MIEI/MI - Estruturas Criptográficas

Trabalho Prático 1

João Alves
a77070@alunos.uminho.pt

Nuno Leite
a70132@alunos.uminho.pt

Universidade do Minho
12 de Março de 2019

1 Introdução

A resolução deste trabalho prático tem como objetivo implementar um abiente seguro, constituído por um emissor e um recetor, através de sessões cifradas e seguras. A solução implmentada, foi estruturada da seguinte forma:

- Construção de uma sessão de comunicação síncrona entre um emissor e um recetor.
- Utilização do protocolo de **Diffie_Hellman** com verificação de chave e autenticação dos agentes, na sessão anteriormente referida.
- Utilização do **DSA** para a autenticação dos agentes.
- Utilização do **TAES** como cifra simétrica, com autenticação do criptograma em cada superbloco.
- Implementar novamente o cenário anterior mas, desta feita vez, utilizando o **ECDH** (Elliptic Curve Diffie-Hellman) em vez do **DH** e o **ECDSA** (Elliptic Curve Digital Signature Algorithm) em vez do **DSA**.

Este relatório está escrito de forma a que o texto escrito entre o código desenvolvido seja suficientemente explicativo do que está a ser implementado, o que, na nossa opinião, permite uma leitura e compreensão facilitada do mesmo.

2 Código comum

Esta secção tem como objetivo definir *snippets* de código que serão utilizados tanto na definição da sessão síncrona com **DH** como na sessão síncrona com **ECDH**, tais como os *imports* de módulos necessários, as funções auxiliares necessárias e a classe de multiprocessamento.

2.1 Imports

De seguida, encontram-se os módulos **Python** importados e necessários para desenvolver a sessão síncrona entre os agentes, utilizando, em primeira instância, o protocolo **DH** e o algoritmo de assinaturas **DSA** e, numa segunda instância, o protocolo **ECDH** e o algoritmo de assinaturas **ECDSA**.

```
In [1]: import os,io,sys
        from multiprocessing import Process,Pipe
        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import hashes, hmac
        from cryptography.exceptions import *
        from cryptography.hazmat.primitives.asymmetric import dsa
        from cryptography.hazmat.primitives.asymmetric import dh
        from cryptography.hazmat.primitives.asymmetric import ec
        from cryptography.hazmat.primitives import serialization
        from cryptography.hazmat.primitives.kdf.hkdf import HKDF
        from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

2.2 Definição das funções do processo criptográfico

Esta secção tem o objetivo de definir funções que serão utilizadas nas sessões síncronas, tanto na que utiliza **DH** como na que utiliza **ECDH**. Essas funções são as seguintes:

- `Hash(s)`, que tem como propósito calcular um *digest* de uma dada mensagem.
- `next_tweak(initial,current_counter)`, que tem como propósito calcular o próximo *tweak* a utilizar no bloco que vai ser cifrado/decifrado, com base no *nounce* inicial e no número do bloco que está a ser processadoa atualmente.
- `bytes_xor(a,b)`, que tem como propósito realizar o *XOR byte* a *byte* de duas dadas sequências de _bytes.
- `calculate_nounce()`, que tem como objectivo calcular o nounce a utilizar em parte de um *tweak*, tendo em conta os *nounces* que já foram utilizados anteriormente.
- `calculate_parity(blocks,number_blocks)`, que tem como objetivo calcular a paridade de um conjunto de blocos, para que seja gerada a sua *tag* de autenticação.
- `calculate_auth_tweak(initial_tweak,number_blocks)`, que tem como objetivo calcular o *tweak* utilizado para cifrar a paridade dos blocos cifrados.
- As funções `cifra_blocos(current_counter,initial_tweak,blocks,number_blocks,cipher_context)` e `decifra_blocos(current_counter,initial_tweak,blocks,number_blocks,decipher_context)` têm como objetivo cifrar/decifrar, respetivamente, um conjunto de blocos aplicando a cada bloco a função de cifra ou de decifragem.
- As funções `cifra(block,tweak,cipher_context)` e `decifra(block,tweak,decipher_context` têm como objetivo cifrar/decifrar um dado bloco utilizando para o efeito a função $\tilde{E}(s,x) = E(s, x \oplus h(w)) \oplus h(w)$.

É relevante referir que as operações de cifra e de decifra utilizam exatamente a mesma fórmula e, logo, são iguais, mas estão separadas apenas por uma questão de mais simples compreensão do problema em questão, bem como da sua resolução. Além disso, neste processo criptográfico foi utilizado o modo **TAE** (*Tweakable Authenticated Encryption*).

```
In [2]: def Hash(s):
            digest = hashes.Hash(hashes.SHA256(),backend=default_backend())
```

```
    digest.update(s)
    return digest.finalize()

def next_tweak(initial, current_counter):
    str = format(current_counter, '063b')
    str = str + '0'
    b_str = int(str,2).to_bytes((len(str) + 7) // 8, 'big')
    return (initial + b_str)

def bytes_xor(a, b) :
    return bytes(x ^ y for x, y in zip(a, b))

def calculate_nounce():
    if not(os.path.isfile('nounces.txt')):
        os.mknod('nounces.txt')
    nounces_file = open('nounces.txt','r')
    nounces_used = nounces_file.read().split('\n')
    nounces_file.close()
    nounces_file = open('nounces.txt','a')
    nounce_to_send = bytes('','utf8')
    ok = False
    while not(ok):
        anyMatch = False
        nounce = os.urandom(8)
        for x in nounces_used:
            if x == str(nounce):
                anyMatch = True
                break
        if not(anyMatch):
            nounces_file.write(str(nounce) + '\n')
            nounce_to_send = nounce
            break
    nounces_file.close()
    return nounce_to_send

#def calculate_parity(blocks,number_blocks):
#    str = format(0,'128b') # O número 0 é elemento neutro no XOR
#    parity = int(str,2).to_bytes((len(str) + 7) // 8, 'big')
#    bytes_processed = 0
#    while(bytes_processed < (16*number_blocks)):
#        max_bytes = bytes_processed + 16
#        block_to_xor = blocks[bytes_processed:max_bytes]
#        parity = bytes_xor(parity,block_to_xor)
#        bytes_processed += 16
#    return parity

def calculate_auth_tweak(initial_tweak,number_blocks):
    str = format(number_blocks*16,'063b')
```

```
    str = str + '1'
    b_str = int(str,2).to_bytes((len(str) + 7) // 8, 'big')
    return (initial_tweak + b_str)

def cifra_blocos(current_counter,initial_tweak,blocks,number_blocks,cipher_context):
    str = format(0,'128b') # 0 bit 0 é elemento neutro no XOR
    parity = int(str,2).to_bytes((len(str) + 7) // 8, 'big')
    bytes_processed = 0
    cryptogram = bytearray(160)
    while(bytes_processed < (16*number_blocks)):
        max_bytes = bytes_processed + 16
        block_to_process = blocks[bytes_processed:max_bytes]
        parity = bytes_xor(parity,block_to_process)
        tweak = next_tweak(initial_tweak,current_counter)
        block_crypt = cifra(block_to_process,tweak,cipher_context)
        cryptogram[bytes_processed:max_bytes] = block_crypt
        bytes_processed += 16
        current_counter += 1
    return {"crypt": bytes(cryptogram), "parity": parity ,"i": current_counter}


def cifra(block,tweak,cipher_context):
    hw = Hash(tweak)
    x_xor_hw = bytes_xor(block,hw)
    e_s_x_hw = cipher_context.update(x_xor_hw)
    cryptogram = bytes_xor(e_s_x_hw, hw)
    return cryptogram

def decifra_blocos(current_counter,initial_tweak,blocks,number_blocks,decipher_context):
    str = format(0,'128b') # 0 bit 0 é elemento neutro no XOR
    parity = int(str,2).to_bytes((len(str) + 7) // 8, 'big')
    bytes_processed = 0
    text = bytearray(160)
    while(bytes_processed < (16*number_blocks)):
        max_bytes = bytes_processed + 16
        block_to_process = blocks[bytes_processed:max_bytes]
        tweak = next_tweak(initial_tweak,current_counter)
        block_text = decifra(block_to_process,tweak, decipher_context)
        parity = bytes_xor(parity,block_text)
        text[bytes_processed:max_bytes] = block_text
        bytes_processed += 16
        current_counter += 1
    return {"text": bytes(text), "parity": parity, "i": current_counter}

def decifra(block,tweak,decipher_context):
    hw = Hash(tweak)
    x_xor_hw = bytes_xor(block,hw)
    e_s_x_hw = decipher_context.update(x_xor_hw)
```

```
            cryptogram = bytes_xor(e_s_x_hw, hw)
            return cryptogram
```

2.3 Definição da classe de multiprocessamento

No seguinte código, é definida a classe de multiprocessamento, que permite uma comunicação bidireccional com o *Emitter* e o *Receiver*, sendo estes dois processos criados e implementados pela **API multiprocessing**.

```
In [3]: class BiConnection(object):
            def __init__(self,left,right):
                left_side,right_side = Pipe()
                self.timeout = None
                self.left_process = Process(target=left,args=(left_side,))
                self.right_process = Process(target=right,args=(right_side,))
                self.left = lambda : left(left_side)
                self.right = lambda : right(right_side)

            def auto(self,proc=None):
                if proc == None:
                    self.left_process.start()
                    self.right_process.start()
                    self.left_process.join(self.timeout)
                    self.right_process.join(self.timeout)
                else:
                    proc.start()
                    proc.join()
```

2.4 Definição do algoritmo utilizado

Esta secção tem como objetivo definir o algoritmo utilizado bem como algumas especificidades em ambas as sessões. O algoritmo utilizado foi exatamente o mesmo, apenas aspetos específicos tais como geração das chaves diferenciaram. O algoritmo aqui definido pretende apenas apresentar o seu modo de funcionamento de uma forma geral, textualmente. Além disso, é conveniente referir que os agentes da sessão síncrona **DH** utilizam chaves permanentes de 3072 bits **DSA** e os agentes da sessão síncrona **ECDH** utilizam chaves permanentes baseadas na curva *NIST P-256*.Neste ponto de início do algoritmo assumimos também que os parâmetros **DH** já estão definidos. Assim sendo, o programa inicia da seguinte forma:

Protocolo de acordo de chaves DH/ECDH com verificação de chave

1. Emissor gera a sua chave privada e chave pública **DH**.
2. Emissor envia para o Recetor a sua chave pública em bytes e a assinatura produzida pela sua chave privada **DSA** sobre a chave pública em bytes.
3. Recetor verifica a mensagem recebida com a chave pública **DSA** do Emissor.
4. Recetor gera a sua chave privada e chave pública **DH**.
5. Recetor envia para o Emissor a sua chave pública em bytes e a assinatura produzida pela sua chave privada **DSA** sobre a chave pública em bytes.

6. Emissor verifica a mensagem recebida com a chave pública **DSA** do Recetor.
7. Emissor produz a chave mestra (g^{xy}), a chave partilhada (derivada da chave mestra através de um *KDF*)
8. Emissor calcula um *Hash* da chave partilhada e envia, juntamente com a assinatura, para o Recetor.
9. Recetor verifica a mensagem recebida.
10. Recetor produz a chave mestra (g^{yx}), a chave partilhada (derivada da chave mestra através de um *KDF*)
11. Recetor calcula o *Hash* da chave partilhada e verifica se coincide com o *digest* recebido.
12. Se a verificação de chave foi bem sucedida, continua para o processo de troca de informação segura.

Processo de troca de informação segura

1. Calculam-se os bytes correspondentes à mensagem a ser enviada, bem como a *stream* de *bytes*.
2. É calculado o tweak inicial (*nounce* de 64 bits) e enviado para o Recetor.
3. É criado o contexto de cifra ou de decifra.
4. Enquanto que existirem blocos para cifrar/decifrar:
    1. Emissor cifra um superbloco de cada vez (10 blocos singulares)
    2. Emissor calcula a *tag* de autenticação do superbloco
    3. Emissor envia para o recetor o criptograma e o *tag* de autenticação e espera por uma mensagem de confirmação do recetor para prosseguir.
    4. Recetor decifra o superbloco
    5. Recetor verifica a *tag* de autenticação após calcular a sua.
    6. Se a *tag* estiver correta envia uma mensagem afirmativa ao Emissor, caso contrário o programa termina com falha de autenticação.
5. Finalmente o emissor envia uma mensagem de finalização do processo para o Recetor e fecha a conexão.
6. Recetor reconhece a mensagem de finalização do processo e fecha também a conexão.

3 Sessão Síncrona com *Diffie-Hellman*

3.1 Geração dos parâmetros *Diffie-Hellman*

A próxima secção implementa a geração dos parâmetros necessários para a derivação de chaves **DH** e **DSA**, por parte de ambos os agentes.

```
In [4]: parameters_dh = dh.generate_parameters(generator=2,key_size=3072,backend=default_backend
```

3.2 Geração das chaves permanentes dos agentes

Esta secção tem como objetivo gerar as chaves permanentes dos agentes envolvidos na comunicação, tendo em conta que estas são conhecidas por ambos, e serão aplicadas na assinatura digital das mensagens.

```
In [5]: # Geração das chaves do Emissor
        emitter_dsa_sk = dsa.generate_private_key(3072,default_backend()) #chave privada DSA
```

```
emitter_dsa_pk = emitter_dsa_sk.public_key() #chave pública DSA

# Geração das chaves do Recetor
receiver_dsa_sk = dsa.generate_private_key(3072,default_backend()) #chave privada DSA
receiver_dsa_pk = receiver_dsa_sk.public_key() #chave pública DSA
```

3.3 Definição do Emissor e do Recetor

Por fim definiu-se o comportamento de cada um dos agentes nesta sessão síncrona segura de troca de informação. Na definição de ambos, é assumido que as chaves públicas **DSA** utilizadas na autenticação dos agentes já são conhecidas por ambos. As chaves **DH** são geradas por sessão de modo a acordar uma chave temporária, para que seja possível cifrar e decifrar uma mensagem utilizando uma primitiva simétrica.

```
In [6]: def Emitter(connection):
            #Implementação Protocolo Diffie-Hellman

            emitter_dh_sk = parameters_dh.generate_private_key()
            emitter_dh_pk = emitter_dh_sk.public_key()
            pub = emitter_dh_pk.public_bytes(
                encoding = serialization.Encoding.PEM,
                format = serialization.PublicFormat.SubjectPublicKeyInfo)
            signature = emitter_dsa_sk.sign(pub,hashes.SHA256())
            message = {'pk': pub, 'sig': signature}
            connection.send(message)
            receiver_first = connection.recv()
            receiver_dh_public_key = serialization.load_pem_public_key(
                receiver_first['pk'],default_backend())
            try:
                receiver_dsa_pk.verify(receiver_first['sig'],receiver_first['pk'],
                    hashes.SHA256())
            except InvalidSignature as i:
                sys.exit("Assinatura de chaves inválida!")
            master_key = emitter_dh_sk.exchange(receiver_dh_public_key)

            #Passar a chave de sessão acordada (master_key) por um kdf
            shared_key = HKDF(
                algorithm = hashes.SHA256(),
                length = 32,
                salt = None,
                info = b'exchange data',
                backend = default_backend()
            ).derive(master_key)

            #Verificar a chave, enviando um hash da chave derivada para confirmar
            #que tanto emissor como recetor possuem a mesma chave.

            key_digest = Hash(shared_key)
```

```
signature = emitter_dsa_sk.sign(key_digest,hashes.SHA256())
connection.send({'digest': key_digest, 'sig': signature})

key_check_obj = connection.recv()
try:
    receiver_dsa_pk.verify(key_check_obj["sig"],bytes(key_check_obj["message"],
        'utf8'),hashes.SHA256())
except InvalidSignature as i:
    sys.exit('Assinatura de chaves inválida!')

if key_check_obj["message"] == "OK":
    #Ambos agentes possuem a mesma chave. Prosseguir com o processo de cifra.
    message_size = 1600
    gen_bytes = os.urandom(message_size)

    #imprimir bytes a cifrar para posterior comparação
    print('gen_bytes')
    print(gen_bytes)

    inputs = io.BytesIO(gen_bytes)

    #inicialização do tweak como um nounce de 64 bits (n/2)
    #e envio para o agente Receiver
    tweak_init = calculate_nounce()
    connection.send(tweak_init)

    buffer = bytearray(160)
    cipher = Cipher(algorithms.AES(shared_key),modes.ECB(),
        backend=default_backend()).encryptor()
    i = 1 # nr de bloco
    try:
        while inputs.readinto(buffer):
            cipher_object = cifra_blocos(i,tweak_init,buffer,10,cipher)
            cryptogram = cipher_object["crypt"]
            parity = cipher_object["parity"]
            auth_tweak = calculate_auth_tweak(tweak_init,10)
            auth_tag = cifra(parity,auth_tweak,cipher)
            i = cipher_object["i"]
            msg_to_send = {'crypt': cryptogram, 'tag': auth_tag}
            connection.send(msg_to_send)
            msg = connection.recv()
            if(not(msg == 'OK')):
                sys.exit("Autenticação num superbloco falhou!")
    except Exception as err:
        print('Erro no emissor! {0}'.format(err))
    connection.send('finalized')
    inputs.close()
else:
```

```
            print('Chaves não coincidiram')
        connection.close()

In [7]: def Receiver(connection):
        #Implementação protocolo Diffie-Hellman
        emitter_first = connection.recv()
        receiver_dh_sk = parameters_dh.generate_private_key()
        receiver_dh_pk = receiver_dh_sk.public_key()
        emitter_dh_public_key = serialization.load_pem_public_key(emitter_first['pk'],
            default_backend())
        try:
            emitter_dsa_pk.verify(emitter_first['sig'],emitter_first['pk'],hashes.SHA256())
        except InvalidSignature as i:
            sys.exit('Assinatura de chaves inválida!')
        master_key = receiver_dh_sk.exchange(emitter_dh_public_key)
        pub = receiver_dh_pk.public_bytes(
            encoding = serialization.Encoding.PEM,
            format = serialization.PublicFormat.SubjectPublicKeyInfo)
        signature = receiver_dsa_sk.sign(pub,hashes.SHA256())
        message = {'pk':pub,'sig':signature}
        connection.send(message)

        #Passar a chave de sessão acordada (master_key) por um kdf
        shared_key = HKDF(
            algorithm = hashes.SHA256(),
            length = 32,
            salt = None,
            info = b'exchange data',
            backend = default_backend()
        ).derive(master_key)

        key_check_obj = connection.recv()
        try:
            emitter_dsa_pk.verify(key_check_obj['sig'],key_check_obj['digest'],
                hashes.SHA256())
        except InvalidSignature as i:
            sys.exit("Assinatura de chaves inválida!")
        digest = Hash(shared_key)
        if digest == key_check_obj["digest"]:
            message = "OK"
            signature = receiver_dsa_sk.sign(bytes(message,'utf8'),hashes.SHA256())
            obj = {"message":message, 'sig': signature}
            connection.send(obj)

            #prosseguir com o modo de decifra síncrono

            outputs = io.BytesIO()
            tweak_init = connection.recv()
```

```
                cipher = Cipher(algorithms.AES(shared_key),modes.ECB(),
                    backend=default_backend()).decryptor()
                i = 1 # nr de bloco
                try:
                    while True:
                        buffer = connection.recv()
                        if(buffer == 'finalized'):
                            outputs.write(cipher.finalize())
                            break
                        else:
                            crypt = buffer['crypt']
                            tag = buffer['tag']
                            decipher_object = decifra_blocos(i, tweak_init,crypt,10,cipher)
                            text = decipher_object["text"]
                            i = decipher_object["i"]
                            parity = decipher_object["parity"]
                            auth_tweak = calculate_auth_tweak(tweak_init,10)
                            auth_tag = decifra(tag,auth_tweak,cipher)
                            if(auth_tag == parity):
                                connection.send('OK')
                            else:
                                connection.send('AUTH ERROR')
                                sys.exit("Autenticação num superbloco falhou!")
                            outputs.write(text)
                    print('decrypted:')
                    print(outputs.getvalue())
                except Exception as Err:
                    print('Erro no recetor! {0}'.format(Err))

                outputs.close()
            else:
                print('Chaves não coincidiram')
            connection.close()
```

```
In [8]: BiConnection(Emitter,Receiver).auto()
```

```
gen_bytes
b'\xd7\xb7\xe6\x9a(i\xac\x18\xd9\x7f-\x19\xcbx&\xa6\t\xccu\xc7\x8f\xa4p\x96
\xff\r\x96\xdaM\xce\x13\x91\x88\xb6\x8e\xa6\xec-\xbd\x9c\x9a\x9f\xb0o
\xf8SUb\xc30\x8f\x98j"Sx\xc4\x9c\xae$\xe6\x08\x9cl\x17\xa8\xe2\xb6D\xd4\x06
\xe0\x80\xda\xb0\x18\xee\xac\xa6b\xaa\xe5\xe0\xa9\x91$&s0\'U\xae\xa1\x92y
\x87\x9b\x1a\x9a$\xaf\xa2}\xb9\xa9\xa3k;\xa9\xc5\x8cU\xce\xcd0\x12\xa3v
\x14\xfe\xabP\xa0\xca\x9c\xb4\x98c\xdd\x80\xc5\xf6\x1a\x87o\x8a\xeb3V{
\xe2|\xa5\xcf)\x81x\x1a\xd79b|\xe3\xa7k\x82\x83\xf9\xdc(I\xd44@\xa6J\xc1
\xf4"\x8d\xde\xb1\xd0e\x98k\xb4\xb3{RK\xbf\xa85\xf3D\xd2\x1c!\xb7\xcf
\nv\x00\x9b\r/\xa0\xe1\x83\x15\xa6\x99(\x9bkv\x02\xe0\x1d(\xeb\xfe\x0f@
```

```
\x98\xe6j\xac\x8aQ\xdb\xac\x944\xc4\xaf\x00t\xc8U\xef\xb4\xee\x872^
\x7f\x8b|ZL;\\kD\xc19\xd2f\xe6\xeaz\xfd\n\r\x9a\x17GD\x8d\xda\xd3\xabC\xa7$
\x91\x94\xe2\x0eS\xae\x99V":c\xa3\xce5\x9f\xb3]\x9f\x0f\x15\x00c\xab\x12\x14
\xa7\xd1;0\xd0\xc2\x0b\xbf\x8a7\x94J\x8e\xbf\xb1\xbc\x01\x99\xe6\xfc\xc8
\x91\x9fV\xbf!\xaa \xac\xea\xad\x0b\xca\xa8\xd0\xafTY)\xed@\x1c\xaf\xd9
\x89{!\xc0\xf3\xc1W\xf7\x02\xec\xd3\x81\x81\x03ijO\xe8\xb3\x1d.B\x84\r\x934b
\xe3`G\xa4\xee\xfd\xfa\xa6\xa3c\xe7\xacK-\xd8\xc4\x0656\xbc\xaaM\xc1\x82F\x15
\xf0\xed\x9a\xbbc\xbd@\x86\x91S\xd0\x94=\xec\xb7uRM\x01\xa7\xccg\xdf\x08\xe8\xff
decrypted:
b'\xd7\xb7\xe6\x9a(i\xac\x18\xd9\x7f-\x19\xcbx&\xa6\t\xccu\xc7\x8f\xa4p\x96
\xff\r\x96\xdaM\xce\x13\x91\x88\xb6\x8e\xa6\xec-\xbd\x9c\x9a\x9f\xb0o\xf8SUb
\xc30\x8f\x98j"Sx\xc4\x9c\xae$\xe6\x08\x9cl\x17\xa8\xe2\xb6D\xd4
\x06\xe0\x80\xda\xb0\x18\xee\xac\xa6b\xaa\xe5\xe0\xa9\x91$&s0\'U\xae\xa1\x92y
\x87\x9b\x1a\x9a$\xaf\xa2}\xb9\xa9\xa3k;\xa9\xc5\x8cU\xce\xcd0\x12\xa3v
\x14\xfe\xabP\xa0\xca\x9c\xb4\x98c\xdd\x80\xc5\xf6\x1a\x87o\x8a\xeb3V{
\xe2|\xa5\xcf)\x81x\x1a\xd79b|\xe3\xa7k\x82\x83\xf9\xdc(I\xd44@\xa6J\xc1
\xf4"\x8d\xde\xb1\xd0e\x98k\xb4\xb3{RK\xbf\xa85\xf3D\xd2\x1c!\xb7\xcf
\nv\x00\x9b\r/\xa0\xe1\x83\x15\xa6\x99(\x9bkv\x02\xe0\x1d(\xeb\xfe\x0f@
\x98\xe6j\xac\x8aQ\xdb\xac\x944\xc4\xaf\x00t\xc8U\xef\xb4\xee\x872^
\x7f\x8b|ZL;\\kD\xc19\xd2f\xe6\xeaz\xfd\n\r\x9a\x17GD\x8d\xda\xd3\xabC\xa7$
\x91\x94\xe2\x0eS\xae\x99V":c\xa3\xce5\x9f\xb3]\x9f\x0f\x15\x00c\xab\x12\x14
\xa7\xd1;0\xd0\xc2\x0b\xbf\x8a7\x94J\x8e\xbf\xb1\xbc\x01\x99\xe6\xfc\xc8
\x91\x9fV\xbf!\xaa \xac\xea\xad\x0b\xca\xa8\xd0\xafTY)\xed@\x1c\xaf\xd9
\x89{!\xc0\xf3\xc1W\xf7\x02\xec\xd3\x81\x81\x03ijO\xe8\xb3\x1d.B\x84\r\x934b
\xe3`G\xa4\xee\xfd\xfa\xa6\xa3c\xe7\xacK-\xd8\xc4\x0656\xbc\xaaM\xc1\x82F\x15
\xf0\xed\x9a\xbbc\xbd@\x86\x91S\xd0\x94=\xec\xb7uRM\x01\xa7\xccg\xdf\x08\xe8\xff
```

A comparação entre os valores dados por **gen_bytes** e por **decrypted** no output, por via da igualdade, permitem concluir que a mensagem que foi gerada é a mesma mensagem que foi decifrada.

4 Sessão Síncrona com *Elliptic Curve Diffie-Hellman*

4.1 Geração das chaves permanentes dos agentes

Esta secção tem como objetivo gerar as chaves permanentes de ambos os agentes, ou seja, as chaves que são utilizadas na autenticação dos mesmos.

```
In [9]: # Geração das chaves do emissor
        #Chave privada
        emitter_ecdsa_sk = ec.generate_private_key(ec.SECP256R1(),default_backend())
        emitter_ecdsa_pk = emitter_ecdsa_sk.public_key() #Chave pública

        # Geração das chaves do recetor
        #Chave privada
        receiver_ecdsa_sk = ec.generate_private_key(ec.SECP256R1(),default_backend())
        receiver_ecdsa_pk = receiver_ecdsa_sk.public_key() #Chave pública
```

4.2 Definição do Emissor e do Recetor

Nesta secção, é definido o comportamento de cada um dos agentes na sessão síncrona segura de troca de informação utilizando o protocolo **ECDH** e o algoritmo de assinatura **ECDSA**. É assumido que as chaves públicas **ECDSA** de cada agente já são conhecidas por ambos, pelo que podem ser utilizadas na autenticação dos agentes ao realizar o protocolo de acordo de chaves **ECDH**. As chaves **ECDH** são geradas por sessão, de forma a acordar uma chave de sessão temporária e comum que permita a utilização de uma primitiva simétrica.

```
In [10]: def Emitter(connection):
             #Implementação do protocolo Elliptic Curve Diffie-Hellman
             emitter_ecdh_sk = ec.generate_private_key(ec.SECP521R1(), default_backend())
             emitter_ecdh_pk = emitter_ecdh_sk.public_key()
             pub = emitter_ecdh_pk.public_bytes(
                 encoding = serialization.Encoding.PEM,
                 format = serialization.PublicFormat.SubjectPublicKeyInfo
             )
             signature = emitter_ecdsa_sk.sign(
                 pub,
                 ec.ECDSA(hashes.SHA256())
             )
             message_to_send = {'pub': pub, 'sig': signature}
             connection.send(message_to_send)
             receiver_info = connection.recv()
             receiver_ecdsa_pk.verify(receiver_info["sig"],receiver_info["pub"],ec.ECDSA(hashes.
             receiver_pub = serialization.load_pem_public_key(receiver_info["pub"],default_backe
             master_key = emitter_ecdh_sk.exchange(ec.ECDH(),receiver_pub)
             shared_key = HKDF(
                 algorithm = hashes.SHA256(),
                 length = 32,
                 salt = None,
                 info = b'exchange data',
                 backend = default_backend()
             ).derive(master_key)

             #Verificação da chave
             shared_key_digest = Hash(shared_key)
             signature = emitter_ecdsa_sk.sign(
                 shared_key_digest,
                 ec.ECDSA(hashes.SHA256())
             )
             msg = {'skd':shared_key_digest,'sig':signature}
             connection.send(msg)
             result = connection.recv()
             receiver_ecdsa_pk.verify(result['sig'],result['msg'],ec.ECDSA(hashes.SHA256()))
             if(result['msg'].decode('utf8') == 'OK'):
                 # Chave verificada. Iniciar processo de cifra

                 message_size = 1600
```

```
            gen_bytes = os.urandom(message_size)

            #Imprimir bytes a cifrar para posterior comparação
            print('bytes before cipher:')
            print(gen_bytes)

            inputs = io.BytesIO(gen_bytes)

            #Inicialização e envio dos primeiros 64 bits do tweak
            tweak_init = calculate_nounce()
            connection.send(tweak_init)

            buffer = bytearray(160)
            cipher = Cipher(algorithms.AES(shared_key),modes.ECB(),
               backend=default_backend()).encryptor()
            i = 1 #nr de bloco
            try:
                while inputs.readinto(buffer):
                    cipher_object = cifra_blocos(i,tweak_init,buffer,10,cipher)
                    cryptogram = cipher_object["crypt"]
                    parity = cipher_object["parity"]
                    auth_tweak = calculate_auth_tweak(tweak_init,10)
                    auth_tag = cifra(parity,auth_tweak,cipher)
                    i = cipher_object["i"]
                    msg_to_send = {'crypt': cryptogram, 'tag': auth_tag}
                    connection.send(msg_to_send)
                    msg = connection.recv()
                    if(not(msg == 'OK')):
                        sys.exit("Autenticação num superbloco falhou!")
            except Exception as err:
                print('Erro no emissor! {0}'.format(err))
            connection.send('finalized')
            inputs.close()
        else:
            print(result['msg'].decode('utf8'))
        connection.close()


In [11]: def Receiver(connection):
        #Implementação do protocolo Elliptic Curve Diffie-Hellman
        emitter_info = connection.recv()
        emitter_pub = emitter_info["pub"]
        sig = emitter_info["sig"]
        emitter_ecdsa_pk.verify(sig,emitter_pub,ec.ECDSA(hashes.SHA256()))
        receiver_ecdh_sk = ec.generate_private_key(ec.SECP521R1(),default_backend())
        receiver_ecdh_pk = receiver_ecdh_sk.public_key()
        pub = receiver_ecdh_pk.public_bytes(
            encoding = serialization.Encoding.PEM,
```

```
    format = serialization.PublicFormat.SubjectPublicKeyInfo
)
signature = receiver_ecdsa_sk.sign(
    pub,
    ec.ECDSA(hashes.SHA256())
)
message_to_send = {'pub': pub, 'sig': signature}
connection.send(message_to_send)
key_check_info = connection.recv()
emitter_ecdsa_pk.verify(key_check_info["sig"],key_check_info["skd"],
   ec.ECDSA(hashes.SHA256()))
emitter_pub = serialization.load_pem_public_key(emitter_pub,default_backend())
master_key = receiver_ecdh_sk.exchange(ec.ECDH(),emitter_pub)
shared_key = HKDF(
    algorithm = hashes.SHA256(),
    length = 32,
    salt = None,
    info = b'exchange data',
    backend = default_backend()
).derive(master_key)

#Verificação da chave
shared_key_digest = Hash(shared_key)
if(shared_key_digest == key_check_info["skd"]):
    message = bytes("OK",'utf8')
    signature = receiver_ecdsa_sk.sign(
        message,
        ec.ECDSA(hashes.SHA256())
    )
    connection.send({'msg':message, 'sig': signature})

    # Iniciar processo de decifra
    tweak_init = connection.recv()
    outputs = io.BytesIO()

    cipher = Cipher(algorithms.AES(shared_key),modes.ECB(),
       backend=default_backend()).decryptor()
    i = 1 # nr de bloco
    try:
        while True:
            buffer = connection.recv()
            if(buffer == 'finalized'):
                outputs.write(cipher.finalize())
                break
            else:
                crypt = buffer['crypt']
                tag = buffer['tag']
                decipher_object = decifra_blocos(i, tweak_init,crypt,10,cipher)
```

```
                        text = decipher_object["text"]
                        i = decipher_object["i"]
                        parity = decipher_object["parity"]
                        auth_tweak = calculate_auth_tweak(tweak_init,10)
                        auth_tag = decifra(tag,auth_tweak,cipher)
                        if(auth_tag == parity):
                            connection.send('OK')
                        else:
                            connection.send('AUTH ERROR')
                            sys.exit('Autenticação num superbloco falhou!')
                        outputs.write(text)
                    print('decrypted:')
                    print(outputs.getvalue())
                except Exception as Err:
                    print('Erro no recetor! {0}'.format(Err))

                outputs.close()
            else:
                message = bytes("ERROR IN KEY CHECK",'utf8')
                signature = receiver_ecdsa_sk.sign(
                    message,
                    ec.ECDSA(hashes.SHA256())
                )
                connection.send({'msg':message, 'sig': signature})
            connection.close()
```

```
In [12]: BiConnection(Emitter,Receiver).auto()
```

bytes before cipher:

b'L\x19\x9c.\xe3\xe4\xb5b\x0b\x1d+\x1b\x81\x7f\xf2T\xb0CP%T\xda\x96\x0c\x82^\xeb\xdd]p\xe1
\x8a\xcaX\xfd9x\xfa\xd1\x1b\xcc\xa1T\x83\x99$\xa3\xecv\xf9\xb5d\xc4\x90\x10\xa4M"p\x87\x12\xb
PB\x8a\xc9\x9bmJ\xd72\x1e\xb2$\x87\x0f\xe1\xdcU!k\x13\xfbL\x9f\x9bx\x8eQ\xef\xb1J\x18\x809\xd
\xcc\x96\x17A8\xecW\r\xab\xd4\x00\xc8\xf0\xc6D\x16\xd2Z\xca-
\xe8%\xa2\x10\xfc;)\xabq\x88\xba %%\x0c\r6\xe5\xa4\xd3}\xf9SA\xdf\xf00
A\xe9\x00\xf5+\x0bb@\xd4\xf5\xdd\x95.\xb5\xc3\xfa+\x91\xe2\xa32\xf4\n\x17\xfc\x8d*\xd3\x08\xe
\xa7\t_\xe6\xccSa\xbc\x9a\xbb/\xbd\x885\xc4\xc5\xf7.\x8cJ7\x11\xda;\xbf\x1a\xeb\xf5\x05\x90.\x1d
\x1f\xd31\xa34 \xb9\xd0\xe3\xf5\xa9a\xa5\xc7aF\xcd\x0b\x82\xa5\x1aS\x1eP\xfa\x03Z\xe7\x88\xdc~
}cUp\xfa\x850\xec\xb5x\xe7U\nO\xb6<\x05c\xf3\xa3\x14*\x02\xba\xff\'\xdf\xf3\xf3\xdf\xf1\x88o\xc
\x90t\xde\xba\xaf\x06\x83\r\x85i\xec\\\xc9\xa2s\xdd\xbcM\n\xc5B?\xf5\x80x\xd3?x\x85\xe6\xa7o\x
\xcb\xf9!\xe0R\xcck\x9c\xdc\x88\xb1\xdb\xab\xdeX{\xa9\x9b\xfe\xe7\x96\xa73l\xe9\xde-
\x16\xac\x17\xdb\xff_\xf3\x8f\xde\xde\xb1\xc6\x96Q\xad\xd4XK\x89\xe8\xb9\xd9\xdb\x8dSGM\xbf
K\x9d\x9d=\xb9\xf9\xc8'\x1a\xd2\xbf^\x82\xcf\x02Y\x88\x8b|\xf0\x1b\xaf\xaf\xea\xed\x17L\xd3\xe

decrypted:

b'L\x19\x9c.\xe3\xe4\xb5b\x0b\x1d+\x1b\x81\x7f\xf2T\xb0CP%T\xda\x96\x0c\x82^\xeb\xdd]p\xe1
\x8a\xcaX\xfd9x\xfa\xd1\x1b\xcc\xa1T\x83\x99$\xa3\xecv\xf9\xb5d\xc4\x90\x10\xa4M"p\x87\x12\xb
PB\x8a\xc9\x9bmJ\xd72\x1e\xb2$\x87\x0f\xe1\xdcU!k\x13\xfbL\x9f\x9bx\x8eQ\xef\xb1J\x18\x809\xd
\xcc\x96\x17A8\xecW\r\xab\xd4\x00\xc8\xf0\xc6D\x16\xd2Z\xca-
\xe8%\xa2\x10\xfc;)\xabq\x88\xba %%\x0c\r6\xe5\xa4\xd3}\xf9SA\xdf\xf00

A\xe9\x00\xf5+\x0bb@\xd4\xf5\xdd\x95.\xb5\xc3\xfa+\x91\xe2\xa32\xf4\n\x17\xfc\x8d*\xd3\x08\xe
\xa7\t_\xe6\xccSa\xbc\x9a\xbb/\xbd\x885\xc4\xc5\xf7.\x8cJ7\x11\xda;\xbf\x1a\xeb\xf5\x05\x90.\x1d
\x1f\xd31\xa34 \xb9\xd0\xe3\xf5\xa9a\xa5\xc7aF\xcd\x0b\x82\xa5\x1aS\x1eP\xfa\x03Z\xe7\x88\xdc~
}cUp\xfa\x850\xec\xb5x\xe7U\nO\xb6<\x05c\xf3\xa3\x14*\x02\xba\xff\'\xdf\xf3\xf3\xdf\xf1\x88o\xc
\x90t\xde\xba\xaf\x06\x83\r\x85i\xec\\\xc9\xa2s\xdd\xbcM\n\xc5B?\xf5\x80x\xd3?x\x85\xe6\xa7o\x
\xcb\xf9!\xe0R\xcck\x9c\xdc\x88\xb1\xdb\xab\xdeX{\xa9\x9b\xfe\xe7\x96\xa73l\xe9\xde-
\x16\xac\x17\xdb\xff_\xf3\x8f\xde\xde\xb1\xc6\x96Q\xad\xd4XK\x89\xe8\xb9\xd9\xdb\x8dSGM\xbf
K\x9d\x9d=\xb9\xf9\xc8'\x1a\xd2\xbf^\x82\xcf\x02Y\x88\x8b|\xf0\x1b\xaf\xaf\xea\xed\x17L\xd3\xe

A comparação entre os valores dados por **bytes before cipher** e por **decrypted** no output, por via da igualdade, permitem concluir que a mensagem que foi gerada é a mesma mensagem que foi decifrada.

5 Conclusão

Os resultados da realização deste trabalho prático são, na nossa opinião, muito satisfatórios visto que fomos capazes de cumprir com todos os objetivos propostos ao implementar duas sessões síncronas seguras de troca de informação entre agentes, uma utilizando o protocolo **DH** e o algoritmo de assinaturas **DSA** e outra utilizando o protocolo **ECDH** e o algoritmo de assinaturas **ECDSA**. Além disso, a apresentação dos resultados (código e explicação) está feita de tal forma que seja mais simples entender o que foi a estratégia do grupo para a resolução dos problemas propostos.

No processo de resolução deste trabalho prático, o grupo deparou-se apenas com duas dificuldades que foram, eventualmente, ultrapassadas com mais ou menos esforço. Uma delas diz respeito à implementação do **TAES** devido ao facto de a primitiva ter que ser efetivamente implementada antes de a utilizar o que, no início, gerou um bocado de confusão em relação ao que devia ser feito. A segunda dificuldade esteve relacionada com o facto da especificação da sessão síncrona, visto que, esta foi a primeira vez que o grupo implementou algo do género, pelo que as características deste tipo de sessão tiveram que ser bem estabelecidades e diferenciadas em relação ao tipo de sessão que estávamos habituados a implementar (sessão assíncrona).

6 Referências

1. Worksheets TP2 fornecidas pelo professor
2. Cryptography - Elliptic Curve cryptography
3. Cryptography - Diffie-Hellman Key Exchange
4. Tweakable Block Ciphers - paper by Liskov, Rivest and Wagner
5. Tweakable Block Ciphers - Notas manuscritas do professor