

# Tp2

March 16, 2020

## 1 Introdução

O objetivo do trabalho realizado foca-se em criar duas implementações que permitam realizar sessões síncronas entre dois agentes designados Emitter e Receiver. Em que a primeira recorre a Criptografia dita “Tradicional” e a segunda a Criptografia de Curvas Elípticas.

Esta comunicação tem que decorrer de forma segura, para tal, pretende-se:

- Encriptação do texto enviado com verificação da integridade do mesmo usando HMACs.
- Acordo de chaves para uso na comunicação com verificação da chave.
- Autenticação dos Agentes

Para tal foi indicado que se usasse a cifra simétrica AES com um modo de operação seguro contra ataques aos vetores de iniciação, protocolo de acordo de Chaves Diffie-Hellman e autenticação dos agentes através do DSA para a primeira implementação. No caso, da sessão que usa Curvas elípticas pretende-se usar a cifra simétrica ChaCha20Poly1305 para encriptação, o protocolo ECDH para o acordo de chaves e o ECDSA para a autenticação.

Seguidamente, será explicado a primeira implementação, isto é, os métodos que as classes implementam, como também, um exemplo de utilização. E o mesmo será feito para a implementação usando Criptografia curvas Elípticas. Finalmente, terminar-se-á este relatório com algumas conclusões ao trabalho realizado e como ainda poderia ser melhorado.

## 2 Implementação usando Criptografia “Tradicional”

As duas implementações contêm uma classe Emitter e uma classe Receiver que junto com uma terceira, neste caso, *encAES* fornecem as propriedades, anteriormente citadas. Para explicar o código e, também, a lógica da implementação ir-se-á explicar utilizando uma vista Top-Down.

A encriptação usada

### 2.1 Classe Emitter

A classe quando é criada precisa que lhe seja passado uma classe que implemente os métodos:

- *gen\_ephemeral\_keys* - que deve gerar as chaves efémeras usadas unicamente para esta comunicação.
- *setParameters* - Estabelece os parâmetros necessários recebendo-os em formato PEM (parâmetros DH).

- `keyAgreementE(connection)` - Executa o protocolo de acordo de chaves podendo ter outros mecanismo associados assumindo o papel de Emitter(Quem inicia a comunicação)
- `messaging(connection)` - Mecanismo de comunicação com o Receiver.

O método de entrada na classe que deve ser usado é o `run()`. Este poderia ser modificado para, no caso de a ligação terminar, tentar outra vez e ate mesmo escolher a porta e o Host a qual se pretende conetar. Assim, aproveitando mais o uso das chaves estaticas e efêmeras. O método `connect(host,port)` estabelece a ligação com o Receiver.

```
In [ ]: class Emitter():
```

```
    def __init__(self, crypto):
        self.crypto = crypto

    def connect(self, host, port):
        self.crypto.gen_ephemeral_key()

        with so.connect((host, port))
            socket.socket(socket.AF_INET, socket.SOCK_STREAM) as so:
                print("Starting key Agreement")
                isAgreed = self.crypto.keyAgreementE(so)
                if isAgreed:
                    print("Messaging with encryption")
                    self.crypto.messaging(so)

    def run(self):
        self.crypto.setParameters(b'-----BEGIN DH PARAMETERS-----\nMEYCQC+nC0/Ujb2mfS
        # voltar a conetar?
        self.connect("localhost", 8001)
```

## 2.2 Classe Receiver

A classe *Receiver* é análoga a classe *Emitter* com a diferença que a primeira fica a espera da conexão, tendo assim um comportamento mais parecido a de um servidor. A diferença do Emitter, a porta e a interface qual fica a escuta fica decidida logo aquando da inicialização do objeto da classe. E, que a classe que lhe é passada tem que implementar:

- `gen_ephemeral_key` - igual ao do Emitter.
- `setParameters` - igual ao do Emitter.
- `KeyAgreementR(connection)` - Executa o acordo de chaves podendo ter outros mecanismo associados assumindo o papel de Receiver (Quem recebe a proposta de comunicação).
- `receiving(connection)` - Mecanismo de recepção de mensagens.

O método de entrada e como as considerações feitas ao mesmo são identicas a classe Emitter. O método `connect()` é o que espera pela conexão na porta e interface escolhida aquando da inicialização do objeto.

```
In [ ]: class Receiver():
```

```

def __init__(self,port,host,crypto):
    self.port = port
    self.host = host
    self.crypto = crypto
    self.connection = None

def connect(self):
    self.crypto.gen_ephemeral_key()

    with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as so:
        so.bind((self.host,self.port))
        so.listen()
        connect,address = so.accept()
        with connect:
            print("Starting key Agreement")
            isAgreed = self.crypto.keyAgreementR(connect)
            if isAgreed:
                print("Receiving with encryption")
                self.crypto.receiving(connect)

def run(self):
    self.crypto.setParameters(b'-----BEGIN DH PARAMETERS-----\nMEYCQQC+nc0/Ujb2mfS
    # voltar a conetar?
    self.connect()

```

## 2.3 Classe encAES

Na implementação realizada, ambas as classes usam esta terceira para implementar os métodos respetivos. Os métodos `messaging(connection)` e `receiving(connection)`, como pode ser visto no código a seguir apresentado, realizam um processo simples. O método do Emitter lê uma String do stdin, utiliza o algoritmo de encriptação + HMAC e envia essa mensagem. Na outra ponta da comunicação, isto é, no método do Receiver a mensagem é recebida, verificada, descriptada e finalmente imprimida no stdout do utilizador.

```

In [ ]: def messaging(self,connection):
        print("Now you can send messages")
        while True:
            data = input("----> ")
            encData = self.encryptThenMac(data)
            connection.send(encData)
            if "Exit" == data:
                break

def receiving(self,connection):
    while True:
        try:
            data = connection.recv(encAES.RCV_BYTES)

```

```

        dencData = self.decryptThenMac(data)
        print(dencData)
    except EOFError as e:
        print("bye bye")
        break

```

Os métodos `keyAgreementE()` e `keyAgreementR()` são o complementar um do outro desta forma, como exemplo mostrar-se-á o código de apenas de um deles para exemplificar como o protocolo de acordo de chaves é executado por esta classe.

O protocolo de acordo de chaves usado foi o DH, mais especificamente, a variante com duas chaves estaticas e duas efémeras tendo como base a recomendação do NIST:

- [Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography](#) - Especificamente as seções 6.1.1 e 6.1.1.1 para o acordo de chaves, e a seção 6.1.1.5.3 para a confirmação de chaves.

Ainda é neste método onde se junta o DSA para autenticação dos agentes. Primeiramente, é feita a troca das chaves estaticas entre os dois agentes, de forma a gerar o segredo partilhado derivado das chaves estaticas. O mesmo é feito com as chaves efémeras, no entanto junto com a chave vem um HMAC para confirmação da chave partilhada gerada a partir da concatenação dos segredos estaticos e efémeros depois de passar por uma função para derivação de chaves (PBKDF2HMAC). Desta forma, confirma-se que a chave gerado por ambos os lados da comunicação é a mesma. Também junto é recebida uma assinatura digital das chaves publicas envolvidas no processo para autenticação dos agentes utilizando o DSA. Finalmente para o Receiver realizar as mesmas confirmações envia-se o HMAC e a assinatura digital gerado pelo Emiiter. Caso algum dos pontos de confirmação falhe o processo termina e a conexão também.

```

In [ ]: def keyAgreementE(self, connection):

    # static
    connection.send(encodePublicKey(self.public_key))
    pk = connection.recv(encAES.RCV_BYTES)
    static_shared_secret = self.generateSharedSecret(pk, self.private_key)

    # ephemeral cryptography.hazmat.primitives.asymmetric.
    connection.send(encodePublicKey(self.e_public_key))
    e_pk_mac = connection.recv(encAES.RCV_BYTES)
    e_pk_mac_load = pickle.loads(e_pk_mac)
    e_shared_secret = self.generateSharedSecret(e_pk_mac_load["e_key"],
                                                self.e_private_key)

    # shared key
    self.generateSharedKey(static_shared_secret, e_shared_secret)

    # DSA

    sign = self.decrypt(e_pk_mac_load["signature"])
    try:
        self.verifySign(pk + encodePublicKey(self.public_key) +

```

```

        e_pk_mac_load["e_key"] +
        encodePublicKey(self.e_public_key) ,sign)
except InvalidSignature as In:
    #connection.send(pickle.dumps({"mac": "mac", "signature": "signature"}))
    print("Invalid Signature")
    return False

# test confirmation
try:
    self.verifyMac(b"KC_1_V" + encodePublicKey(self.e_public_key) +
        e_pk_mac_load["e_key"], e_pk_mac_load["mac"])
except InvalidSignature as In:
    #connection.send(pickle.dumps({"mac": "mac", "signature": "signature"}))
    print("Key Confirmation Failed")
    return False

# Send mac and sign
mac_and_sign = {"mac": self.mac(b"KC_1_U" + encodePublicKey(self.e_public_key)
    + e_pk_mac_load["e_key"]), "signature":
    self.encrypt(
        self.sign(
            encodePublicKey(self.public_key) + pk +
            encodePublicKey(self.e_public_key) +
            e_pk_mac_load["e_key"]
        )
    )
}

connection.send(pickle.dumps(mac_and_sign))

e_pk_mac = None
e_pk_mac_load
e_shared_secret = None
static_shared_secret = None
pk = None

return True

```

O método de combinação de encriptação e verificação é o Encrypt-Then-MAC. Em que a encriptação usada é o AES com CTR mode que garante segurança enquanto aos IV desde que este seja usado como o nonce, isto é, nunca seja repetido a sua utilização para uma certa chave. Porém, este modo de operação é susceptível a bit flipping attacks por este motivo tem que ser acompanhado por um mecanismo de verificação de integridade. O gerador de nonces usado na implementação foi o recomendado pela biblioteca Cryptography, ou seja, o `os.urandom()` que é usado por cada vez que é encriptada uma mensagem.

```

In [ ]: def encrypt(self, msg):
        nonce = os.urandom(16)
        # encryption

```

```

cipher = Cipher(algorithms.AES(self.shared_key[:encAES.ENCRIPTION_KEY_SIZE]),
                 modes.CTR(nonce),
                 backend=self.backend)
enc = cipher.encryptor()
ct = enc.update(msg) + enc.finalize()
ret = {"ct": ct, "nonce": nonce}
return pickle.dumps(ret)

def decrypt(self, ct):
    ct = pickle.loads(ct)
    nonce = ct["nonce"]
    cipher = Cipher(
        algorithms.AES(self.shared_key[:encAES.ENCRIPTION_KEY_SIZE]),
        modes.CTR(nonce),
        backend=self.backend)
    dec = cipher.encryptor()
    msg = dec.update(ct["ct"]) + dec.finalize()
    return msg

def mac(self, msg):
    macer = hmac.HMAC(
        self.shared_key[encAES.ENCRIPTION_KEY_SIZE:encAES.HMAC_KEY_SIZE],
        hashes.SHA256(), backend=self.backend)
    macer.update(msg)
    return macer.finalize()

def verifyMac(self, msg, mac):
    macer = hmac.HMAC(
        self.shared_key[encAES.ENCRIPTION_KEY_SIZE:encAES.HMAC_KEY_SIZE],
        hashes.SHA256(), backend=self.backend)
    macer.update(msg)
    macer.verify(mac)

def encryptThenMac(self, msg):
    dump = self.encrypt(msg.encode())
    mac = self.mac(dump)
    return pickle.dumps({"dump": dump, "mac": mac})

def decryptThenMac(self, ct):
    ct_dump = pickle.loads(ct)
    try:
        self.verifyMac(ct_dump["dump"], ct_dump["mac"])
        return self.decrypt(ct_dump["dump"])
    except InvalidSignature as In:
        print("INVALID")
        return None

```

Note-se ainda que existem mais metodos usados nestas funções que não estão aqui expostos

que correspondem a chamada a biblioteca Cryptography, como metodos de serialização e deserialização para envio e processamento dos dados.

### 3 Exemplo de Utilização

Seguidamente, mostra-se dois exemplos de códigos que corresponde a utilização destas classes.

```
In [ ]: import DH_AES_DSA.encAES as encAES
import DH_AES_DSA.Emitter as Emitter
from cryptography.hazmat.backends import default_backend

dsa_private = b'-----BEGIN PRIVATE KEY-----\nMIIBSwIBADCCASwGBYqGSM44BAEwggEfAoGBANjO/
dsa_public = b'-----BEGIN PUBLIC KEY-----\nMIIBtjCCASsGBYqGSM44BAEwggEeAoGBAJ4f0ZDjyq9

enc = encAES.encAES(encAES.decodePublicKey(dsa_public,default_backend()),
                    encAES.decodePrivateKey(dsa_private,default_backend()))
emi = Emitter.Emitter(enc)
emi.run()
```

```
In [2]: from DH_AES_DSA.Receiver import Receiver

dsa_private =b'-----BEGIN PRIVATE KEY-----\nMIIBSgIBADCCASsGBYqGSM44BAEwggEeAoGBAJ4f0Z
dsa_public = b'-----BEGIN PUBLIC KEY-----\nMIIBtzCCASwGBYqGSM44BAEwggEfAoGBANjO/ORXzN+

enc = encAES.encAES(encAES.decodePublicKey(dsa_public,default_backend()),
                    encAES.decodePrivateKey(dsa_private,default_backend()))
rec = Receiver(8001,"localhost",enc)
rec.run()
```

Starting key Agreement  
Receiving with encryption

```
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
b''
```

b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
b''  
bye bye



### 3.1 Exercício 2

Adaptação do exercício 1 baseada no uso de algoritmos com Curvas Elípticas.

Para a geração das chaves criou-se mais uma vez um script.py. O resultado de correr este script foi posteriormente adicionado aos ficheiros Emitter.py e Receiver.py como as chaves publicas e privadas dos mesmos.

Nos ficheiros Emitter.py e Receiver.py as diferenças passam por substituir todas as referências feitas nestes á classe “encAES” do ficheiro “encAES.py” pela classe “encChaCha20Poly1305” do ficheiro “encChaCha20Poly1305.py”.

#### 3.1.1 Script.py

```
In [ ]: from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import serialization
        from cryptography.hazmat.primitives.asymmetric import ec

        # geração da chave privada com curvas elípticas genérico
        priv_key_ECDSA = ec.generate_private_key(ec.SECP256R1(), default_backend())
        priv_key_ECDSA_bytes = priv_key_ECDSA.private_bytes(
                                encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
                                encryption_algorithm=serialization.NoEncryption())

        # geração da chave publica com curvas elípticas genérico
        pub_key_ECDSA = priv_key_ECDSA.public_key()
        pub_key_ECDSA_bytes = pub_key_ECDSA.public_bytes(encoding=serialization.Encoding.PEM,
                                                         format=serialization.PublicFormat.SubjectPublicKeyInfo)

        #print das chaves geradas
        print(priv_key_ECDSA_bytes)
        print(pub_key_ECDSA_bytes)
```

#### 3.1.2 Emitter.py

```
In [ ]: import socket
        import encChaCha20Poly1305
        from cryptography.hazmat.backends import default_backend

        class Emitter():
            #criação do Emitter
            def __init__(self, crypto):
                self.crypto = crypto

            #estabelecimento da conexão emitter-receiver
            def connect(self, host, port):
                self.crypto.gen_ephemeral_key()

                with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as so:
```

```

        so.connect((host,port))
        print("Starting key Agreement")
        isAgreed = self.crypto.keyAgreementE(so)
        if isAgreed:
            print("Messaging with encryption")
            self.crypto.messaging(so)

    def run(self):
        self.crypto.setParameters(b'-----BEGIN DH PARAMETERS-----\nMEYCQC+ncO/Ujb2mfS
        # voltar a conetar?
        self.connect("localhost",8002)

emitter_private = b'-----BEGIN PRIVATE KEY-----\nMIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AweHBB
receiver_public = b'-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEuu

#inicialização da classe encChaCha20Poly1305
enc = encChaCha20Poly1305.encChaCha20Poly1305(
    encChaCha20Poly1305.decodePublicKey(receiver_public,default_backend()),
    encChaCha20Poly1305.decodePrivateKey(emitter_private,default_backend()))
#criação do Emitter
emi = Emitter(enc)
#chamada ao estabelecimento da conexão
emi.run()

```

### 3.1.3 Receiver.py

```

In [ ]: import socket
import encChaCha20Poly1305
from cryptography.hazmat.backends import default_backend

class Receiver():
    #criação do Receiver
    def __init__(self,port,host,crypto):
        self.port = port
        self.host = host
        self.crypto = crypto
        self.connection = None

    #estabelecimento da conexão emitter-receiver
    def connect(self):
        self.crypto.gen_ephemeral_key()

        with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as so:
            so.bind((self.host,self.port))
            so.listen()
            connect,address = so.accept()
            with connect:
                print("Starting key Agreement")

```

```

        isAgreed = self.crypto.keyAgreementR(connect)
        if isAgreed:
            print("Receiving with encryption")
            self.crypto.receiving(connect)

    def run(self):
        self.crypto.setParameters(b'-----BEGIN DH PARAMETERS-----\nMEYCQC+nC0/Ujb2mfS
        # voltar a conetar?
        self.connect()

receiver_private = b'-----BEGIN PRIVATE KEY-----\nMIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHB
emitter_public = b'-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEW8B

#inicialização da classe encChaCha20Poly1305
enc = encChaCha20Poly1305(encChaCha20Poly1305(
    encChaCha20Poly1305.decodePublicKey(emitter_public,default_backend()),
    encChaCha20Poly1305.decodePrivateKey(receiver_private,default_backend())))
#criação do Receiver
rec = Receiver(8002,"localhost",enc)
#chamada ao estabelecimento da conexão
rec.run()

```

### 3.1.4 encChaCha20Poly1305

Neste ficheiro, a adaptação passou por:

Na definição da função encrypt:

- Modificar o nonce de `os.urandom(16)` para `os.urandom(12)`.

Nas definições das funções encrypt e decrypt:

- Modificar a criação da cifra para `ChaCha20Poly1305(self.shared_key[:encChaCha20Poly1305.ENCRYPTION_KEY_SIZE])`.

Nas definições das funções sign e verifySign:

- Substituir o código `"hashes.SHA256()"` pela respetiva versão para curvas elípticas `"ec.ECDSA(hashes.SHA256())"`.

```

In [ ]: import os
        import pickle

        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives.asymmetric import dh
        from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
        from cryptography.hazmat.primitives import hashes,hmac
        from cryptography.hazmat.primitives.serialization import Encoding,ParameterFormat
        from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
        from cryptography.exceptions import InvalidSignature

```

```

from cryptography.hazmat.primitives.serialization import load_pem_private_key,
    load_pem_parameters, load_pem_public_key, PublicFormat, ParameterFormat
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.hazmat.primitives.asymmetric import ec

class encChaCha20Poly1305():

    RCV_BYTES = 1024
    HMAC_KEY_SIZE = 32
    ENCRYPTION_KEY_SIZE = 32

    #inicialização da classe encChaCha20Poly1305
    def __init__(self, dsa_public_key, dsa_private_key):
        self.parameters = None # Parametros para o AES
        self.private_key = None # static chave privada
        self.public_key = None # static chave publica
        self.e_private_key = None # ephemeral chave privada
        self.e_public_key = None # ephemeral chave publica
        self.shared_key = None # chave derivada a partir do segredo compartilhado
        self.dsa_private_key = dsa_private_key # Chave privada DSA
        self.dsa_public_key = dsa_public_key # Chave pública DSA do outro
        self.backend = default_backend()

    #geração dos parâmetros da chave
    def gen_key_params(self):
        self.parameters = dh.generate_parameters(generator=2, key_size=512,
                                                backend = default_backend())

        self.private_key = self.parameters.generate_private_key()
        self.public_key = self.private_key.public_key()

    #geração da chave da sessão
    def gen_ephemeral_key(self):
        self.e_private_key = self.parameters.generate_private_key()
        self.e_public_key = self.e_private_key.public_key()

    #atribuição dos parâmetros
    def setParameters(self, parameters):
        parametersD = decodeParameters(parameters, self.backend)
        if isinstance(parametersD, dh.DHParameters):
            self.parameters = parametersD
            self.private_key = self.parameters.generate_private_key()
            self.public_key = self.private_key.public_key()
            return True
        return False

    #geração do segredo compartilhado
    def generateSharedSecret(self, publicKey, privateKey):
        publicKeyD = decodePublicKey(publicKey, self.backend)

```

```

        if isinstance(publicKeyD,dh.DHPublicKey):
            return privateKey.exchange(publicKeyD)
        return None

#geração da chave partilhada
def generateSharedKey(self,sSharedSecret,eSharedSecret,salt=b"0"):
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),
                      length=(encChaCha20Poly1305.ENCRYPTION_KEY_SIZE
                              + encChaCha20Poly1305.HMAC_KEY_SIZE),
                      salt=salt,iterations=100,
                      backend=self.backend)

    self.shared_key = kdf.derive(sSharedSecret + eSharedSecret)
    sSharedSecret = None
    eSharedSecret = None

#encriptação
def encrypt(self,msg):
    nonce = os.urandom(12)
    #frase extra para complicar
    aad = b"frase estranha para complicar a cifragem"

    #cypher generation
    cip = ChaCha20Poly1305(
        self.shared_key[:encChaCha20Poly1305.ENCRYPTION_KEY_SIZE])

    #encryption ChaCha20Poly1305
    ct = cip.encrypt(nonce, msg, aad)

    ret = {"ct": ct, "nonce": nonce}
    return pickle.dumps(ret)

#desencriptação
def decrypt(self,ct):
    #nonce retrieval
    ct = pickle.loads(ct)
    nonce = ct["nonce"]
    aad = b"frase estranha para complicar a cifragem"

    #cypher generation
    cip = ChaCha20Poly1305(
        self.shared_key[:encChaCha20Poly1305.ENCRYPTION_KEY_SIZE])

    msg = cip.decrypt(nonce, ct["ct"], aad)

    return msg

#generation of MAC over a message

```

```

def mac(self,msg):
    macer = hmac.HMAC(
        self.shared_key[encChaCha20Poly1305.ENCRYPTION_KEY_SIZE:
                        encChaCha20Poly1305.HMAC_KEY_SIZE],
        hashes.SHA256(),backend=self.backend)
    macer.update(msg)
    return macer.finalize()

#verification of MAC
def verifyMac(self,msg,mac):
    macer = hmac.HMAC(
        self.shared_key[encChaCha20Poly1305.ENCRYPTION_KEY_SIZE:
                        encChaCha20Poly1305.HMAC_KEY_SIZE],
        hashes.SHA256(),backend=self.backend)
    macer.update(msg)
    macer.verify(mac)

# encryption followed by MAC
def encryptThenMac(self,msg):
    dump = self.encrypt(msg.encode())
    mac = self.mac(dump)
    return pickle.dumps({"dump": dump,"mac":mac})

#decryption after verifying MAC
def decryptThenMac(self,ct):
    ct_dump = pickle.loads(ct)
    try:
        self.verifyMac(ct_dump["dump"],ct_dump["mac"])
        return self.decrypt(ct_dump["dump"])
    except InvalidSignature as In:
        print("INVALID")
        return None

#sign message for authentication
def sign(self,msg):
    return self.dsa_private_key.sign(msg,ec.ECDSA(hashes.SHA256()))

#verifying message authentication
def verifySign(self,msg,signature):
    self.dsa_public_key.verify(signature,msg,ec.ECDSA(hashes.SHA256()))

def keyAgreementE(self,connection):

    # static
    connection.send(encodePublicKey(self.public_key))
    pk = connection.recv(encChaCha20Poly1305.RCV_BYTES)
    static_shared_secret = self.generateSharedSecret(pk,self.private_key)

```

```

# ephemeralcryptography.hazmat.primitives.asymmetric.
connection.send(encodePublicKey(self.e_public_key))
e_pk_mac = connection.recv(encChaCha20Poly1305.RCV_BYTES)
e_pk_mac_load = pickle.loads(e_pk_mac)
e_shared_secret = self.generateSharedSecret(e_pk_mac_load["e_key"],
                                             self.e_private_key)

# shared key
self.generateSharedKey(static_shared_secret,e_shared_secret)

#DSA

sign = self.decrypt(e_pk_mac_load["signature"])
try:
    self.verifySign(pk + encodePublicKey(self.public_key) +
                    e_pk_mac_load["e_key"] +
                    encodePublicKey(self.e_public_key) ,sign)
except InvalidSignature as In:
    #connection.send(pickle.dumps({"mac": "mac","signature":"signature"}))
    print("Invalid Signature")
    return False

# test confirmation
try:
    self.verifyMac(b"KC_1_V" + encodePublicKey(self.e_public_key) +
                  e_pk_mac_load["e_key"],e_pk_mac_load["mac"])
except InvalidSignature as In:
    #connection.send(pickle.dumps({"mac": "mac","signature":"signature"}))
    print("Key Confirmation Failed")
    return False

# Send mac and sign
mac_and_sign = {"mac": self.mac(b"KC_1_U" + encodePublicKey(self.e_public_key)
                               + e_pk_mac_load["e_key"]
                               ),
                "signature": self.encrypt(self.sign(
                    encodePublicKey(self.public_key) + pk +
                    encodePublicKey(self.e_public_key) +
                    e_pk_mac_load["e_key"] )) }
connection.send(pickle.dumps(mac_and_sign))

e_pk_mac = None
e_pk_mac_load
e_shared_secret = None
static_shared_secret = None
pk = None

```

```

        return True

def keyAgreementR(self, connection):
    # static
    pk = connection.recv(encChaCha20Poly1305.RCV_BYTES)
    connection.send(encodePublicKey(self.public_key))
    static_shared_secret = self.generateSharedSecret(pk, self.private_key)

    # ephemeral
    e_pk = connection.recv(encChaCha20Poly1305.RCV_BYTES)
    e_shared_secret = self.generateSharedSecret(e_pk, self.e_private_key)

    # shared key
    self.generateSharedKey(static_shared_secret, e_shared_secret)

    #key confirmation
    key_and_mac_and_sig = pickle.dumps(
        {
            "e_key": encodePublicKey(self.e_public_key),
            "mac": self.mac(b"KC_1_V" + e_pk + encodePublicKey(self.e_public_key)),
            "signature": self.encrypt(self.sign(
                encodePublicKey(self.public_key) + pk +
                encodePublicKey(self.e_public_key) + e_pk))
        }
    )
    connection.send(key_and_mac_and_sig)

    # mac verification
    mac_and_sign = connection.recv(encChaCha20Poly1305.RCV_BYTES)
    mac_and_sign_load = pickle.loads(mac_and_sign)

    sign = self.decrypt(mac_and_sign_load["signature"])
    try:
        self.verifySign(pk + encodePublicKey(self.public_key) + e_pk +
            encodePublicKey(self.e_public_key), sign)
    except InvalidSignature as In:
        #connection.send(pickle.dumps({"mac": "mac", "signature": "signature"}))
        print("Invalid Signature")
        return False

    try:
        self.verifyMac(b"KC_1_U" + e_pk + encodePublicKey(self.e_public_key),
            mac_and_sign_load["mac"])
    except InvalidSignature as In:
        print("Key Confirmation Failed")
        return False

```



```

        e_pk = None
        e_shared_secret = None
        static_shared_secret = None
        pk = None
        key_and_mac = None

    return True

#getting message to emite
def messaging(self,connection):
    print("Now you can send messages")
    while True:
        data = input("----> ")
        encData = self.encryptThenMac(data)
        connection.send(encData)
        if "Exit" == data:
            break

#retrieving message emitted
def receiving(self,connection):
    while True:
        try:
            data = connection.recv(encChaCha20Poly1305.RCV_BYTES)
            dencData = self.decryptThenMac(data)
            print(dencData)
        except EOFError as e:
            print("bye bye")
            break

def encodeParameters(parameters):
    return parameters.parameter_bytes(Encoding.PEM,ParameterFormat.PKCS3)

def decodeParameters(parameters,backend):
    return load_pem_parameters(parameters,backend=backend)

def encodePublicKey(publicKey):
    return publicKey.public_bytes(Encoding.PEM,PublicFormat.SubjectPublicKeyInfo)

def decodePublicKey(key,backend):
    return load_pem_public_key(key,backend=backend)

def decodePrivateKey(key,backend):
    return load_pem_private_key(key,None,backend=backend)

```

### 3.1.5 Instruções de utilização dos ficheiros anexados

Anexados seguem as pastas dos dois exercícios com os respetivos ficheiros script.py, Emitter.py e Receiver.py. As instruções seguintes dizem respeito ao modo de proceder dentro de cada pasta.

Para gerar as chaves privada e pública para o Emitter e o Receiver basta na linha de comandos executar o ficheiro script.py com o seguinte comando:

```
python script.py
```

Posteriormente para correr o estabelecimento da conexão emitter-receiver basta abrir 2 terminais na pasta onde os ficheiros Emitter.py e Receiver.py se encontram e posteriormente executar por ordem os seguintes comandos, um em cada terminal:

```
python Receiver.py
```

```
python Emitter.py
```