    blocks_decipher = []

    # Para cada bloco vamos decifrar / one time pad
    for w in range(0, number_blocks):
        plaintext = cipher(ckey, nonce, ctweaks[w], blocks[w], ad) # pode ser paralelizado numa futura versão
        blocks_decipher.append(plaintext)


    # aplicamos o ciclo da figura novamente para obter o auth (sumatorio do de todos os blocos )
    auth = blocks_decipher[0]
    for i in range(1, number_blocks):
        xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(auth, blocks_decipher[i])] # xor
        auth = b"".join(xored)

    auth = auth[:r]
    authtweak = authtweak[:r]


    gen_tag = cipher(ckey, nonce, authtweak, auth, ad)

    # verificar se o que foi enviado não foi alterado (se está autencidado)
    if tag == gen_tag:
        blocks_decipher[-1] = blocks_decipher[-1][:r] # retirar o padding ao último bloco
        plaintext = b"".join(blocks_decipher)
```

```
    else :
        sys.exit("Tag not valid!")

    return plaintext.decode('utf-8')
```

Por último, vem a função de cifrar a mensagem. Primeiramente, devidimos a mensagem em blocos de 32 bytes. O último bloco, caso tenha tamanho inferior a 32, é feito padding com bits a 0.

Depois de feito a divisão passamos para a geração dos tweaks. Para isso precisamos antes de gerar um nonce a ser utilizado para a geração dos tweaks anteriormente explicado.

Geramos também o nounce a utilizar com o `ChaCha20Poly1305`. Para cada um dos blocos da mensagem ciframos e juntamos a uma lista de blocos cifrados. Fazemos isto para os n-1 primeiro blocos.

Para o último bloco, passamos o tamanho do bloco para uma lista de bytes de tamanho 32, ciframos e fazemos o xor com o último bloco da mensagem (plaintext) com pad.

A última fase que temos de fazer é a de gerar o tag. Para isso primeiro temos que gerar o auth. Este é auth é gerado através do xor de cada um dos blocos da mensagem (plaintext) com o seguinte. Pegamos nisso e no authtweak gerado anterior mente e fazemos a cifragem. Ao ir buscar a tag ignoramos os bytes que foram adicionados com o padding.

É de notar que o nounce gerado para a cifra e para os tweaks é diferente, só seria o mesmo se as mensagens tivessem comprimento de 24 bytes (o ChaCha20Poly1305 aceita nounces de 12 bytes).

Este passos todos estão como demonstrados na figura presente na página da Unidade Curricular.

[ ]:
```
def encrypt_TAE(plaintext, ckey, ad):
    # Dividir a mensagem em vários blocos
    blocks = []
    for i in range(0, len(plaintext), 32):
        block = plaintext[i:i+32].encode('utf-8')
        # padding
        r = len(block) # no final vai dar o comprimento do último bloco
        if r < 32:
            blocks.append(block + ((32 - r) * b'\0'))
        else:
            blocks.append(block)

    length = len(plaintext)
    number_blocks = len(blocks)

    # gerar os tweaks
    ntweak = os.urandom(16)
    ctweaks, authtweak = gen_tweaks(number_blocks, length, ntweak)

    # blocos cifrados
    encrypted_blocks = []
```

```
    # precisamos de gerar um nounce utilizado pela primitiva chacha20
    nonce = os.urandom(12)

    # vamos cifrar os primeiros n-1 blocos (aqueles que não têm o padding)
    for w in range(0, number_blocks - 1):
        ciphertext = cipher(ckey, nonce, ctweaks[w], blocks[w], ad)
        encrypted_blocks.append(ciphertext)

    # cifrar o último bloco
    # primeiramente passamos o valor de r para bytes
    r_bytes = int(r).to_bytes(32, byteorder='big')
    # ciframos isso com o último tweak e com chave como mostra a figura
    ct = cipher(ckey, nonce, ctweaks[-1], r_bytes, ad)
    # fazemos agora xor com o último bloco do plaintext(ou seja já com padding)
    # com o resultado anterior (ct)
    xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(ct, blocks[-1])]
    # dá o último bloco cifrado
    lciphertext = b"".join(xored)
    # juntamos à lista de blocos cifrados
    encrypted_blocks.append(lciphertext)

    # Fase de obter o tag
    # o auth é o xor de todo os blocos de plaintext
    auth = blocks[0]
    for i in range(1, number_blocks):
        xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(auth,␣
↪blocks[i])]
        auth = b"".join(xored)

    # a tag é obtida então com a cifra do auth, com o authtweak
    tag = cipher(ckey, nonce, authtweak, auth, ad)[:r]

    # juntamos todos os blocos cifrados
    ciphertext = b"".join(encrypted_blocks)

    return ciphertext, nonce, ntweak, tag
```

Classe Channel De modo a responder ao segundo ponto pedido na resolução deste exercicio, foi criada uma classe **channel** para construir um canal de informação assíncrono privado com acordo de chaves utilizando a troca de chaves X448 e Ed448 para autenticação do agente, incluindo uma fase de confirmação de chaves.

A primitiva Ed448 é utilizada para verificar e assinar os pacotes trocados entre duas entidades. Cada uma das entidades, tem uma chave privada usada para assinar o que quer enviar. Do outro lado a chave pública da primitiva Ed448 verifica se a assinatura ou o pacote foram alterados. É de notar que a chave pública é gerada através da privada.

A primitiva X448 foi utilizada para cada um dos lados gerar a sua chave pública e privada; haver uma

troca de chaves públicas e através disso obter um segredo comum a ambas as partes. Este segredo é utilizado para derivar uma chave a ser utilizada para a cifragem e decifragem das mensagens. Assim a chave é conhecida apenas pelas partes que vão comunicar, não tendo necessidade de ser transmitida por um canal de comunicação.

É importante dizer que esta classe é utilizada por ambos os modos, tendo comportamentos semelhantes independentemente do valor da flag **mode TAE**.

Vamos explicar cada uma destes métodos de modo mais teórico e na secção da função `main`, será explicada a chamada de cada um dos métodos. Deste modo a classe tem 6 métodos:

- O primeiro método é utilizado para criar uma instância do objeto da classe e recebe como argumento um objeto queue, que utilizamos na resolução deste exercício. Esta queue é utilizada uma vez que decidimos para a resolução deste exercício `asyncio`. Mais à frente, a quando da inicialização desta classe.
- Em segundo está o método de gerar as chaves. Neste método e tal como o nome diz, geramos as chaves tanto da primitiva X448 como da primitiva Ed448.
- Depois de geradas as chaves, criamos um método para partilhar as chaves. Este método coloca na queue do asyncio as chaves públicas (apenas esta podem ser partilhadas). É de realçar também que depois de cada chave pomos também uma assinatura da chave.
- O terceiro método é o de receber as chaves. Este método começa primeiro por receber as chaves públicas e as suas assinaturas. Utiliza a função `verify` para verificar se ou a assinatura ou chave foi alterada durante a sua transmissão. Se foi, é lançada uma `Exception` que termina o programa. Feito isto, deriva-se um segredo através da utilização da nossa chave privada e da pública do peer (utilizador com o qual estamos a comunicar). Este segredo, e como apenas são comunicadas as chaves públicas, é apenas conhecido pelas partes que vão comunicar. No entanto, este segredo é uma sequência longa de bits. Para combater isto e para combinar a chave a utilizar no AEAD, foi utilizada um `HKDF` para derivar uma chave a utilizar. Este algoritmo usa uma função de hash também para tornar a chave derivada mais dificil de "advinhar" por um atacante.
- Nestes 3 métodos, criamos a base para esta e para o seguinte, o de enviar e receber mensagens respetivamente. A função começa pela criação de `associated data` que é a codificação em bits da hora e dia em que vai ser mandada a mensagem. De seguida e dependendo se o modo TAE está ativo ou não são mandadas mais ou menos dados. No entanto, o desenho geral é o mesmo: Primeiro uma atribuição da key para a chave combinada na função anterior; De seguida, cifragem da mensagem; Por último, colocação da mensagem e diversas informações na queue do `asyncio`.
- O método de receção da mensagem tem uma lógica semelhante ao método de envio. Há uma primeira fase de receção das várias informações da queue do `asyncio`; decifrar o criptograma de acordo com o modo TAE; Por último imprime o a mensagem decifrada.
- Em último lugar está uma função simples feita para efeitos de debug e para testes, em que imprime a chave concordada no 3º método.

É de mencionar que utilizamos para um modo TAE a verdade utilizamos a primitiva `ChaCha20Poly1305`. Por outro lado, o modo TAE falso utiliza o `AES256`.

É também de mencionar a criação de 3 testes que se encontram comentados. Para a sua execução

basta apenas que deixem de estar comentados. À frente deles está uma explicação do que cada um faz.

Decidimos em baixo comentar cada fase dos vários métodos para melhor compreensão.

[ ]:
```
class channel:
    def __init__(self, queue):
        self.queue = queue

    # função para gerar as chaves privadas e públicas das duas primitivas
    async def gen_keys(self):

        self.priv_Ed448_key = Ed448PrivateKey.generate()
        self.pub_Ed448_key = self.priv_Ed448_key.public_key() # chave pública é gerada através da privada

        # processo análogo para a primitiva X448
        self.priv_x448_key = X448PrivateKey.generate()
        self.pub_x448_key = self.priv_x448_key.public_key()

    # função para partilhar as chaves com a pessoa com quem se quer comunicar
    async def share_keys(self):

        #self.pub_Ed448_key = Ed448PrivateKey.generate().public_key() # Teste 2: Chave foi alterada
        # colocamos na queue a chave pública da primitiva
        await self.queue.put(self.pub_Ed448_key)
        # assinamos a chave pública
        sigEd448 = self.priv_Ed448_key.sign(
            self.pub_Ed448_key.public_bytes(Encoding.Raw, PublicFormat.Raw)
        )

        #sigEd448 += b":eyes:" # Teste 2 : Autenticação. Verificar uma alteração no pacote ou na assinatura.

        # colocamos a assinatura na queue
        await self.queue.put(sigEd448)

        # processo análogo para a primitiva ED448
        await self.queue.put(self.pub_x448_key)

        sigx448 = self.priv_Ed448_key.sign(
            self.pub_x448_key.public_bytes(Encoding.Raw, PublicFormat.Raw)
        )

        await self.queue.put(sigx448)

    # função que recebe as chave e trata delas
```

```
    async def receive_keys(self):

        # recebemos as key e a assinatura da queue
        peer_pub_Ed448_key = await self.queue.get()
        peer_pub_Ed448_key_signature = await self.queue.get()

        # utilizamos o método verify para verificar se ou a chave ou a␣
↪assinatura foram modificadas. Se alguma foi o programa pára a execução
        peer_pub_Ed448_key.verify(peer_pub_Ed448_key_signature,␣
↪peer_pub_Ed448_key.public_bytes(Encoding.Raw, PublicFormat.Raw))
        self.peer_verify_key = peer_pub_Ed448_key


        # recebemos a chave e a assinatura do protocolo x448
        peer_pub_x448_key = await self.queue.get()
        peer_pub_x448_key_signature = await self.queue.get()

        # utilizamos o método verify para verificar se ou a chave ou a␣
↪assinatura foram modificadas. Se alguma foi o programa pára a execução
        self.peer_verify_key.verify(peer_pub_x448_key_signature,␣
↪peer_pub_x448_key.public_bytes(Encoding.Raw, PublicFormat.Raw))


        # peer_pub_x448_key = X448PrivateKey.generate().public_key() # Teste 3 :
↪ Chave acordada diferente dos 2 lados

        # vamos derivar uma chave a partir do HKDF
        shared_secret = self.priv_x448_key.exchange(peer_pub_x448_key)
        derived_key = HKDF(
            algorithm = hashes.SHA256(),
            length = 32, # tanto o chacha20 como o AES256 usam chaves de 256␣
↪bits / 32 bytes
            salt = None,
            info = b"handshake data",
        ).derive(shared_secret)
        self.agreed_key = derived_key


    # Função que trata da cifra da mensagem e enviar
    async def send(self, plaintext):
        # criação da associated data
        ad = str(datetime.now()).encode('utf-8')

        # a chave a utilizar foi a chave derivada através do HKDF
        key = self.agreed_key
```

```
        print("Plaintext Sent: "+str(plaintext))

        # dependendo do modo temos diferentes comportamentos
        if mode_TAE:
            ciphertext, nounce, tweak, tag = encrypt_TAE(plaintext, key, ad)

        else:
            # passamos o plaintext para b""
            plaintext = plaintext.encode()
            # geramos um nounce e um tweak com recurso ao urandom
            nounce = os.urandom(16)
            tweak = os.urandom(8)
            # utilização do AES256 em modo CTR
            aes = Cipher(algorithms.AES256(key), modes.CTR(nounce)).encryptor()
            # ciframos o texto pela primeira vez
            ciphertext = aes.update(plaintext)
            # fazemos xor com o tweak do texto cifrado
            xored = padding(tweak, ciphertext)
            # voltamos a cifrar : E(k,w   E(k,x))
            ciphertext = aes.update(xored) + aes.finalize()

        print("\tCiphertext Sent: "+str(ciphertext))

        # dependendo do modo há informações que precisam ou não de ser passadas
        if mode_TAE:

            await self.queue.put(self.priv_Ed448_key.sign(tag))
            await self.queue.put(tag)

            await self.queue.put(self.priv_Ed448_key.sign(ad))
            await self.queue.put(ad)

        await self.queue.put(self.priv_Ed448_key.sign(ciphertext))
        await self.queue.put(ciphertext)


        await self.queue.put(self.priv_Ed448_key.sign(nounce))
        await self.queue.put(nounce)

        await self.queue.put(self.priv_Ed448_key.sign(tweak))
        await self.queue.put(tweak)


    # este método trata de receber as informações e decifrar o criptograma
    async def receive(self):
```

```
        # a chave a utilizar foi a chave derivada através do HKDF
        key = self.agreed_key


        # verificamos sempre o que recebemos
        # dependendo do modo podemos ou não receber informações
        if mode_TAE :
            sig_tag = await self.queue.get()
            tag = await self.queue.get()
            self.peer_verify_key.verify(sig_tag, tag)

            adsig = await self.queue.get()
            ad = await self.queue.get()
            self.peer_verify_key.verify(adsig, ad)


        sig_ctext = await self.queue.get()
        ciphertext = await self.queue.get()
        self.peer_verify_key.verify(sig_ctext, ciphertext)


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

        sig_tweak = await self.queue.get()
        tweak = await self.queue.get()
        self.peer_verify_key.verify(sig_tweak, tweak)

        print("\tCiphertext Received: "+str(ciphertext))

        # dependendo do modo há maneiras diferentes de decifrar
        if mode_TAE:
            plaintext = decrypt_TAE(ciphertext, key, ad, tag, nounce, tweak)
        else :
            aes = Cipher(algorithms.AES256(key), modes.CTR(nounce)).decryptor()
            plaintext = aes.update(ciphertext)
            xored = padding(tweak, plaintext)
            plaintext = aes.update(xored) + aes.finalize()


        print("Decrypted: "+str(plaintext)+"\n")

    # função auxiliar usada para imprimir chave
    async def print_agreed_key(self):
        print(self.agreed_key)
```

Função Main Sobre esta função

Relativamente à utilização de uma única queue pelas duas instâncias da classe **channel** foi feita de modo a simplificar a resolução deste exercicios. Funcionaria de modo análogo a utilização de 2 queues (assim como é num cenário real entre 2 utilizadores remotos), mas não era essencial para a resolução do exercício. As únicas mudanças seriam na inicialização das classes e leitura das queues.

[ ]:
```
async def main():
    # criar a queue do asyncio
    queue = asyncio.Queue()

    # criar duas entidades do canal
    emissor = channel(queue)
    receptor = channel(queue)

    # criamos as queis dos 2
    await emissor.gen_keys()
    await receptor.gen_keys()

    # envia se as keys de um para o outro
    await emissor.share_keys()
    await receptor.receive_keys()
    await receptor.share_keys()
    await emissor.receive_keys()

    # se a chave acordada for a mesma otimo, se não programa termina
    if (emissor.agreed_key == receptor.agreed_key):
        print("Chave acordada: " + str(emissor.agreed_key) + "\n")
    else :
        print(f"Chave não foi acordada\nChave emissor: {str(emissor.
 ↪agreed_key)}\nChave recetor: {str(receptor.agreed_key)}")
        sys.exit("Chave não foi acordada")

    # alguns testes de envio de mensagens
    await emissor.send("Brave Sir Robin ran away. Bravely ran away away. . .␣
 ↪Monty Python and The Holy Grail!")
    await receptor.receive()
    await receptor.send("A+!")
    await emissor.receive()

asyncio.run(main())
```

```
Chave acordada: b'\xdc:\xca\xfe@\x10\x11\xaf\xb5\xba\xe3\xe4\xb0?\x9fp\xc3\xb3\x
aa\x7f\x80\xcd\xa3Ph\xc1\xee\x14\x1eT^h'

Plaintext Sent: Brave Sir Robin ran away. Bravely ran away away. . . Monty
Python and The Holy Grail!
        Ciphertext Sent: b"\xf5\x92\xec\x1e<\x1f!\x89d\x7f\xffc!\xfa\x89\xd6\x81
```

```
\xbf\xbf\xf8\xd7\xca\xba('a\xa9 \xa5\x18S\xac\xce\xc0\xff\t7\x1f\x13\x97w&\x8dm4
\xf2\x9e\xd8\xd3\xf0\xf1\xf6\x96\xf0\xb4?}8\xcb\x02\xbd\x1a^\xaf\xd9\xc0\xec\x06
=\x1f&\x88s\x7f\xe5c/\xea\xc7\xb1\x81\xbf\xb8\xb4\x97\xbd\xdbQ\tA\xebR\xc4n6\xd5
"
        Ciphertext Received: b"\xf5\x92\xec\x1e<\x1f!\x89d\x7f\xffc!\xfa\x89\xd6
\x81\xbf\xbf\xf8\xd7\xca\xba('a\xa9 \xa5\x18S\xac\xce\xc0\xff\t7\x1f\x13\x97w&\x
8dm4\xf2\x9e\xd8\xd3\xf0\xf1\xf6\x96\xf0\xb4?}8\xcb\x02\xbd\x1a^\xaf\xd9\xc0\xec
\x06=\x1f&\x88s\x7f\xe5c/\xea\xc7\xb1\x81\xbf\xb8\xb4\x97\xbd\xdbQ\tA\xebR\xc4n6
\xd5"
Decrypted: Brave Sir Robin ran away. Bravely ran away away. . . Monty Python and
The Holy Grail!

Plaintext Sent: A+!
        Ciphertext Sent: b'\xe1(\xca\xcf~\x8d\xdf\x9c\xad\xcb45\xb5\xba\x10{\xab
j\xee\x9bg\xc8O\xb4\x18\x1a\xbeYm\xe1l\xe6'
        Ciphertext Received: b'\xe1(\xca\xcf~\x8d\xdf\x9c\xad\xcb45\xb5\xba\x10{
\xabj\xee\x9bg\xc8O\xb4\x18\x1a\xbeYm\xe1l\xe6'
Decrypted: A+!
```