Universidade de Aveiro
Mestrado em Cibersegurança
Responsible: André Zuquete

1st Project

Tuesday 31st October, 2023

Ricardo Covelo(102668) - Telmo Sauce(104428)

Contents

1 Introduction

Proposed by the instructors of the ”Criptografia Aplicada” course, this project entailed the implementation of a modified version of DES cryptography, namely E-DES. This variation requires fewer operations in the encryption process, resulting in faster processing. Additionally, E-DES has a 256-byte key, surpassing DES in key length.

All the results and the running process are located in the README.md file.

2 Key and Sbox generation

Provided with a password, the program will produce a 256-byte key. This key will then be utilized to generate all s-boxes. The key and the SHA-1 algorithm will be employed to carry out this task using the predetermined salt of /0x00’.

```python
def random_gen(pwd,len):
    if(isinstance(pwd, str)):
        pwd = pwd.encode()
    salt = b'\x00'
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA1(),
        length=len,
        iterations=1000,
        salt=salt,
    )
    key = kdf.derive( pwd )
    return key
```

```
PKCS5_PBKDF2_HMAC_SHA1(pwd.c_str(), pwd.length(), &salt, 1, 1000, sizeof (key_256), key_256);
```

Figure 1: Random generator function in C++ and Python

To generate the Sboxes, we employ a random generator to generate 4096 bytes from the 256-byte key. These 256 bytes are then arranged into 16 Sboxes, each containing 256 bytes.

```
static Sboxes sbox_gen(std::string pwd) {
    unsigned char salt = 0;
    uint8_t key_256[256];
    PKCS5_PBKDF2_HMAC_SHA1(pwd.c_str(), pwd.length(), &salt, 1, 1000, sizeof (key_256), key_256);
    uint8_t key_4096[4096];
    PKCS5_PBKDF2_HMAC_SHA1(reinterpret_cast<const char*>(key_256), sizeof(key_256), &salt, 1, 1000, sizeof(key_4096), key_4096);
    int arr_size = sizeof(key_4096) / sizeof(key_4096[0]);
    std::vector<unsigned char> vec(key_4096,key_4096 + arr_size);
    Sboxes box(vec);
    return box;
}
```

```python
def sboxes( bytes_variable):
    current_inner_array =[]
    sboxes_matrix=[]

    for i in range(0, len(bytes_variable)):
        byte_pair = bytes_variable[i]
        current_inner_array.append(byte_pair)

        # If the current inner array has 256 pairs (512 bytes), create a new one
        if len(current_inner_array) == 256:
            sboxes_matrix.append(current_inner_array)
            current_inner_array = []

    return sboxes_matrix
```

Figure 2: Sbox generation in C++ and Python

3 Encryption and Decryption

3.1 Padding

Following the professor's instructions, we incorporated PKCS-7 padding into our project. Nonetheless, we utilized the Cryptography library for implementing the padding in Python, while in C++, we developed our function, instead of resorting to a library. This padding was necessary when the encrypted text was not divisible by 64 bytes, to ensure seamless functioning of our code, irrespective of the size of the encrypted text.

```python
#Padder
binary_plaintext = plaintext.encode()
padder = padding.PKCS7(512).padder() # 64 bytes = 512 bit
binary_plaintext= padder.update(binary_plaintext)
binary_plaintext += padder.finalize()

#Unpadder
unpadder = padding.PKCS7(512).unpadder()
decrypted_text=unpadder.update(decrypted_text)
decrypted_text += unpadder.finalize()
decrypted_text = decrypted_text.decode()
```

Figure 3: Padding in python

```cpp
std::vector<unsigned char> addPaddingPKCS7(const std::vector<unsigned char>& input) {
    size_t block_size =64;
    size_t padding_value = block_size - (input.size() % block_size);
    std::vector<unsigned char> output = input;
    for (size_t i = 0; i < padding_value; i++) {
        output.push_back(static_cast<unsigned char>(padding_value));
    }
    return output;
}

// Function to remove PKCS#7 padding from a block
std::vector<unsigned char> removePaddingPKCS7(const std::vector<unsigned char>& input) {
    if (input.empty()) {
        throw std::runtime_error("Input vector is empty.");
    }
    unsigned char last_byte = input.back();
    if (last_byte > input.size()) {
        throw std::runtime_error("Invalid padding.");
    }
    for (size_t i = input.size() - last_byte; i < input.size(); i++) {
        if (input[i] != last_byte) {
            throw std::runtime_error("Invalid padding.");
        }
    }
    return std::vector<unsigned char>(input.begin(), input.end() - last_byte);
}
```

Figure 4: Padding in C++

3.2 Text To Blocks

Before performing encryption, we broke down the padded text into blocks, each consisting of 64 bytes for EDES and 8 bytes for DES. These blocks will then be encrypted individually.

```python
# Transformar para blocos de 64 bytes
def bytesToArraybytes( bytesOriginal):
    blocksize=64
    byte_array=[]
    for i in range(0,len(bytesOriginal),blocksize):
        subarray=bytesOriginal[i:i+blocksize]
        byte_array.append(subarray)
    return byte_array
```

Figure 5: 64 byte block Function in python

```cpp
std::vector<std::vector<unsigned char>> splitIntoBlocks(const std::vector<unsigned char>& bytes, int blockSize) {
    std::vector<std::vector<unsigned char>> byteBlocks;

    for (size_t i = 0; i < bytes.size(); i += blockSize) {
        byteBlocks.push_back(std::vector<unsigned char>(bytes.begin() + i, bytes.begin() + std::min(i + blockSize, bytes.size())));
    }

    return byteBlocks;
}
```

Figure 6: 64 byte block Function in C++

3.3 Using the Sboxes

Upon receiving the 32-byte blocks, we proceed to divide them into an array consisting of 8 cells, with each cell comprising 4 bytes. Subsequently, we apply the substitution techniques as instructed by the professor to each cell.

```python
def sbox_result( sbox,plaintext):
    cyphertext=b''

    array=[]
    for i in  range(0,len(plaintext)):
        subarray=plaintext[i].to_bytes(1,byteorder="big")
        array.append(subarray)

    for i in range(0,len(array)-3,4):
        byte1=array[i]
        byte2=array[i+1]
        byte3=array[i+2]
        byte4=array[i+3]

        result1=sbox[int.from_bytes(byte4,byteorder='big')].to_bytes(1,byteorder="big")
        result2=sbox[(byte3[0]+byte4[0])%256].to_bytes(1,byteorder="big")
        result3=sbox[(byte2[0]+byte3[0]+byte4[0])%256].to_bytes(1,byteorder="big")
        result4=sbox[(byte1[0]+byte2[0]+byte3[0]+byte4[0])%256].to_bytes(1,byteorder="big")

        cyphertext+=result1+result2+result3+result4
    return cyphertext
```

Figure 7: Encrypting a 32-byte text with the Sboxes in Python

```cpp
static std::vector<unsigned char> sbox_result(std::vector<unsigned char> sbox, std::vector<unsigned char> plaintext) {
    std::vector<unsigned char> cyphertext;

    for (int i = 0; i < plaintext.size() - 3; i += 4) {
        unsigned char byte1 = plaintext[i];
        unsigned char byte2 = plaintext[i + 1];
        unsigned char byte3 = plaintext[i + 2];
        unsigned char byte4 = plaintext[i + 3];

        unsigned char res1 = byte4;
        unsigned char res2 = (byte3 + byte4);
        unsigned char res3 = (byte2 + byte3 + byte4);
        unsigned char res4 = (byte1 + byte2 + byte3 + byte4);

        unsigned char result1 = sbox[res1];
        unsigned char result2 = sbox[res2];
        unsigned char result3 = sbox[res3];
        unsigned char result4 = sbox[res4];

        cyphertext.push_back(result1);
        cyphertext.push_back(result2);
        cyphertext.push_back(result3);
        cyphertext.push_back(result4);
    }
    return cyphertext;
}
```

Figure 8: Encrypting a 32-byte text with the Sboxes in C++

3.4 Feistel networks

To encrypt data, we split the 64-byte blocks into two halves, with 32 bytes each. We call them Li and Ri 9 10. We then move Ri to the start and use it to help calculate a result from an S-box, then the result is used to XOR Li, which is moved to the last position 11. This process is repeated 16 times. To decrypt, we just reverse the process, however important to know that the Sbox since they are non-reverseble Sbox 13 12.

```python
#Separate L and R
midPoint = len(binary_plaintext) // 2
Li = binary_plaintext[:midPoint]
Ri = binary plaintext[midPoint:]
```

Figure 9: Divide into 32 bytes in python

```cpp
//Devide into 2 equal parts
int midPoint = 64/2;
std::vector<unsigned char> Li(bytes.begin(), bytes.begin() + midPoint);
std::vector<unsigned char> Ri(bytes.begin() + midPoint, bytes.end());
```

Figure 10: Divide into 32 bytes in C++

```cpp
for (int i = 0; i < 16; i++) {
    std::vector<unsigned char> box_result = Sboxes::sbox_result(box.sboxes[i], Ri);
    std::vector<unsigned char> xor_box_li;
    for (int f = 0; f < 32; f++) {
        xor_box_li.push_back(box_result[f] ^ Li[f]);
    }
    Li = Ri;
    Ri = xor_box_li;
}
```

```python
    for l in range(16):
        sbox_xor_li = bytes(x ^ y for x, y in zip(sbox_result(sboxes[l],Ri), Li))
        Li = Ri
        Ri = sbox_xor_li
```

Figure 11: Encription in C++ and Python

```python
for l in range(15, -1, -1):
    sbox_xor_ri = bytes(x ^ y for x, y in zip(sbox_result(sboxes[l],Li), Ri))
    Ri = Li
    Li = sbox_xor_ri
```

Figure 12: Decryption Operations in python

```cpp
for (int i = 15; i >= 0; i--) {
    std::vector<unsigned char> box_result = Sboxes::sbox_result(box.sboxes[i], Li);
    std::vector<unsigned char> xor_box_ri;
    for (int j = 0; j < box_result.size(); j++) {
        xor_box_ri.push_back(box_result[j] ^ Ri[j]);
    }
    Ri = Li;
    Li = xor_box_ri;
}
```

Figure 13: Decryption Operations in C++

4 Encryption and Decryption Applications

We began building these applications by receiving the key and the desired encryption type as arguments, with DES as the default option. The plain text is read through stdin and padded before any encryption operations are performed. Depending on the encryption type, we execute the necessary operations, using all the components mentioned above.

To ensure compatibility between the two different languages, we transformed the output of stdout into hexadecimal format 14 16, allowing for conversion back to bytes in the appropriate language format 17 15.

```python
sys.stdout.write(encrypted_text.hex())
```

Figure 14: To hex in python

```python
binary_string = sys.stdin.read()
encrypted_text = bytes.fromhex(binary_string)
```

Figure 15: From hex in python

```cpp
//Convert to hexadecimal
for (int i = 0; i < encrypted_text.size(); i++) {
    std::cout << std::hex << std::setfill('0') << std::setw(2) << (int)encrypted_text[i];
}
```

Figure 16: To hex in C++

```cpp
std::string hexInput;
std::vector<unsigned char> encrypted_text;
// Read hexadecimal input from stdin
while (std::cin >> hexInput) {
    // Convert the hexadecimal string to bytes
    std::vector<unsigned char> bytes = hexStringToBytes(hexInput);
    encrypted_text.insert(encrypted_text.end(), bytes.begin(), bytes.end());
}
```

Figure 17: From hex in C++

5 Speed

To implement the Speed App, a buffer of 4096 random values was created from ”/dev/urandom” and utilized as input 18. Encryption and decryption operations were then carried out, with timing only beginning after key generation, as it cannot be included in the final time. After we just iterated the operations 100000 times and got the fastest and average times 19. The clock_gettime function from Linux was utilized for time precision.

```python
# Fill page with random values
page_size = 4096  # 4KiB
buffer = bytearray(page_size)

# Open /dev/urandom
with open("/dev/urandom", "rb") as urandom:
    # Read random data into the buffer
    urandom.readinto(buffer)
```

```cpp
const int page_size = 4096; // 4KiB
std::vector<unsigned char> buffer(page_size);

// Open /dev/urandom
ifstream urandom("/dev/urandom", ios::binary);
urandom.read(reinterpret_cast<char*>(buffer.data()), page_size);
urandom.close();
```

Figure 18: Buffer of random values in Python and C++

```python
measurements = 100000
nano = 1000000000

# Perform the measurement for edes
fastest_time_edes = -1
average_edes = 0
for a in range(measurements):
    elapsed_time = measure_time_for_Edes()
    if fastest_time_edes > elapsed_time or fastest_time_edes == -1:
        fastest_time_edes = elapsed_time
    average_edes +=  elapsed_time
print(f"Fastest Time for Edes: {fastest_time_edes/nano} seconds")
print(f"Average for Edes: {(average_edes/measurements)/nano} seconds")

# Perform the measurement for des
fastest_time_des = -1
average_des = 0
for a in range(measurements):
    elapsed_time = measure_time_for_Des()
    if fastest_time_des > elapsed_time or fastest_time_des == -1:
        fastest_time_des = elapsed_time
    average_des +=  elapsed_time
print(f"Fastest Time for Des: {fastest_time_des/nano} seconds")
print(f"Average for Des: {(average_des/measurements)/nano} seconds")
```

```cpp
int main() {
    const int measurements = 100000;
    const long long nano = 1000000000;

    // Perform the measurement for Edes
    long long fastest_time_edes = -1;
    long long average_edes = 0;
    for (int a = 0; a < measurements; a++) {
        std::cout << "Measurement " << a << " Edes" << std::endl;
        long long elapsed_time = measure_time_for_Edes();
        if (fastest_time_edes > elapsed_time || fastest_time_edes == -1) {
            fastest_time_edes = elapsed_time;
        }
        average_edes += elapsed_time;
    }

    // Perform the measurement for Des
    long long fastest_time_des = -1;
    long long average_des = 0;
    for (int a = 0; a < measurements; a++) {
        std::cout << "Measurement " << a << " Des" << std::endl;
        long long elapsed_time = measure_time_for_Des();
        if (fastest_time_des > elapsed_time || fastest_time_des == -1) {
            fastest_time_des = elapsed_time;
        }
        average_des += elapsed_time;
    }

    std::cout << "Fastest Time for Edes: " << static_cast<double>(fastest_time_edes) / nano << " seconds" << std::endl;
    std::cout << "Average for Edes: " << static_cast<double>(average_edes / measurements) / nano << " seconds" << std::endl;

    std::cout << "Fastest Time for Des: " << static_cast<double>(fastest_time_des) / nano << " seconds" << std::endl;
    std::cout << "Average for Des: " << static_cast<double>(average_des / measurements) / nano << " seconds" << std::endl;

    return 0;
}
```

Figure 19: Interations in Python and C++