Relatório Trabalho 2 - Implementação AES

Aluno: Felipe Renato Alvarenga Batalha (19/0012862) Turma 2

6 de Novembro de 2023

1 Parte I - Implementação do AES (bloco de 128 bits)

A primeira parte do trabalho consistiu na elaboração do AES de 128 bits, e contemplou os aspectos de cifração e decifração. A implementação se deu na linguagem *Python* e é necessário o uso de algumas bibliotecas, como *numpy* para lidar com os blocos na forma de matrizes e *pytest* para rodar alguns testes no mesmo, sendo necessário instalar ambos via *pip* contidas no arquivo *requirements.txt* do projeto. A seguir, a implementação da principal função que coordena a criptografia:

```
1  def aes_encryption(rounds, state, key):
2
3      keys = key_expansion(key, rounds)
4      state = add_round_key(state,key)
5
6      for round in range(1,rounds+1):
7          state = sub_bytes(state)
8          state = shift_rows(state)
9          if round < rounds:
10             state = mix_columns(state)
11         state = add_round_key(state,keys[round])
12     return hex_to_string(state.transpose())
```

A decifração efetuou o inverso de cada passo na ordem contrária, conforme a seguir:

```
1 def aes_decryption(rounds, state, key):
2
3     keys = key_expansion(key, rounds)
4
5     keys.reverse()
6
7     for round in range(0,rounds):
8         state =
             undo_add_round_key(state,keys[round])
9         if round > 0:
10             state = undo_mix_columns(state)
11        state = undo_shift_rows(state)
12        state = undo_sub_bytes(state)
13    state = undo_add_round_key(state,keys[-1])
14    return hex_to_string(state.transpose())
```

A validação conforme mencionado anteriormente se deu através do *pytest*, e os testes podem ser verificados rodando o arquivo *test-aes.py* disponibilizado, com o comando: ***pytest test-aes.py***. Nele, alguns exemplos de vetores de teste foram utilizados como referência.

2 Parte 2 - Modo CTR

O modo CTR foi implementado utilizando o modo de criptografia implementado na parte 1, e pode ser executado sobre um arquivo iterando sobre cada 16 bytes do mesmo fazendo a utilização do *nonce*, obtido de um *iv* e um contador iniciado em zero. A decifração e a cifração foram similares, e curiosamente, a decifração no modo CTR não necessitou da utilização do algoritmo de decifração da parte 1, já que para decifrar em modo CTR, bastar repetir os passos da cifração com relação ao *nonce* e realizar o XOR entre o arquivo cifrado e o *keystream*. Abaixo, implementação da cifração, onde *aes-encryption* é função implementada na parte 1:

```
1 def encrypt(rounds, input, output, key, iv):
2     size = os.path.getsize(input_file)
3     with open(input, 'rb') as file_in, open(output,
         'wb') as file_out:
```

```
4           counter = 0
5           encrypted_data = bytearray()
6           while True:
7               data = file_in.read(16)
8               os.system('cls' if os.name == 'nt' else
                    'clear')
9               print(f"Cifrando: {(((counter *
                    16)/size) * 100):.2f}%")
10
11              if len(data) == 0:
12                  break
13
14              # Create the unique nonce
15              nonce = iv + counter.to_bytes(8,
                    byteorder='big')
16              keystream = aes_encryption(rounds,
                    string_to_hex(nonce), key)
17              ciphertext = bytes(x ^ y for x, y in
                    zip(keystream, data))
18              encrypted_data += ciphertext
19              counter += 1
20          file_out.write(encrypted_data)
```

Para validação, foi utilizado o *openssl* conforme sugerido na especificação do trabalho, e a comparação entre os *outputs* do mesmo e o da implementação realizada foi realizada utilizando o comando *diff*. Uma coisa importante a salientar, é que o IV não é armazenado no arquivo criptografado no *openssl*, sendo assim, o mesmo assim como a chave, é solicitado ao rodar o programa, na cifração e decifração. O comando utilizado para criptografar os arquivos foi o seguinte:

```
openssl enc -aes-128-ctr -in arquivo -out encrypted-openssl.bin -K "$hex_key" -iv "$hex_iv"
```

É importante ressaltar que no openssl, a chave passada e o iv devem estar na forma hexadecimal, enquanto na implementação desse trabalho, deve ser gerado em plaintext. Sendo assim, o *hex-key* e *hex-iv* utilizados acima podem ser gerados através de variáveis *key* e *iv* conforme exemplo a seguir (exemplo baseado no Linux, instanciar as variáveis pode diferir no Windows):

```
key="Thats my Kung Fu"
iv="12345678"
hex_key=$(echo -n "$key" | xxd -p -c 32
hex_iv=$(echo -n "$iv" | xxd -p -c 32)}
```

Para rodar o arquivo *ctr-mode.py* referente a implementação do modo CTR, basta passar os parâmetros a seguir, se aproveitando da *key* e do *iv* previamente instanciados:

```
python3 ctr-mode.py cifrar arquivo encrypted.bin key iv
```

A decifração é igual o detalhado acima, porém seria necessário substituir o *cifrar* por *decifrar*, e utilizar um arquivo cifrado no lugar de *arquivo* e no output, se atentar para a extensão do arquivo pretendido. E finalmente, a comparação foi feita entre o arquivo gerado pelo *openssl* e o arquivo gerado pelo *ctr-mode.py*:

```
diff -s encrypted.bin encrypted-openssl.bin
```

O *output* do comando acima deve indicar que os arquivos são idênticos, validando a corretude da implementação. Atenção: ao testar a decifração, basta incluir o -d no *openssl*, e se atentar para a extensão do arquivo de *output* como mencionado anteriormente.

3 Conclusão

A realização do trabalho permitiu verificar empiricamente como se dá o funcionamento de um algoritmo de criptografia simétrica, e apresentou alguns desafios. Entre eles, o de uma implementação rápida com arquivos maiores como imagens.

A implementação realizada apresentou um comportamento mais lento do que o de programas como o *openssl*, possivelmente devido a linguagem de implementação ou devido ao uso de matrizes e operações de matrizes. A biblioteca *numpy* facilitou a checagem do funcionamento do algoritmo durante a elaboração, mas é possível que as operações com matrizes tenham sido computacionalmente custosas. Esse custo não ficou evidente nos testes com entradas de 128 bits antes da implementação do modo CTR para lidar com arquivos maiores, que podem vir a rodar a operação milhares de vezes.