EDES - Enhanced DES
Applied Cryptography, Project 1, 2022/2023
Universidade de Aveiro

Name: Camila Fonseca
Nmec: 97880

Index

Note: The code present in this report is from the Python implementation of EDES, however, they are fully compatible with one another.

Quick Start

How to run the included code

The scripts accept input through stdin - usage of pipes is recommended. Python requires the libraries specified in *requirements.txt*.

Python

Encrypt (EDES):

```
<content to encrypt> | python3 encrypt.py <key>
```

Decrypt (EDES):

```
<content to decrypt> | python3 decrypt.py <key>
```

By default, the encrypted data is stored in the “encrypted” file. To quickly check the code is working, the following command can be used:

```
cat encrypted | python3 decrypt.py <key>
```

Encrypt (DES):

```
<content to encrypt> | python3 encrypt.py <key> -des
```

Decrypt (DES):

```
<content to decrypt> | python3 decrypt.py <key> -des
```

Speed:

```
python3 speed.py -des/-edes
```

Note: Both *-des* and *-edes* arguments can be used at the same time to measure both algorithms at once.

Go

Encrypt (EDES):

```
<content to encrypt> | ./edes -e <key>
```

Decrypt (EDES):

```
<content to decrypt> |  ./edes -d <key>
```

By default, the encrypted data is stored in the “encrypted” file. To quickly check the code is working, the following command can be used:

```
cat encrypted |  ./edes -d <key>
```

Speed:

```
./edes -s des/edes
```

Note: Both *des* and *edes* arguments can be used at the same time to measure both algorithms at once.

Implementation

Encryption

The scripts accept a variable-length textual password, which is then derived into a **256-bit** (32 bytes) digest using the SHA-256 algorithm, implemented by external libraries (*cryptography* in python and *crypto/sha256* in Go). This 32-byte key is then used to either generate or retrieve the S-Boxes, which will be explained further ahead.

EDES is a block cipher, processing an 8-byte block at a time. The last block is always padded (or just padding), using PKCS#5 to do so - The number of missing bytes to a block is added to a block until it is 8-byte long. If there's no bytes missing, an entire block of padding is added instead. - The values added as padding are the **binary** representation of the numbers themselves and **not** the ascii encoded characters which would represent the number. (0x08 instead of 0x56, for "8", for example)

To unpad, the last byte of the decoded plaintext is read, and then a number of characters equal to its value is removed from the plaintext.

Each block goes through 16 rounds in a Feistel Network. On each round, the only thing that differs is the S-Box used. Since there's no error propagation or any kind of diffusion, each block's processing could be parallelized, there's no need for it to be sequential.

Feistel Network

Each block is split into two halves, *left* and *right*.

Output Left block is an unaltered copy of Input right.

Output Right block is the Input right block after being shuffled by the S-Box, XOR-ed with the Input Left Block. Below is the code snippet in python for this operation:

Note: the zip() function iterates and returns the first element of list1 together with the first element of list2, second element of list1 together with second element of list2 and so on.

```python
def feistel_round(block: bytes, sbox: list) -> bytes:
    (...)
    left = block[:4]
    tmp = left
    right = block[4:8]

    outp = shuffle(right, sbox)
    left = right
    right = bytes([a ^ b for a, b in zip(outp, tmp)])

    return left+right
```

S-Box Shuffle

4-byte blocks are shuffled at a time, since the input is half a block. Before the s-box shuffling takes place, there are some operations done over the bytes to introduce some additional entropy in the process.
(in0 refers to the first byte of the input, and so on)

```
def shuffle(inp: bytes, sbox: list) -> bytes:
(...)
out0 = (in0+in1+in2+in3) % 256
out1 = (in0+in1+in2) % 256
out2 = (in0+in1) % 256
out3 = in0

out0 = sbox[out0]
out1 = sbox[out1]
out2 = sbox[out2]
out3 = sbox[out3]
```

Decryption

The decryption process is mostly the same as the encryption, only inverted. The sboxes used for decryption must be the same for encryption, which means the key used must be the same.The order of the sboxes is also inverted, meaning that the last sbox is used first, and so on, and the padding is processed (removed) at the end.

Feistel Network

Below is the code that makes up a single round for one block on the decrypting phase of the feistel network. It works mostly the same, but it is to note that the names are the opposite, in order to reverse the operation previously done when encrypting.
The ‘unshuffle’ function is exactly the same as the ‘shuffle’ function shown previously.

```
right = block[:4]
left = block[4:8]
tmp = left

outp = unshuffle(right,sbox)
left = right
right = bytes([a^b for a,b in zip(outp,tmp)])

return right+left
```

S-Box Generation

S-boxes depend only on the key.

Keeping in mind that we start off already with a 32-byte key, it is derived using **Shake256** into a **256-byte** digest. Then, it is split into an array, where each element of the array is one byte of the digest - from 0x00 to 0xFF.

```
digest = hashes.Hash(hashes.SHAKE256(256))
digest.update(key)
seed = digest.finalize()
seedbox=[]

for ch in seed:
    seedbox.append(ch)
```

However, this cannot be used straight as it is since we have no assurance that the bytes are unique, and for the S-Box to work as intended (Well, for decryption to be possible.) there must be no repeat bytes in the array. For this, we instead generate a list with every byte from 0x00 to 0xFF, totaling 256 bytes.

```
box = [str(i).encode('utf-8') for i in range(256)]
```

Our objective here is to shuffle this ‘box’, in a deterministic manner using only the previous hash/array, which will be referred to as **seedbox**. For this, the bytes in the seedbox are converted into their numeric values (0x0 to 0, 0xA to 10, 0x11 to 17…), and then operated over in the following manner, to obtain a value between 0 and 255:

```
seedbox=[(seedbox[i]+i)%len(seedbox) for i in range(len(seedbox))]
```

Each value in *seedbox*, with a given index, is the position where the value with that same index in *box* will be moved to.

```
shuffle_pairs = [(x,y) for x,y in zip(seedbox, box)]

shuffle_pairs.sort(key=lambda y: y[0])
box = [y for x,y in shuffle_pairs]
```

And thus we have an S-Box generated! Between every box generation, the key is hashed again to provide a different 256-byte digest.

```
key = key + b'\x01'
key = keygen(key,32)
```

S-Box Retrieval

To be able to reuse sboxes between operations, we need to give the file that contains them an unique identifier (filename), yet not be able to get any information about the key used to generate it from the file name.
For this, the key is “salted” via a simple operation, and then hashed using SHA-256.

```
def salt_key(key:bytes) -> bytes:
   salt=0
   for b in key:
       salt = salt + b
   keycopy = key+str(salt).encode('utf-8')

   a= keygen(keycopy,32)
   return a
```

S-Boxes are stored as hexadecimal values separated by commas, so when retrieving them it’s necessary to convert from the hex string into actual byte values.

Speed Measurements (Results)

	Python	**Go**
EDES	0.019103050231933594	0.010493
DES	0.0019078254699707031	0.000091

- All values are presented in **seconds**
- Results obtained by measuring each algorithm in each language 100000 times, and taking the fastest time for each.

EDES could be *significantly* sped up in Go by using parallelization, but as it stands, it still is almost as twice as fast as python.